OHJ-2506
Program Verification
5 cr

Antti Valmari
Antero Kangas

Tampere University of Technology
Institute of Software Systems

P.O.Box 553, FIN-33101 Tampere

Tietotalo, room F 211

phone 3115 4390

email Antero.Kangas@tut.fi

web http://www.cs.tut.fi/~antero/

August 26, 2011

Course Bureaucracy

Newsgroup: tut.ot.ohjtod
Web: http://www.cs.tut.fi/kurssit/OHJ-2506
Accomplishment: by exercises

- informal grading system

Literature: lecture material and exercises

- no book covers the whole topics
- lecture material in English is available on the web page
- former lecture material in Finnish is sold in decreased form $4 \rightarrow 1$ by TiTe
- possible additional material is delivered by web
- suitable additional reading:
  - R.C. Backhouse: *Program Construction and Verification*, Prentice-Hall 1986
  - D. Gries: *The Science of Programming*, Springer-Verlag 1981
  - (D.Gries, F.B. Schneider: *A Logical Approach to Discrete Math*, Springer-Verlag 1993))
  - ((N.Francez: *Program Verification*, Addison-Wesley 1992))

Contents

1 Introduction 0-3
1.1 Why Program Verification is needed? . 0-5
1.2 Exactness and Formality 0-14
1.3 The Goal and Contents of the Course 0-31

2 Specification of a Program 0-33
2.1 State of a Program 0-35
2.2 Writing a State Predicate 0-47
2.3 Specification of a Program 0-67

3 Proving Correctness of a Program 0-84
3.1 General Principle 0-85
3.2 Semantics of Statements 0-95
3.3 Proving Loop Statements 0-118
3.4 Proof Examples 0-142

4 Applications for Proving 0-181
4.1 Proving Correctness as a Means for Review 0-182
4.2 Designing a Program and its Proof Together 0-199

5 Proving Algorithms 0-231
5.1 Using an Abstract Data Structure . . 0-233
5.2 Example of a Hard Proof of an Algorithm 0-248

6 Summary 0-270

1 Introduction

In this course we discuss about mathematical rigour in

- specification and
- verification

of parts of programs and algorithms.

> A proof or demonstration is said to be rigorous if the validity of each step and the connections between the steps is explicitly made clear in such a way that the result follows with certainty. "Rigorous" proofs often rely on the postulates and results of formal systems that are themselves considered rigorous under stated conditions.

The main interest in this course is on *semantics*, that is, mathematically rigorous treatment of meaning.

- *syntax*, that is, rules for spelling, is discussed on other courses
  - easier

The method for software engineering where both syntax and semantics are mathematical rigour is called *formal methods*.

⇒ in this course we actually do not discuss about formal methods

- after the course it is easy to understand and to apply formal methods

Formal methods have been developed for many tasks

- stating requirements, specification, design, development, verification, testing, etc.
- what you will be learning in this course helps mostly for the underlined tasks

In this section we

- justify the importance of program verification
- take a brief look to formal methods and their present state
- state the goal of the course

1.1 Why Program Verification is needed?

Example

- A student of professor R. Backhouse returned the following Pascal program, the task was to compare the equality of strings:

  $issame := (string1.length = string2.length);$
  if $issame$ **then**
  for $i := 1$ **to** $string1.length$ **do**
  $issame := string1.char[i] = string2.char[i];$
  write($issame$);

- Let us do some tests...
  - "`university`" "`university`" ⇝ True •/•
  - "`course`" "`course`" ⇝ True •/•
  - "" "" ⇝ True •/•
  - "`university`" "`course`" ⇝ False •/•
  - "`lecture`" "`course`" ⇝ False •/•
  - "`precision`" "`exactness`" ⇝ False •/•
- seems working!
- one more test:
  - "`true`" "`pure`" ⇝ True ?!?!?!
  - the program writes "True" if the strings have same length and same *last character*

  - the student answered:
    "But it worked last Tuesday!"

Had the student been careless while testing the program?

- He had run three series of tests:

  (a) strings with different length

  (b) strings with equal length but different strings

  (b) equal strings

  and the results were always correct

⇒ You cannot argue that the testing had been careless

How much the program should have been tested that the error had probably been revealed?

- starting point: the tester cannot suspect any specific string in advance
  - if he could, the error could be revealed immediately by reasoning
- the error appears only in the case (b)
- we calculate the probability for detection of the error for random (b)+(c) test material

- if the number of different characters is m and the length of the string is k, then the probability that the error will be detected in one test is

$$\frac{1}{m} - \frac{1}{m^k} < \frac{1}{m}$$

and in n tests it is below

$$1 - \left(1 - \frac{1}{m}\right)^n$$

- Let $m = |\{‘a', \ldots, ‘z'\}| = 26$
  - for detection probability ≥ 0.5, it is needed ≥ 18 tests
  - for detection probability ≥ 0.9, it is needed ≥ 59 tests

Remarks

- hardly no one have enough strength to test such a little program even 18 times
- if this part of program is for use, it is pretty sure that it will be ran at least 59 times
- for bigger programs the probability to detect an error in testing and the probability that it appears in production run are usually smaller

- the ratio of testing time and production time is usually very small
  - ⇒ the probability that the error appears in production run is much bigger than in testing
- the behaviour of a program may change when the program is transferred from testing environment to production environment
  - different computer and operating system
  - different load
- the behaviour of a program cannot be extrapolated reliable
  - cf. if a bridge can stand 10 tons, it stands all lesser loads
  - ⇒ in bridge design you can use safety factors: just use a thicker girder
  - what would be the safety factor for a logical information? maybe 20 % longer phone numbers?

⇒ *if the functioning of a program is ensured only by random tests then it is almost sure that in production run there will appear failures*

- an experienced programmer sees the above error direct by reading code, without tests

⇒ reasoning is often much efficient than testing

- conclusion:

  blind testing alone is not sufficient, the functionality must be investigated also by reasoning

A counterargument:

"a careful program tester does not use pure random material, but he tries to ensure that every branch of the program will be tested"

- it is often difficult or laborious to force the program to execute its every branch
  - for instance the code that processes a fault

  ⇒ some parts of the code are easily left without testing
  - what kind of results you have got from coverage analysis?
- the student of Backhouse executed every branch
  - the partition (a) - (b) - (c)
  - the test material executed every branch of the program

⇒ execution of every branch does not guarantee total coverage

- to become in some extent sure of the coverage of testing, the functionality of the program must *also* be reasoned

A counterargument:

> "mathematical rigour is too difficult and expensive"

- most of this is true
  - ⇒ it pays off to use it strictly only when quality and reliability are especially important
- it is mostly a question of training
  - a big one!
- the techniques of the course can and are worth to be applied in different levels of preciness according the situation
- very precise in the critical parts of a program
- informally in normal programming
- in the same way as mathematicians do in algorithm design

> *to know how to apply suitably inexact, one must know how to apply very exact*

In any case while you are programming you are reasoning at least a little

- even reviewing is based on reasoning and it is perceived to be very efficient
- now you can learn how to reason more, more rigorously, and more systematically

Challenges for programming: can you make these work reliable and fast without the means of the course?

- finding the largest 0-square in a matrix whose elements are either 0 or 1

0	1	1	0	0	0	1
0	0	1	0	1	1	0
1	1	0	0	0	1	1
1	0	0	0	0	1	0
0	1	0	0	0	0	0
0	0	0	1	0	0	0

- removing elements so that the remaining elements form a proper increasing sequence that is as long as possible
  - [3,1,6,1,9,4,5,8,1,2,4,0] $\leadsto$ [–,1,–,–,–,4,5,8,–,–,–,–]
- making a calculator for rational numbers
  - $[2/5] * ([2/3] + 4[1/3]) \leadsto 2[2/5]$

- finding the fastest train connection when departure time can be anything

  $\Rightarrow$ waiting on departure and arrival stations is not counted in but waiting on change stations is counted in

- sharing a resource (e.g. a printer) between several clients so that in any time at most one client has the resource and every client that has asked her turn eventually gets it

Mathematical rigour does not necessary mean hard to understand:

- A chess board consists of $8 \times 8 = 64$ squares. We have domino pieces that cover exactly two squares. Using these pieces the chess board can easily be covered so that no part of any piece is over the board and the pieces do not overlap. Next we remove the two opposite corner squares. Is it still possible to cover the board using the same kind of pieces? Justify your answer.

- My son used to play this game when he was at elementary school. The beginner says any number between 1, ..., 9. After that every player on his turn adds 1 or 2. The winner is the player who reaches 21. Is there a winning strategy? Justify your answer.

- Two containers, A and B, both have exactly the same amount of liqued: A has water and B 100% alcohol (we assume that it is possible to produce 100 % alcohol and it does not absorb water from air, actually it is important only that they are different liqueds).

  We execute the following procedure exactly in the following order: Take from container A some amount of liqued and pour it to B (the containers have room enough). Then take exactly the same amount of liqued from B and pour it to A. By "*purity*" we mean how much original liqued a container has.

  The question is "which container is purer after the previous process". Justify your answer.

1.2 Exactness and Formality

What does the term “formal” mean?

- dictionary: “formal” = (among other things) relating to the form or structure of something, ceremonial, official, organized, well-structured and planned, etc.
- in mathematics: independent of the meaning
  - formal calculation is only applying rules to formulas, a kind of game of marks
  - we do not think about the contents of a formula: it is allowed to calculate by the rule even it would lead to an obviously “incorrect” result
  - even a clearly “correct” calculation is forbidden unless there is a rule that allows it

$\Rightarrow$ to avoid stupidities the used collection of rules must be carefully planned and used

  - the rules must be very exact
  - they must be obeyed very exactly
  - the rules must have been chosen very carefully

- for software engineering
  - *formal* = mathematically exact
  - a *formal method* = a method used in software engineering where the target is expressed and the operations for it are implemented mathematically exact.

Formal methods have been developed for software engineering for following purposes:

- specification of a system, a program, a part of program, a data structure, how to express information, a protocol, etc.
- how to derive the implementation (e.g. of a program) from its specification
- modifying a program (e.g. to more efficient) without changing its meaning
- analysing the (logical) properties of a program
- proving the correctness of a program
- creating test material
- automating testing

In this course we cover specification and proving the correctness of parts of programs and algorithms. Because the rules of a formal method is applied without thinking their meaning, applying a single rule is mechanical

- cf. making a move in chess
- requires great exactness

⇒ poor for a human being, easy for a computer

A collection of formal rules does not define what rule and for what purpose should be applied next in order to get to the target

- cf. choosing a move in chess
- cf. reduction in mathematics: $a(b+c) = ab+cd$

⇒ choosing the rule leaves space for creativity

- it is a hard difficulty for automating formal methods of software engineering

⇒ altogether formal methods are hard for both human beings and computers

The basis of all the other formal methods is formal specification

- specification has two parts:
  - the *syntax*: which texts, formulas, etc. do mean anything at all
  - the *semantics*: what the syntactically correct texts etc. do mean
- either can be formal or informal

  - e.g. the syntax of a programming language is usually specified formally and the semantics informally
  - e.g. for mathematical notations the situation is usually contrawise

- use of computers requires that at least the syntax is formally specified (at least almost. . .)

Good means for formal specification of syntax were invented already in 1960s

- soon also good algorithms for many tasks dealing with syntax were found
  - especially for parsing

⇒ pure formal processing of syntax is not anymore called "formal method"

- therefore the term "formal method" requires that both the syntax and the semantics are formally specified
  - it is another story what the salesmen are telling

- the formalisms of syntax are discussed on the course OHJ-2100 Ohjelmistotieteen perustyökaluja (Basic Tools for Software Science)

Formal processing of semantics is hard and under eager research

- the target of this course is on the levels of code, algorithm, and specification
- semantics of parallel and concurrent programs is discussed e.g. on course OHJ-2606 State Machines
- specification level is discussed e.g. on seminars of Z, VDM, and B

Despite the semantics in this course is formal, it, nor programs, specifications, etc. are not presented using formal syntax

$\Rightarrow$ this is not a course of formal methods

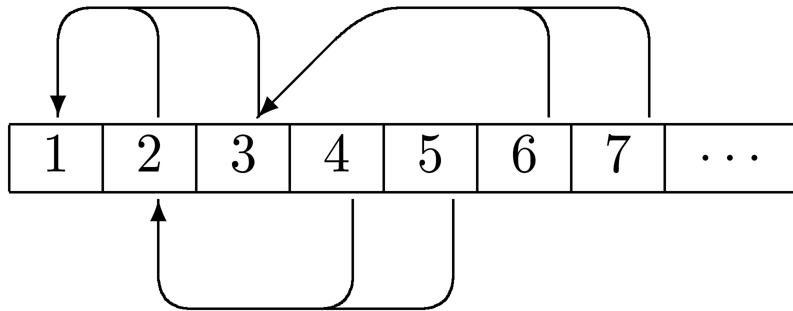
- we keep the freedom of mathematicians
- for semantics we do have a lot of common with formal methods

The term *semiformal* means same as informal

- compare to half-truth
- **it does not hold** formal = computer readable

A simple example of a formal modification of a program

- array $A[1, \ldots, n]$ is *heap* iff $A[\,\lfloor \frac{i}{2} \rfloor\,] \geq A[i]$, when $2 \leq i \leq n$
  - $\lfloor \frac{i}{2} \rfloor$ is $\frac{i}{2}$ rounded down (= truncated) to the nearest integer



- the following algorithm transforms A to heap

for $i := \lfloor \frac{n}{2} \rfloor$ **downto** 1 **do**
  $j := i$; $x := A[i]$;
  repeat
    $old := j$; $j := 2 \cdot j$;
    if $j < n$ and $A[j+1] > A[j]$ **then**
      $j := j + 1$ **endif**;
    if $j \leq n$ and $A[j] > x$ **then**
      $A[old] := A[j]$ **endif**
  until $j > n$ or $A[j] \leq x$;
  $A[old] := x$
endfor

- task: the definition of heap and the algorithm must be modified so that the table is indexed $0, \ldots, n-1$, just like in C++

- shall we first try to modify the program without the following advices?

1. modification of the definition of heap

- append "−1" to every indexing without changing anything else

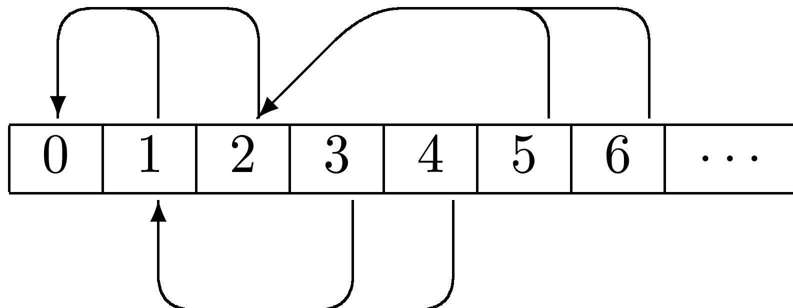
  ⇒ the indexing range shifts as we want
  $A[\,\lfloor \frac{i}{2} \rfloor - 1\,] \geq A[i-1]$, when $2 \leq i \leq n$

- substitute every i by $(i+1)$ in everywhere

  ⇒ reduction becomes possible
  $A[\,\lfloor \frac{(i+1)}{2} \rfloor - 1\,] \geq A[(i+1)-1]$,
  when $2 \leq (i+1) \leq n$

- reduce

  $A[\,\lfloor \frac{i-1}{2} \rfloor\,] \geq A[i]$, when $1 \leq i \leq n-1$



2. modification of the algorithm

- append "-1" to every indexing without changing anything else

for $i := \lfloor \frac{n}{2} \rfloor$ **downto** 1 **do**
  $j := i$; $x := A[i-1]$;
  repeat
    $old := j$; $j := 2 \cdot j$;
    if $j < n$ and $A[j+1-1] > A[j-1]$ **then**
      $j := j+1$ **endif**;
    if $j \leq n$ and $A[j-1] > x$ **then**
      $A[old-1] := A[j-1]$ **endif**
  until $j > n$ or $A[j-1] \leq x$;
  $A[old-1] := x$
endfor

- substitute i, j, and *old* everywhere by "$i+1$" etc.

  ⇒ we get rid of many "-1"

  – there becomes some "uncode" but we are going to get rid of it

for $i+1 := \lfloor \frac{n}{2} \rfloor$ **downto** 1 **do**
  $j+1 := i+1; x := A[i+1-1]$;
  repeat
    $old+1 := j+1; j+1 := 2 \cdot (j+1)$;
    if $j+1 < n$ and
      $A[j+1+1-1] > A[j+1-1]$ **then**
      $j+1 := j+1+1$ **endif**;
    if $j+1 \leq n$ and $A[j+1-1] > x$ **then**
      $A[old+1-1] := A[j+1-1]$ **endif**
  until $j+1 > n$ or $A[j+1-1] \leq x$;
  $A[old+1-1] := x$
endfor

- reduce
  - assignments are treated as equations
  - $j+1 := 2 \cdot (j+1) \rightsquigarrow j := 2 \cdot (j+1) - 1 = 2 \cdot j + 1$
  - $j+1 \leq n \rightsquigarrow j < n$

```
for i := ⌊n/2⌋ − 1 downto 0 do
    j := i; x := A[i];
    repeat
        old := j; j := 2 · j + 1;
        if j < n − 1 and A[j + 1] > A[j] then
            j := j + 1 endif;
        if j < n and A[j] > x then
            A[old] := A[j] endif
    until j ≥ n or A[j] ≤ x;
    A[old] := x
endfor
```

- compare the result to the original program!

Benefits of exact and totally formal methods

- quality of specifications and other documents improves
  - uniqueness
  - exactness
  - understandability – at least sometimes

- the specifications become independent from the implementation
  - to "what" level instead "how" level
- if also the syntax is formalised then documents can be processed by computers
  - checking the internal consistency
  - simulation / animation
  - different kinds of ordinary and intelligent tests
  - proving correctness
- if a program is formally derived or its correctness is proved then it "definitely" fills its specification
  - think about: how definitely
- formal modification of a program allows e.g. increasing its speed without accidentally breaking its functionality
- testing intensifies and becomes more versatile

$\Rightarrow$ for programs

  - the design process becomes easier (or not)
  - maintainability improves
  - less errors

- it has been noticed that even a precise specification alone reduces significantly the amount of errors
- knowing the basic ideas of formal methods improves significantly also the quality of informal programming

Drawbacks

- laborious
  - or that is what the users feel in the beginning
  - extra work pays it back later – or not
- they require new kind of thinking and new skills
  - hard to learn (especially for those who already are used to other methods)
  - ⇒ few trained people are available
- the methods are not (yet) mature and stabilised
- not very much tools or support is available
- some of the methods are computationally hard (intractable)
  - carried out by computers the methods need a lot of computing capacity
    ⇒ sufficiently efficient tools are not available

  - carried out by human beings they require mathematical creativity
    $\Rightarrow$ not doable by a common engineer

Most promising applications

- security critical systems
  - nuclear plant process control, steering an aeroplane, hospital devices, ...
- systems that are difficult or laborious to repair
  - space probes, silicon chips, telecommunication protocols, ...
  - the cost of repair becomes terrible expensive
- reactive systems
  - e.g. control systems of mechatronics, multiuser systems, telecommunications
  - understanding and controlling concurrency has turned out to be extremely difficult (for human beings)
- parts of programs for libraries
  - execute in various environments
    $\Rightarrow$ testing is difficult

Totally formal methods have become common very slowly

- there are companies whose line of business is selling formal method supporting tools and/or service
  - Praxis, UK; Meta Software, USA; Formal Systems, UK; Telelogic, Sweden, ...
- there exist notable users
  - Inmos has used at least from 1989 formal methods in designing silicon chips
  - Lucent Bell Labs has a strong group that concentrates for automatic analysis of protocols
  - Intel awaked after Pentium floating point error, and is now a big user
  - Nokia has been interested in formal methods for concurrent systems
  - ...
- the instructions of authorities may now or in the future, direct or indirect (strict product liability), force to use of formal methods
  - the standard 00-55 (1991) of the ministry of defense in UK requires use of formal methods

- imposing applications have been reported
  - "Paris Métro Line 14: Some features of Line 14's train control system are run under the OpenVMS Operating System. Its control system is noted in the field of software engineering of critical systems because safety properties on some safety-critical parts of the systems were proved using the B-Method, a formal method."
- in spite of all their use is still quite small
- the topic is large, disconnected, and immature

Reasons for unsucces of formal methods

- formalising even the smallest details is necessary
- formalising the smallest details is very laborious but gives very little
  - in principle it is the way to versatile automatic processing
  - in practise only the automates that do only simple things are working, the others are not efficient enough
- programming errors are usually quite tolerated
  - no one notices if every 10,000th mobile phone bill is not charged

  - the customers are used to quite apparent errors

⇒ there is no reason to pay for removing errors

Mathematical rigour without formalising the syntax is a good compromise

- gives a big deal of the quality improvement promised by formal methods
- unessential details need not to be formalised
- the used formalism can be chosen in a flexible manner

Comparison to other fields

- Most fields of technology utilise efficiently mathematics or mathematical theories
  - construction engineering: strength of materials, properties of materials, ...
  - electronics: theory of electricity, convection (heat transfer), ...
  - ...
- fields of technology have usually started as handcraft and tradition
- little by little there has been developed theories for understanding important phenomenons

  - e.g. Maxwell's equations in electromagnetism

- there have been developed different kinds of calculation and other methods to make designing easier
  - e.g. the Laplace and Fourier transforms
- nowadays these methods have been implementd for computers, even so, that the user needs not to know very deeply how the methods work
  - e.g. numerical solving of partial differential equations
  - e.g. simulation softwares for elecronic circuits
  - e.g. statistical softwares

handcraft → *science* → *automation*

- Valmari: software engineering is ready to move from handcraft to science but is is not yet ready to be automated

⇒ software mathematics works but formal methods do not

The common know-how of software mathematics in Finland is weak cf. other countries.

1.3 The Goal and Contents of the Course

For the previously mentioned reasons

- it does not pay off to concentrate on any specific formal method
- it is profitable to concentrate on the things that are common in many formal methods
  - specification by logic and set theory
- it pays off to keep the flexibility of mathematics

Program verification

- requires specification using logic
- requires deduction

$\Rightarrow$ practices most of the basic skills of formal methods

- quite stabilised basis of theory

$\Rightarrow$ its contents hardly becomes outdated

$\Rightarrow$ the course concentrates on it

Contents

- state of a program and how to discuss about it
- repetition of basic logic and set theory
- discussion of properties of a part of a program by logic and set theory
- proving the correctness of a program
- applications of proving
- proving the correctness of an algorithm

2 Specification of a Program

In this section

- we learn how to discuss about the properties of a state of a program using state predicates
- we combine the specification of a program from state predicates
- we discuss how a state predicate and a specification can be made to express the appropriate things

Even the subject is state of a program, many of the presented issues hold also widerly

Literature

- the content of the section does not correspond any specific book but the followings may be useful:
  - R.C. Backhouse: *Program Construction and Verification*, Prentice-Hall 1986
  - D. Gries: *The Science of Programming*, Springer-Verlag 1981
- in the next book mathematics is teached in an abnormal – more suitable for computer scientists – way

– D.Gries, F.B. Schneider: *A Logical Approach to Discrete Math*, Springer-Verlag 1993

2.1 State of a Program

In this subsection

- we investicate the notion of state of a program
- we justify why it is reasonable to discuss about it using logics and set theory

Definition

> The *state of a program* = values of the variables + the location of execution

If it is not otherwise mentioned, in this course the type of variables and the set of used numbers is the set of integers. Not e.g. the set of real numbers or the set of 32-bit integers.
Examples: initial values, a program, and its all states

- initially $x = y = 0$

1: $x := 1;$
2: $y := 2$
3:

execution	x	y
1	0	0
2	1	0
3	1	2

- initially $x = 10$

```
1: while x > 1 do
2:     x := x div 2
3: endwhile
4:
```

exec.	1	2	3	1	2	3	1	2	3	1	4
x	10	10	5	5	5	2	2	2	1	1	1

- initially $i = -5$ and $A[1] = A[2] = A[3] = 0$

```
1: for i := 1 to 3 do
2:     A[i] := 2 · i
3: endfor
4:
```

exec.	1	2	3	1	2	3	1	2	3	1	4
i	-5	1	1	1	2	2	2	3	3	3	?
$A[1]$	0	0	2	2	2	2	2	2	2	2	2
$A[2]$	0	0	0	0	0	4	4	4	4	4	4
$A[3]$	0	0	0	0	0	0	0	0	6	6	6

State predicates

- listing values of all variables is often laborious and unnecessary
- we want to discuss of properties of states without specifying values of all variables

$\Rightarrow$ we introduce the notion *state predicate*

- we say that an expession is a *logical expression* if it is meant to be interpreted as a claim that either holds or not
- definition

  The *state predicate* = a logical expression which value depends on the values of variables

- examples
  - $x > 0$
  - $y = 2 \cdot x + 1$
  - $\forall i \;;\; 1 \leq i \leq n : A[i] = -1 \lor A[i] \geq x$

State predicates and truth values

- most programming languages have some way to express "yes" and "no"
  - it is needed among other case in the condition of **if** statement
- examples
  - Pascal: type Boolean: **false** = no, **true** = yes
  - C: `int` 0 = no, other `int` = yes
  - Lisp: Nil = no, other values = yes

- an expression of a programming language is called *truth valued* if its result is either “yes” or “no”
  - C: result “yes” = 1
  - Lisp: result “yes” = “T”
- a truth valued expression of a programming language **is not** state predicate
  - a truth valued expression is *in* the program
  - a state predicate is telling *about* the program but stands outside it

  $\Rightarrow$ they live in different worlds
- example

  1: $found := \mathsf{False};\ i := 1;$
  2: **while** $i \leq n \wedge \neg found$ **do**
  3: **if** $A[i] = key$ **then** $found := \mathsf{True}$
  4: **else** $i := i + 1$
  5: **endif**
  6: **endwhile**
  7:

  - on the line 2 “$i \leq n \wedge \neg found$” is truth valued expression
  - “$i \leq n \wedge (found = \mathsf{False})$” is state predicate and it holds in the beginning of line 3 but not in the beginning of line 7

- when wanted, a truth valued expression is easy and natural to *interpret* as a state predicate
  - an expression e interpreted as a state predicate means the same as the state predicate "$e =$ yes"

$\Rightarrow$ yes $\sim$ holds, and
no $\sim$ holds not

- an example: the state predicate "$i \leq n \wedge (found = \mathsf{False})$" can now be shortened to "$i \leq n \wedge \neg found$"

$\Rightarrow$ between a truth valued expression and a state predicate there is

  - a difference of principle
  - usually there is no need to care of the difference, and a truth valued expression can be interpreted as a special case of state predicate

State predicates and the location of execution

- usually a state predicate does not speak of the location of execution

$\Rightarrow$ does a state predicate hold or not depends in which location of the program it is examined

- important locations of interpretation

- "in the beginning" = just before the execution of the program has started
- "in the end" = after the execution has finnished normally
  (not e.g. after the program has crashed)
- "on line n" = **every time** when the program is ready to execute line n

- a state predicate that is meant to be interpreted in the end speaks something about the results of the program
  - the results remain in these variables
- a state predicate that is meant to be interpreted in the beginning is interpreted differently than the state predicates in other locations!
  - it expresses the *precondition* for values of variables
  - these variables contain the inputs of this part of program
- it is often handy to write the state predicate in to the location of the program where it is wanted to be interpreted
  - in curly brackets "{" and "}"
    (later also in to angle brackets "$\langle$" and "$\rangle$")

  - then the linenumbers become unnecessary

- an example

  if $x < 0$ **then**
  $\{\ x < 0\ \}$
  $x := -x$
  $\{\ x > 0\ \}$
  endif $\{\ x \geq 0\ \}$

  - if the **then** branch is not executed then in the end $x \geq 0$
  - if it is executed then in the end $x > 0$

  $\Rightarrow$ in every case in the end $x \geq 0$

- an example (all variables are integers)

```
 1:  { n ≥ 1 }
 2:  a := 1; y := n;
 3:  { a = 1 ∧ y = n ≥ 1 } (* corollary: a ≤ y *)
 4:  while a < y do
 5:    { a < y }
 6:    v := (a + y) div 2;
 7:    { a ≤ v < y }
 8:    if A[v] < key then
 9:        { v < y }
10:        a := v + 1
11:        { a ≤ y }
12:    else
13:        { a ≤ v }
14:        y := v
15:        { a ≤ y }
16:    endif
17:    { a ≤ y }
18:  endwhile
19:  { a = y }
```

  - requirement for inputs: $n \geq 1$
  - line 7 can be deduced from line 5 by using properties of mean and truncation
  - lines 9 and 13 follow from line 7
  - line 17 follows from lines 11 and 15

  - on the line 19 $a \geq y$ because of the condition on line 4; also $a \leq y$ because of lines 3 and 17; from these it follows $a = y$

Assertions

- an *assertion* is a statement written in the program that halts the program if the given condition does not hold
- assertions are **very** useful especially while tracing errors of complicated data structures and algorithms
- C++:

```
#include <assert>
   ...
assert ( i >= 0 && 1 < n );
```

- AV:
  - the parameters are evaluated in any case
  - ⇒ it is not the most efficient solution, but...
  - simple and informative
  - so small loss of efficiency is not worthwhile until the programming skills are really good

```
inline void check(
   bool cond, const char *v0,
   int i1 = 0, const char *v1 = 0,
   int i2 = 0, const char *v2 = 0
){
   if( !cond ){
      std::cout.flush();      // yes, cout!
      std::cerr << “Int. error \ n???  ”  << v0;
      if( v1 ){ std::cerr << i1 << v1; }
      if( v2 ){ std::cerr << i2 << v2; }
      std::cerr << std::endl; exit( 1 );
   }
}
```

- example: how to use it

1: *check*($n \geq 1$, “too small n”);
2: $a := 1; y := n;$
4: **while** $a < y$ **do**
6: $v := (a + y)$ div 2;
8: **if** $A[v] < key$ **then**
10: $a := v + 1$
12: **else**
14: $y := v$
16: **endif**
17: *check*($a \leq y$, “too big a”, a, “ ”, y,“”)
18: **endwhile**
19: *check*($a = y$, “$a \neq y$”, a, “ ”, y, “”)

- if the truth value of a state predicate is easy to compute then you can easily get an assertion from it

- checking of a more complex state predicate can be done in a subprogram that is called only sometimes
  - e.g. every 100th round of the main loop
  - for debugging its call frequuency can be increased
  - it can be left out in the final program e.g. by commenting it out or by `#ifdef`

- example: heap
  $\forall i \; ; \; 2 \leq i \leq n : A[\, \lfloor \frac{i}{2} \rfloor \,] \geq A[i]$

- example: bi-directional linked list
  $x\!\uparrow\!.\, next\!\uparrow\!.\, prev = x \wedge x\!\uparrow\!.\, prev\!\uparrow\!.\, next = x$

⇒ by using state predicates you can test programs

Later we shall calculate a lot of with state predicates

⇒ it is worth to repeat or reread rules for calculation, that is, the basics of logic in e.g. the lecture material of course OHJ-2100

For specification of parts of programs we need

- notations for terms of context

  - e.g. a sorting algorithm "<"

- set theory to discuss about data structures
- logic for connecting parts of definitions
  - propositional logic: logical operators and rules
  - predicate logic: variables of the context, quantifiers

For proving the correctness of a program we need

- rules for calculus of context
  - e.g.
    if $a < b$ and $b < c$ then $a < c$
- semantical theory of programming language
  - e.g.
    $\textbf{if } x < 0 \textbf{ then } x := -x \textbf{ endif } \{\ x \geq 0\ \}$
- logic for deduction

Semantics of programming language is discussed on Sections 3.2 and 3.3

2.2 Writing a State Predicate

An example: array $A[1 \ldots n]$ is in order

- why this does not work?

$$\mathit{ordered}(A[1 \ldots n]) :\Leftrightarrow$$
$$\forall i \;;\; 1 \leq i \leq n : A[i] \leq A[i+1]$$

- fixed version

$$\mathit{ordered}(A[1 \ldots n]) :\Leftrightarrow$$
$$\forall i \;;\; 1 \leq i \leq n-1 : A[i] \leq A[i+1]$$

- what extra requirement the following states?

$$\mathit{ordered2}(A[1 \ldots n]) :\Leftrightarrow$$
$$\forall i \;;\; 1 \leq i \leq n-1 : A[i] < A[i+1]$$

Definition notations $:\Leftrightarrow$ and $:=$

- they mean “is defined to mean”
  - $:\Leftrightarrow$ for predicates
  - $:=$ for others
- after definition $\mathit{symbol} := \mathit{expression}$ it holds that $\mathit{symbol} = \mathit{expression}$
  - similarly for $:\Leftrightarrow$
  - a definition makes also more than that it is an equation

  - $*$ it introduces a new symbol

$\Rightarrow$ it is good to have an own, asymmetric notation

- in literature you can see also $=_{\mathsf{def}}$, $\triangleq$, etc.
- $:\Leftrightarrow$ and $:=$ imitate assignment in programming languages

Example: selection of position in ordered array

- we assume $\mathit{ordered}(A[1\ldots n])$ and $n \geq 1$
- we want to define a predicate $\mathit{location}(A[1\ldots n], \mathit{key}, x)$ so that
  - if key is in the array then x is pointing to its location
  - otherwise x points to where element key "should be"
- x points in to array

  $\Rightarrow$ reasonable values for x are between $1 \leq x \leq n$

  $\Rightarrow$ it does not matter even the predicate is undefined elsewhere
- what "should be" means?
  - e.g. $\mathit{key} = 7$

3	5	6	9	11	12	18	18	22	23
1	2	3	4	5	6	7	8	9	10
		↑	↑						
		x?	x?						

- let us select "should be" $\sim$ "the nearest that is greater than"

- 1. try: $\mathit{location}(A[1 \ldots n], \mathit{key}, n) :\Leftrightarrow$
  $A[x] = \mathit{key} \vee (A[x-1] < \mathit{key} \wedge A[x] > \mathit{key})$

  - undefined when $x = 1$ and $A[1] > \mathit{key}$

- $\mathit{location2}(A[1 \ldots n], \mathit{key}, x) \ :\Leftrightarrow$

$$\begin{array}{ll} & A[x] = \mathit{key} \\ \vee & (A[x] > \mathit{key} \wedge (x = 1 \vee A[x-1] < \mathit{key})) \end{array}$$

  - what if $\mathit{key} = 24$?

  $\Rightarrow$ value $x = n + 1$ seems to be needed

- is the need of value $x = n + 1$ caused by the problem itself or is our trial just poor?

- test: if $n = 1$ then

  - we need 2 alternative answers "here" and "after" depending is $A[1] \geq \mathit{key}$ or not
  - the range $1 \leq x \leq n = 1$ allows only one answer
  - the notice generalises to every size of the array

⇒ the task itself requires that the range has $n+1$ values

- we choose range $1 \leq x \leq n+1$

- $\mathit{location3}(A[1\ldots n], \mathit{key}, x)$:⇔
$$
\begin{array}{ll}
 & A[x] = \mathit{key} \\
\vee & (A[x] > \mathit{key} \wedge (x = 1 \vee A[x-1] < \mathit{key})) \\
\vee & (x = n+1 \wedge A[n] < \mathit{key})
\end{array}
$$
  - if there exist several is such that $A[i] = \mathit{key}$ then it does not specify which i is chosen
  - the smallest should be chosen

- $\mathit{location4}(A[1\ldots n], \mathit{key}, x)$:⇔
$$
\begin{array}{ll}
 & (A[x] = \mathit{key} \wedge (x = 1 \vee A[x-1] < \mathit{key})) \\
\vee & (A[x] > \mathit{key} \wedge (x = 1 \vee A[x-1] < \mathit{key})) \\
\vee & (x = n+1 \wedge A[n] < \mathit{key})
\end{array}
$$
⇔
$$
\begin{array}{ll}
 & (A[x] \geq \mathit{key} \wedge (x = 1 \vee A[x-1] < \mathit{key})) \\
\vee & (x = n+1 \wedge A[n] < \mathit{key})
\end{array}
$$

- since $n \geq 1$, then $\mathit{location4}$ ⇔
$$
\begin{array}{ll}
 & (A[x] \geq \mathit{key} \wedge (x = 1 \vee A[x-1] < \mathit{key})) \\
\vee & (x = n+1 \wedge A[x-1] < \mathit{key})
\end{array}
$$
⇔
$$
\begin{array}{ll}
 & (x = n+1 \vee A[x] \geq \mathit{key}) \\
\wedge & (x = 1 \vee A[x-1] < \mathit{key})
\end{array}
$$
⇔: $\mathit{location5}(A[1\ldots n], \mathit{key}, x)$

- this is already quite clear!
- change of name was necessary since the reduction was based on assumption $n \geq 1$

- the parts have a clear meaning:
  - $x = n + 1 \vee A[x] \geq key \Leftrightarrow x$ is big enough
  - $x = 1 \vee A[x-1] < key \Leftrightarrow x$ is not too big
- we choose $location(A[1 \ldots n], key, x) :\Leftrightarrow location5(A[1 \ldots n], key, x) \Leftrightarrow (x = n + 1 \vee A[x] \geq key) \wedge (x = 1 \vee A[x-1] < key)$

Notices

- a predicate is not always indented to be used with all combinations of the possible values of its arguments
  - taking into account the other combinations makes a predicate often unnecessary complicated

$\Rightarrow$ it is worth to decide for which range it is meant and forget the other values

  - in the example $n > 0$ and $1 \leq x \leq n + 1$

- if used domain is K then predicates P_1 and P_2 are "*equal*" iff
  $K \Rightarrow P_1 \leftrightarrow P_2$

- if you do want a predicate that is specified in everywhere then the values outside the used domain can be fastened afterward
  - e.g. "False" by writing $1 \leq x \leq n + 1 > 1 \wedge P(x)$
  - if you want you can reduce the restriction inside $P(x)$

$\Rightarrow$ in the design of $P(x)$ you need not to worry about the values outside the used domain

  - remember to tell the used domain if it is $\neq$ True !

- attemption to write a state predicate often arouses questions which have not been noticed before
  - e.g. what "should be" exactly means?
  - e.g. which x is chosen if there are several alternatives?
  - pros and cons of exactness!
- removing the situation outside the used domain of the predicate may need effort and caution
- formalising may reveal incorrect preassumptions

- e.g. " domain $1 \leq x \leq n$ suffices"
- these kind of assumptions are a troublesome source of errors!

- formalisation may reveal ambiguities
  - ambiguity is not always bad!
  - but it is good to know if there exists ambiguity
- the above explain the "de facto" notice that formal specification increases significally the quality of the final product and reduces the time for development
  - although the cost is high
- it is worth to try to reduce the result or to transform it to different forms
  - a simple predicate is usually a better predicate
  - the reduced forms can support intuition better than the original
  - the reduced forms may be interpreted by a different intuition

  $\Rightarrow$ the belief that the predicate is "correct" strenghtens
- the reliability of reduction does not require intuition

  - the rules can (and is worth to) apply totally mechanically
  - this is essential for that the intuition of result really would increase the belief that the predicate is "correct"

- a well designed predicate is sometimes valid also outside its orignal domain
  - e.g. *location* "works" also when $n = 0$

Writing a predicate resembles programming in many ways

- coding an intuitive idea to a clumsy language
  - sometime the idea must be presented very indirectly!
- usually does not success in the first trial
- special cases must be thought
- semantical restrictions like "division by zero is not allowed" must be taken into account
- the permitted range for "inputs" must be decided
- it is worth to aim at a "nice" result

There exist also differencies

- you can calculate with predicates, with programs normally not
  - a lot of more possibilities for reduction
- efficiency aspects need not to be worried

⇒ better possibilities to reach an understandable result

  - there is no need to bargain for understandability because of other things
  - of course it is not allowed to change the meaning of the claim because of understandability!

- the collection of means is different
  - e.g. the order of calculation cannot be changed
  - e.g. in programming there is no as powerfull operators as "∃"
- you cannot test-drive predicates!
- (there exists a number theoretical formula $\varphi(x)$ such that **there cannot exist** an algorithm that takes x as its input and gives always an answer and even correctly, does $\varphi(x)$ hold or not)

A big question:

How can you be sure that a state predicate is correct?

- the final answer is: **you cannot be sure in any way**
- it is possible to state some formal, universal criterions for correctness
  - the value of state predicate shall not depend on properties of undefined ($\perp$), if the arguments are on the valid range
  - it is suspicious if the state predicate produces always either False or True, if the arguments are legal
  - c.f. programs
- these kind of criterions do not tell is the *meaning* desired
- to prove the correctness of the meaning of a state predicate should require that we had a formal knowledge what is the *correct meaning*

$\Rightarrow$ we should need another formal formula

- how could we know that it is correct?

$\Rightarrow$ it is not possible to prove the last formal formula φ of the chain since the measure of its correctness is not in formal form

  - if it would be, then φ would not be the last of the chain

- the belief of the "correctness" of the last formula of the chain can be based only in intuition

- the task of a state predicate in the specification of a program is to be part of the measure of correctness in proving the correctness of the program

⇒ it is usually meant to be the last formula of the chain

⇒ its correctness cannot be proved

  - in other words it is not possible to prove that its *meaning* is desired
  - it is a totally another story that it is possible to prove whether it *holds* in some state

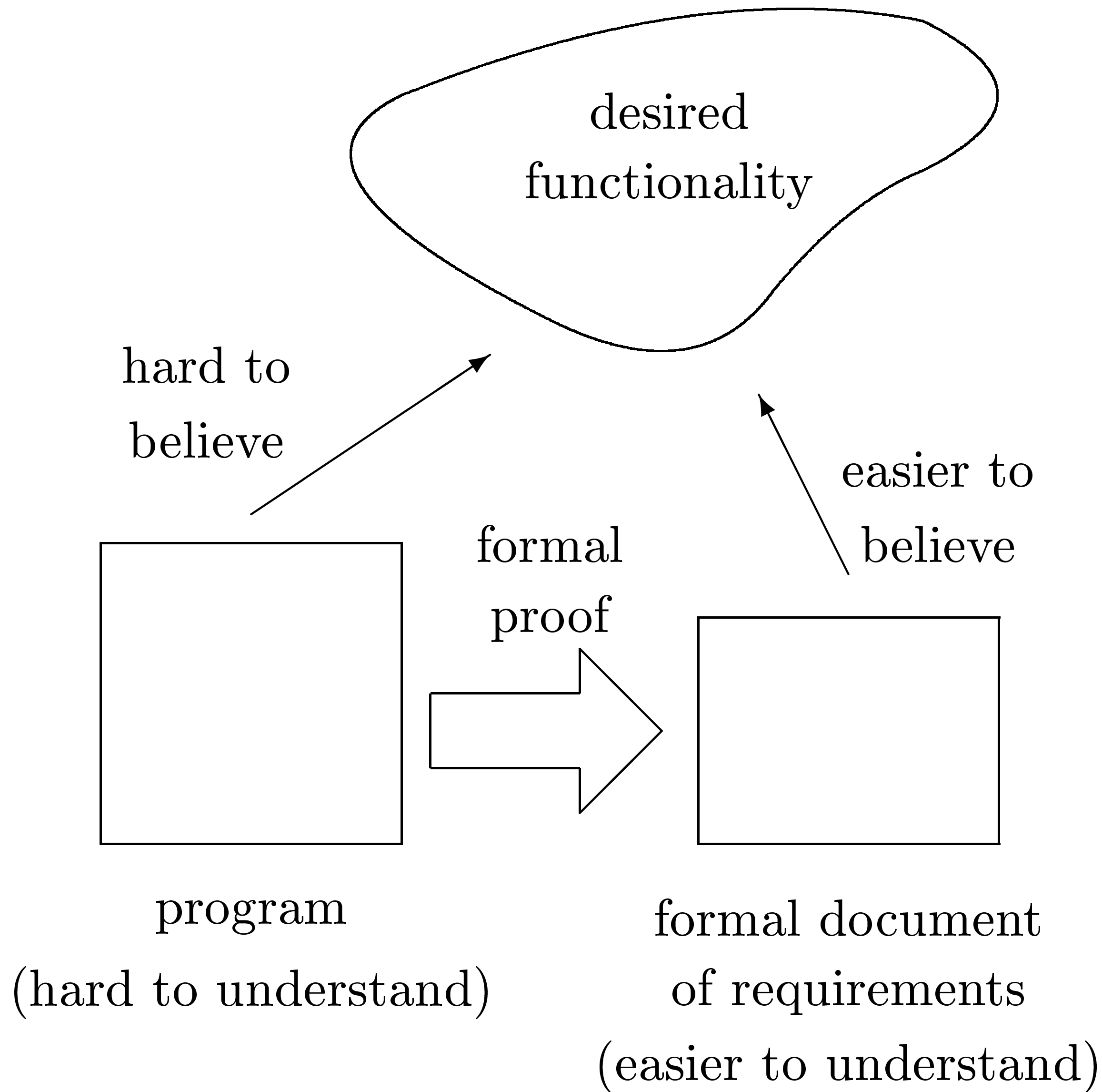
Corollaries

- a question: If it is not possible to be sure that the state predicate used in the proof of correctness is corret, then how it is possible to be sure that the program is correct?

- the answer: not in any way!

- it is not possible to prove that a program works as desired
- it is possible to prove that the program works e.g. according to a formal requirement specification composed of state predicates
- it is a matter of belief is the requirement specification correct or not

- a question: What benefits we then can get by proving the correctness of a program?
- the answer: by it it is possible to “reduce” the step that just has to be believed.



- corollaries:
  - the state predicate should be as easy to understand as possible (of course with the restriction that it must still tell the correct fact
  - it pays off to test state predicates

Ways to increase the belief that a state predicate is correct

- in the following examples we use the predicate $location(A[1 \ldots n], x, key) \Leftrightarrow (x = n + 1 \vee A[x] \geq key) \wedge (x = 1 \vee A[x-1] < key)$
- check that the value of the predicate does not depend on the properties of the value of the undefined, if the arguments are on the legal range
  - e.g. the only dangerous operations for *location* are $A[x]$ when $x = n + 1$, and $A[x-1]$ when $x = 1$
  - then the environment determines the result
  - (what about if $n = 0$)
- test by giving values for arguments, by reducing the formula, and by ensuring that the results corresponds to intuition

- especially it is worth to test borderline cases
- e.g. when $n = 0$ then
  location $\Leftrightarrow$ $(x = 1) \vee$ **?** •/•
- e.g. when $n = 1$ then $1 \leq x \leq 2$, so
  location
  $\Leftrightarrow (x = 2 \vee A[x] \geq key) \wedge (x = 1 \vee A[x - 1] < key)$
  $\Leftrightarrow (x = 2 \vee A[1] \geq key) \wedge (x = 1 \vee A[1] < key$
  $\Leftrightarrow (A[1] < key \rightarrow x = 2) \wedge (A[1] \geq key \rightarrow x = 1)$
  which holds, since it lists correct answers as a function of the result of the test "$A[1] < key$"

- you can also test by giving values for all arguments and by checking the result
  - it is worth to test with both the False and True cases
  - often this is very laborious and therefore an unefficient way of testing

  $\Rightarrow$ usually reducing is a better way to test

- reduce the formula in another form and check that the result corresponds the intuition
- example: $location(A[1 \ldots n], x, key)$ $\Leftrightarrow$

$$
\begin{array}{ll}
& x = 1 \land x = n+1 \\
\lor & (x = n+1 \land A[n] < \mathit{key}) \\
\lor & (x = 1 \land A[1] \geq \mathit{key}) \\
\lor & (A[x] \geq \mathit{key} \land A[x-1] < \mathit{key})
\end{array}
$$

- $x = 1 = n + 1$ processess correctly the case $n = 0$
- $x = n + 1 \land A[n] < \mathit{key}$ is the correct way to process the right border of the array when $n > 0$
- $x = 1 \land A[1] \geq \mathit{key}$ is the correct way to process the left border when $n > 0$
- $A[x] \geq \mathit{key} \land A[x-1] < \mathit{key}$ is correct in the middle of the array (it is undefined in the borders but then either of the previous cases produces True)

  It pays off to use the fact that you can calculate with state predicates.

Example: none the elements of array $A[1 \ldots n]$ are equal

- 1. try: $\mathit{nonequals1}(A[1 \ldots n]) :\Leftrightarrow \forall i, j \in \{1, \ldots, n\} : A[i] \neq A[j]$
- let us test it by the borderline case $n = 1$
  - then $\mathit{nonequals1} \Leftrightarrow A[1] \neq A[1]$ oops!

- *nonequals2*$(A[1 \ldots n] :\Leftrightarrow$
  $\forall i, j \in \{1, \ldots n\} : (i \neq j \rightarrow A[i] \neq A[j])$
  - works with the borderline cases
    $n = 0$ and $n = 1$
  - does not index A illegally
- according to intuition it should hold that
  ordered2$(A[1 \ldots n]) \;\Leftrightarrow$
  ordered$(A[1 \ldots n]) \wedge$ *nonequals*$(A[1 \ldots n])$
  let us check!
  - "$\Leftarrow$": easy
  - "*ordered2*$(A[1 \ldots n]) \Rightarrow$ *ordered*$(A[1 \ldots n])$"
    directly
  - "*ordered2*$(A[1 \ldots n]) \;\Rightarrow$
    nonequals2$(A[1 \ldots n])$" :
    when $i < j$ then $1 \leq i < n$ thus $A[i] < A[i+1] < \cdots < A[j]$, and therefore $A[i] \neq A[j]$; similarly, when $j < i$; the case $i = j$ directly

$\Rightarrow$ we believe that *nonequals2* is correct
nonequals$[A[1 \ldots n]) :\Leftrightarrow$ *nonequals2*$(A[1 \ldots n])$

An example of how to write a state predicate with sets: same elements

- *same-elems*$(A[1 \ldots n], B[1 \ldots n]) \;\sim\;$ arrays A and B have the same elements

- 1. try: $\textit{same-elems1}(A[1\ldots n], B[1\ldots n]) :\Leftrightarrow$ $\forall i\ ;\ 1 \leq i \leq n : \exists j\ ;\ 1 \leq j \leq n : A[i] = B[j]$
  - allows $A = [1, 1], B = [1, 2]$
- $\textit{same-elems2}(A, B) :\Leftrightarrow$ $\textit{same-elems1}(A, B) \wedge \textit{same-elems1}(B, A)$
  - allows $A = [1, 1, 2], B = [1, 2, 2]$

$\Rightarrow$ it seems necessary to count how many each element there is

- we introduce an auxiliary notation: $\textit{amount}(x, A[1\ldots n]) :=$ $|\,\{\, i \mid 1 \leq i \leq n \wedge A[i] = x \,\}\,|$
- (:= means here "is defined to mean")
- $\textit{same-elems3}(A, B) :\Leftrightarrow \forall i\ ;\ 1 \leq i \leq n :$ $\textit{amount}(A[i], A) = \textit{amount}(A[i], B)$
- $\textit{same-elems3}(A, B)$ is asymmetric with respect to A and B

  $\Rightarrow$ can it be correct?

  - the antithesis: the symmetric formula does not hold

  $\Rightarrow$ $\exists k\ ;\ 1 \leq k \leq n :$ $\textit{amount}(B[k], A) \neq \textit{amount}(B[k], B)$

- if $amount(B[k], A) > 0$ then $\exists i \;;\; 1 \leq i \leq n : B[k] = A[i]$, thus *same-elems3* $\Rightarrow$ $amount(B[k], A) = amount(B[k], B)$

$\Rightarrow$ $amount(B[k], A) = 0 \;\Rightarrow$ $amount(B[k], B) > 0$

$\Rightarrow$ B has more elements than A

$\Rightarrow$ a contradiction

$\Rightarrow$ the antithesis is incorrect

$\Rightarrow$ the asymmetry is only virtual

$\Rightarrow$ we accept that *same-elems*$(A, B) :\Leftrightarrow$ *same-elems3*(A, B)

Formatting a state predicate is sometimes difficult

- the language of logic is rigid
  - designed by mathematicians
  - it is not designed for expressing large claims
  - it is not designed to be read by a computer
- on the other hand logic allows adding own notations

  $\Rightarrow$ it shall suffice

  - in this course own notations can be introduced **but they must be defined** (except if the meaning is obvious)

  - e.g. since *ordered* has been defined it can be used as a part of other predicates

- when using new data abstractions it is often necessary to introduce new notations to be used with the abstraction

  - e.g. queue, stack, graph

- the reason of difficulties when writing a state predicate is often that the exact content of the claim for all situations has not been thought earlier

  - so for the reason of the difficulty of a state predicate is often that **the matter itself** is difficult
  - without formalising the claim its gaps usually remain unnoticed
    $\Rightarrow$ troublesome errors

- the first and often the biggest benefit is that formalism forces to think the matter exactly

2.3 Specification of a Program

Correctness of a program

- it has turned out to be useful to divide the requirement of correctness into two parts:
  - *partial correctness (osittainen oikeellisuus)*
  - *termination (pysähtyvyys)*
- partial correctness = the program is not allowed to do anything wrong
  - it shall never give an incorrect answer
  - if it has terminated then it shall not stay in an incorrect state
  - (if a program terminates then the final state is correct, the right result has been computed; and if a program fails to terminate then it may never produce the correct answer)
- termination = the program must eventually terminate in a controlled manner
  - crashing because of a fault (division by zero, indexing pass the border of an array, etc.) is not controlled termination

  ⇒ the definition of termination denies division by zero, etc.

- the next program is partially correct for any specification:

```
while true do
    (* don't do anything! *)
endwhile
```

- *total correctness (täysi oikeellisuus)* = partial correctness + termination
- the requirement of termination is same for all programs
- the requirement of partial correctness depends on the case
  - e.g. in the end of a sorting program the array contains all the original elements in ascending order
  - e.g. in the end of a searching program i points to the searched record

⇒ we need a way to represent the partial correctness requirement of a program

Specification of partial correctness

- the requirement of partial correctness of a program can be expressed in form

  { *precondition* } *program* { *postcondition* }

  where *precondition (alkuehto)* and *postcondition (loppuehto)* are state predicates

- example: raising to a power

  $\{\ n \geq 0\ \}$
  $x := 1;$
  for $i := 1$ **to** n **do**
  $\quad x := x \cdot a$
  endfor
  $\{\ x = a^n\ \}$

- example: initialisation

  $\{\ \mathsf{True}\ \}$
  for $i := 1$ **to** n **do**
  $\quad A[i] := 0$
  endfor
  $\{\ \forall i\ ;\ 1 \leq i \leq n : A[i] = 0\ \}$

- whenever necessary we introduce auxiliary symbols to represent values that do not change during the program
  - they do not represent program variables but their initial values
  - represented often by subindex 0, e.g. $x_0, A_0[i]$
  - so-called *ghost variables (haamumuuttujat)*
- e.g. swapping the values of variables

$$\{\ x = x_0 \wedge y = y_0\ \}$$
$$t := x;\ x := y;\ y := t$$
$$\{\ x = y_0 \wedge y = x_0\ \}$$

- so: in this course the notation

  { *precondition* } *program* { *postcondition* }

  means the following:

  > **if**, when the *program* starts, the *precondition* holds
  > **then**, after the *program* has terminated, the *postcondition* holds.

- **warning!** the meaning alternates slightly with different writers
  - our choice is same as Backhouse's

  - e.g. Gries expresses the same in form
    precondition { *program* } *postcondition*

Values of which variables can be changed?

- the following program clearly fills its formal specification!

  $\{\ n \geq 0\ \}$
  $x := 1;\ a := 1;\ n := 1$
  $\{\ x = a^n\ \}$

- changing the values of variables can be denied by requiring that their values in the end are same as in the beginning:

  $\{\ n = n_0 \geq 0 \wedge a = a_0\ \}$
  $\ldots$
  $\{\ x = a^n \wedge n = n_0 \wedge a = a_0\ \}$

- laborious compared to the commonness of the problem

⇒ the solution: let us partition the variables in two groups according to whether the program is allowed to change their values or not

  - *fixed* variables
  - ordinary variables

- how to concern on the following?

  $\{ n \geq 0 \}$
  $tmp := n;\ n := 0;$
  $\ldots$ (* here tmp is not touched *)
  $n := tmp$
  $\{ x = a^n \}$

- the solution
  - it is hard (sometimes impossible) to check if the program really changes the value of some variable
  - it is easy to check if in the program there exist any assignment statements directed to the variable in question

  $\Rightarrow$ **decision:** even trying to make an assignment to a fixed variable is denied
  - (the problem of several access paths)

- fixed variables are a different thing than ghost variables
  - ghost variables do not exist in the program, only in state predicates

- input, auxiliary, and output variables
  - the inputs are given in the input variables

  - the results are returned in the output variables
  - the auxiliary variables are for subresults

- the partition to fixed and ordinary variables is a different thing than the partition to input, auxiliary, and output variables
  - fixed variables are usually input variables
  - all input variables are not fixed:
    e.g. sorting an array in place

- usually the partition to ordinary and fixed variables is assumed to be known and is left without mention

  ⇒ these groups do not even have established names

Partial correctness and termination

- the next program is partially correct, were P and Q whatever:

  $\{ P \}$
  while True **do**
  endwhile
  $\{ Q \}$

⇒ a specification is not complete unless termination of the program is required

- usually termination is desired
  - an important exception: reactive programs (not discussed on this course)
- proving termination requires often remarkable additional effort
  - e.g.

$x := 1;\ y := 1;\ z := 1;\ n := 3;$
while $x^n + y^n \neq z^n$ **do**
  if $y > 1$ **then** $x := x + 1;\ y := y - 1$
  elsif $z > 1$ **then** $z := z - 1;\ y := x + 1;\ x := 1$
  elsif $n > 3$ **then** $n := n - 1;\ z := x + 1;\ x := 1$
  else $n := x + 3;\ x := 1$
endwhile
$\{\ x \geq 1 \land y \geq 1 \land z \geq 1 \land n \geq 3 \land x^n + y^n = z^n\ \}$

$\Rightarrow$ partial correctness and termination are usually proved separately

$\Rightarrow$ we are going to deal with a lot of interphases where termination is not required

$\Rightarrow$ in this course you must be carefull about when termination is required and when not

Specification of termination

- termination is required (in this course) by putting state predicates in angle brackets:

  $\langle\ x = x_0 \wedge y = y_0\ \rangle$
  $t := x;\ x := y;\ y := t$
  $\langle\ x = y_0 \wedge y = x_0\ \rangle$

- so: in this course the notation

  $\langle$ *precondition* $\rangle$ *program* $\langle$ *postcondition* $\rangle$

  means the following:

  > **if**, when the *program* starts and the *precondition* holds
  > **then** the *program* terminates and the *postcondition* holds.

- **warning!** even this notation has not been established
  - for some writers $\{\ P\ \}$ *program* $\{\ Q\ \}$ includes the requirement of termination (Gries)
  - our way: N. Francez: *Program Verification*

- the requirement of nothing but termination can be expressed as follows:

  $\langle$ *precondition* $\rangle$ *program* $\langle$ True $\rangle$

General assumptions

- for that it should not be necessary to list a big set of general definitions in examples, exams, etc., we assume the followings, unless something else is mentioned
- variables get their values from the set $\mathbb{Z}$
- the notation $A[a \ldots y]$ means an array which index range is $a, a+1, \ldots, y$
  - a and y are fixed
  - $y \geq a - 1$
  - multidimensional arrays $A[1 \ldots n, 1 \ldots m]$
- after the borders of an array have been given, the same name of the array can be used without borders
- the type, size, etc. of a ghost variable are determined by the correspondings of the corresponding program variable
  - e.g. if $A[1 \ldots n]$ then $A = A_0$ means that the index range of A_0 is $1, \ldots, n$, and $\forall i \; ; \; 1 \leq i \leq n : A_0[i] = A[i]$

Specification example: sorting array $A[1 \ldots n]$

- 1. try:

  $\langle$ True $\rangle$
  sort
  $\langle$ *ordered*(A) $\rangle$

- a problem: it allows

  $\langle$ True $\rangle$
  for $i := 1$ **to** n **do** $A[i] := 0$ **endfor**
  $\langle$ *ordered*(A) $\rangle$

- the solution:

  $\langle A = A_0 \rangle$
  sort
  $\langle$ *ordered*$(A) \wedge$ *same-elems*(A, A_0) $\rangle$

How serious the lack in the original specification is?

- often the only operation that changes the array in a sorting algorithm is swapping two elements

$\Rightarrow$ it is easy to see whether *same-elems*(A, A_0) holds or not

$\Rightarrow$ the main interest of the proof concentrates proving the claim *ordered*(A)

- however, sometimes $\mathit{same\text{-}elems}(A, A_0)$ is not obvious
  - moving a hole in an array as in the following example

```
Insertion-Sort(A[1 ... n])
for j := 2 to n do
    key := A[j];
(* add A[j] to ordered sequence
    A[1 ... j − 1] *)
    i := j − 1;
    while i > 0 and A[i] > key do
        A[i + 1] := A[i];
        i := i − 1
    endwhile
    A[i + 1] := key
endfor
```

  - *same-elems*(B, A_0) in the following example

Counting-Sort$(A[1 \ldots n], B[1 \ldots m], M)$
$\langle\ \forall i\ ;\ 1 \le i \le n : 0 \le A[i] \le M\ \rangle$
for $k := 0$ **to** M **do** $C[k] := 0$ **endfor**
for $i := 1$ **to** n **do**
    $C[A[i].key] := C[A[i].key] + 1$
endfor
(* now $C[k]$ knows how many element has $key = k$ *)

for $k := 1$ **to** M **do** $C[k] := C[k] + C[k-1]$ **endfor**
(* now $C[k]$ knows how many element has $key \le k$ *)
for $i := n$ **downto** 1 **do**
    $B[C[A[i].key]] := A[i]$;
    $C[A[i].key] := C[A[i].key] - 1$
endfor

⇒ how serious an error is depends how the specification is used

  - if it is given to a malicious or hostile reader (american layer), it must be correct
  - when it is used in a situation where the missing part is clearly obvious to everyone, then the error is not serious

- c.f. specification
  "you get dessert after the plate is empty"

- it is better to express the requirement *same-elems*(A, A_0) even informally than leave it out

Example: binary search and its specification

- $A[1 \ldots n]$ and *key* are fixed

```
⟨ ∀i ; 1 ≤ i ≤ n − 1 : A[i] ≤ A[i + 1] ⟩
a := 1 y := n + 1;
while a < y do
    v := (a + y) div 2;
    if A[v] < key then a := v + 1
    else y := v
    endif
endwhile
⟨ (a = n + 1 ∨ A[a] ≥ key) ∧ (a = 1 ∨ A[a − 1] < key) ⟩
```

Example: binary power and its specification

- a and n are fixed
- x, a, and b are of some number type, e.g. $\mathbb{R}$

```
⟨ n ≥ 0 ⟩
i := n; b := a; x := 1;
while i > 0 do
    if i mod 2 = 1 then x := b · x endif;
    i := i div 2; b := b · b;
endwhile
⟨ x = a^n ⟩
```

Example: finding the largest 0-square in a matrix whose elements are either 0 or 1

- an auxiliary state predicate:
  $zerosquare(i, j, k, A[1 \ldots n, 1 \ldots m]) :\Leftrightarrow$
  $i + k - 1 \leq n \wedge j + k - 1 \leq m \wedge$
  $\forall h \; ; \; i \leq h \leq i + k - 1 :$
  $\forall l \; ; \; j \leq l \leq j + k - 1 : A[h, l] = 0$

  - domain: $1 \leq i \leq n, 1 \leq j \leq m, k \geq 0$
  - why condition $i+k-1 \leq n \wedge j+k-1 \leq m$ is needed?

- A is fixed, and m and n are fixed without any notification because of the general assumption

- for the answer:

  - the corner which has the smallest coordinates is in location (i, j)
  - its size is k

$\langle \; \forall i \; ; \; 1 \leq i \leq n : \forall j \; ; \; 1 \leq j \leq m : A[i, j] \in \{0, 1\} \; \rangle$
find_the_largest_zerosquare
$\langle \; 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge k \geq 0 \wedge$
$zerosquare(i, j, k, A[1 \ldots n, 1 \ldots m]) \wedge$
$\forall i \; ; \; 1 \leq i \leq n : \forall j \; ; \; 1 \leq j \leq m :$
$\forall h \; ; \; h > k : \neg zerosquare(i, j, h, A[1 \ldots n, 1 \ldots m]))$
$\rangle$

Example: going trough the direct and indirect prerequisites of a course

- the set of all courses is *Courses*
- the direct prerequisites of course k is set *dir-prereq*(k)
- we introduce notations $(k, g \in \mathit{Courses})$
  - $k \rightarrow g :\Leftrightarrow g \in \mathit{dir\text{-}prereq}(k)$
  - $k \rightarrow^{+} g :\Leftrightarrow \exists n \in \mathbb{N} : n > 0 \land$ $\exists k_0, k_1, \ldots, k_n : k = k_0 \land k_n = g \land$ $\forall i \;;\; 1 \leq i \leq n : k_{i-1} \rightarrow k_i$
- the algorithm and its specification
  - *Courses*, *dir-prereq*, and k_0 are fixed
  - the type of variables k, g, and k_0 is *Courses*
  - *unfinished*, *found*, and *dir-prereq* are of type 2^{Courses}, that is, set of courses

$unfinished := \{k_0\};\ found := \emptyset$
while $unfinished \neq \emptyset$ **do**
  $choose\text{-}some\ k \in unfinished$
  forall $g \in dir\text{-}prereq(k)$ **do**
    if $g \notin found$ **then**
      $found := found \cup \{g\}$
      if $g \neq k_0$ **then**
        $unfinished := unfinished \cup \{g\}$
      endif
    endif
  endfor
  $unfinished := unfinished - \{k\}$
endwhile
$\langle\ found = \left\{\ k \in Courses \ \middle|\ k_0 \rightarrow^{+} k\ \right\})\ \rangle$

- conclusion: when using complicated or abstract structures it is often necessay to introduce domain specific notations
- (correspondingly in proofs we may have to use domain specific results)

3 Proving Correctness of a Program

In this section we examine means for proving that a small part of a program fulfills its specification

$\Rightarrow$ we have to discuss about the semantics of statements of a programming language

These means are useful especially for designing and verifying a small part of a program whose operating principle is difficult

- searching and browsing tasks
- sorting
- code conversions
- arithmetic tasks
- processing a difficult data structure
- processing strings, ...

On same kind of ideas is based also proving the correctness of

- bigger programs
- concurrent and parallel programs
- abstract specifications
- complicated algorithms

3.1 General Principle

A suggestion for proof method: symbolic execution

- the program is went through by one statement at a time
- express the state of the program in every place by a state predicate

Example: swapping the values of variables

- it must be proven that the following part of a program swaps the values of variables x and y

  $t := x;\ x := y;\ y := t$

- straightforward solution
  - execute the program symbolically by keeping book of the values of variables

    $$\begin{array}{l} \langle\ x = x_0 \wedge y = y_0\ \rangle \\ \qquad t := x; \\ \langle\ x = x_0 \wedge y = y_0 \wedge t = x_0\ \rangle \\ \qquad x := y; \\ \langle\ x = y_0 \wedge y = y_0 \wedge t = x_0\ \rangle \\ \qquad y := t \\ \langle\ x = y_0 \wedge y = x_0 \wedge t = x_0\ \rangle \end{array}$$

- since in the end $x = y_0$ and $y = x_0$ the program swaps the values of x and y

⇒ the program could be proven to be correct by symbolic execution

A harder example: searching from an array

- the problem
  - let array $A[1 \ldots n]$ $(n \geq 1)$ and element x be given
  - the task is to find the smallest i such that $A[i] = x$
  - if such i does not exist then $i = 0$ must be returned
  - A and x are fixed
- requirement of (total) correctness

$$\begin{array}{l} \langle\ n \geq 1\ \rangle \\ \textit{search} \\ \langle\ (1 \leq i \leq n \wedge A[i] = x \wedge \\ \qquad\qquad \forall j\ ;\ 1 \leq j \leq i-1 : A[j] \neq x) \ \ \vee \\ \qquad i = 0 \wedge \forall j\ ;\ 1 \leq j \leq n : A[j] \neq x\ \rangle \end{array}$$

- the program to be proven correct

(* *search* *)
$i := 1$;
while $i < n \wedge A[i] \neq x$ **do**
    $i := i + 1$
endwhile;
if $A[i] \neq x$ **then** $i := 0$ **endif**

- let's try to execute *search* symbolically:

$\langle\ n \geq 1 \wedge A = A_0 \wedge x = x_0\ \rangle$
$i := 1;$
$\langle\ n \geq 1 \wedge A = A_0 \wedge x = x_0 \wedge i = 1\ \rangle$
while $i < n \wedge A[i] \neq x$ **do**
$\langle\ n \geq 1 \wedge A = A_0 \wedge x = x_0 \wedge i =?? \wedge A[??] \neq x_o\ \rangle$
...

does not work!

⇒ symbolic execution does not usually work if the program has loops

- we need some other way to reason about the functioning of the program

A better proof method

- write the requirement of partial correctness in the form

$\{$ *precondition* $\}$ *program* $\{$ *postcondtion* $\}$

- decorate the program with **suitably chosen** state predicates so that in the beginning there is $\{$ *precondition* $\}$ and in the end $\{$ *postcondition* $\}$

- show that each predicate holds using the previous predicate and the executable statements between them

  - loops have special rules, we'll come back to them

  ⇒ in the end the program gives the correct result, if it gets there

- by using predicates show that anything illegal is never done
  - division by zero, overflow of an index of an array, ...

  ⇒ the program does not halt before end

- show that every loop is eventually exited
  - we"ll come back to means

  ⇒ the program terminates

⇒ the program runs to end, terminates there, and gives the correct result

Example: *search* decorated by state predicates

- the following state predicates are chosen by using the knowledge of how *search* works
- in the sequel some of them are derived from the other
- in the sequel loops are handled in a different way

$\{\ n \geq 1\ \}$
$i := 1;$
$\{\ n \geq 1 \land i = 1\ \}$
while $i < n \land A[i] \neq x$ **do**
    $\{\ 1 \leq i < n \land \forall j\ ;\ 1 \leq j \leq i : A[j] \neq x\ \}$
    $i := i + 1$
    $\{\ 1 < i \leq n \land \forall j\ ;\ 1 \leq j \leq i - 1 : A[j] \neq x\ \}$
endwhile;
$\{\ (1 \leq i \leq n \land \forall j\ ;\ 1 \leq j \leq i - 1 : A[j] \neq x) \land$
    $(i \geq n \lor A[i] = x)\ \}$
if $A[i] \neq x$ **then**
    $\{\ n \geq 1 \land i = n \land \forall j\ ;\ 1 \leq j \leq i : A[j] \neq x\ \}$
    $i := 0$
    $\{\ n \geq 1 \land i = 0 \land \forall j\ ;\ 1 \leq j \leq n : A[j] \neq x\ \}$
(* **else**
    $\{\ 1 \leq i \leq n \land A[i] = x \land$
     $\forall j\ ;\ 1 \leq j \leq i - 1 : A[j] \neq x$ *)
endif
$\{\ (1 \leq i \leq n \land A[i] = x \land$
        $\forall j\ ;\ 1 \leq j \leq i - 1 : A[j] \neq x)$
  $\lor\ n \geq 1 \land i = 0 \land \forall j\ ;\ 1 \leq j \leq n : A[j] \neq x\ \}$

Choosing state predicates

- state predicates need not necessary define the state totally
- they must be **strong enough** so that by using them the program can be proven to be correct
- they shall **not** be **too strong**
  - they shall not assert false things
  - proving a property that is correct, but hard to prove and unnecessary for the total proof, is useless
  - too much strength can prevent reduction of state predicate to an easy form

⇒ requires vision how the program works

  - what is true and where
  - what truths are essential

- we shall soon learn calculation tehcniques by which most of state predicates can be derived from others
- still two things remain under proof maker's creativity:
  - suitable weakening of state predicates when needed
  - choosing state predicates for handling loops

How shall state predicates put into angle brackets be interpreted in the middle of a program ?

$\langle\ x \neq 0\ \rangle$
$a := 1;$
$\langle\ x \neq 0 \wedge a = 1\ \rangle$
if $x < 0$ **then**
  $\langle\ x < 0 \wedge a = 1\ \rangle\ x := -x\ \langle\ x > 0 \wedge a = 1\ \rangle$
endif
$\langle\ x > 0 \wedge a = 1\ \rangle$

- the *baseline of progression* of program location l is
  - the beginning of the program if l is not inside any structural statement
  - otherwise it is the beginning of the branch of the nearest surrounding structural statement where l is
- e.g. P_0 is in the baseline of progress of every P_i, and correspondingly Q, R, and S

  $\langle\ P_0\ \rangle\ a := 1;\ \langle\ P_1\ \rangle\ y := n + 1;\ \langle\ P_2\ \rangle$
  while $a < y$ **do**
    $\langle\ Q_0\ \rangle\ v := (a + y)\ \mathsf{div}\ 2;\ \langle\ Q_1\ \rangle$
    if $A[v] < key$ **then** $\langle\ R_0\ \rangle\ a := v + 1\ \langle\ R_1\ \rangle$
    else $\langle\ S_0\ \rangle\ y := v\ \langle\ S_1\ \rangle$
    endif $\langle\ Q_2\ \rangle$
  endwhile
  $\langle\ P_3\ \rangle$

- in this course we use the following interpretation:
  notation $\langle\ P\ \rangle$ in location l claims that if the precondition held when the program was started then
  - P holds every time the execution is in location l
  - if the execution is now in the baseline of progress of l, then eventually it will be on location l

Summary of how state predicates are interpreted in different locations

- the notation $\{\ P\ \}$ or $\langle\ P\ \rangle$ in the beginning of a program specifies that the precondition of the program is P
- if the precondition of the program is not defined then it is True
- notation $\{\ P\ \}$ in location l claims that P holds every time the execution is in location l, requiring that the precondition held when the program was started
- notation $\{\ \mathsf{True}\ \}$ in location l claims that if the execution is in the baseline of progress of l, then eventually it will be on location l, requiring that the precondition held when the program was started

- notation $\langle\ P\ \rangle$ means the same as notations $\{\ P\ \}$ and $\langle\ \mathsf{True}\ \rangle$ together

General rules for decorating a program

- let
  - S be a sequence of zero or more consequtive statements
  - P and Q be state predicates
- if $\langle\ P\ \rangle\ S\ \langle\ Q\ \rangle$
  then $\{\ P\ \}\ S\ \{\ Q\ \}$
- if $\{\ P\ \}\ S\ \{\ Q\ \}$, $P_s \Rightarrow P$, and $Q \Rightarrow Q_w$
  then $\{\ P_s\ \}\ S\ \{\ Q_w\ \}$
  - rule of thumb: $s \sim$ stronger, $w \sim$ weaker
  - reasoning: if the state before execution of S fulfills P_s, then it fulfills also P
- if $\langle\ P\ \rangle\ S\ \langle\ Q\ \rangle$, $P_s \Rightarrow P$, and $Q \Rightarrow Q_w$
  then $\langle\ P_s\ \rangle\ S\ \langle\ Q_w\ \rangle$
  - reasoning: as previous
- if $\{\ P\ \}\ S\ \{\ Q_1\ \}$ and $\{\ P\ \}\ S\ \{\ Q_2\ \}$
  then $\{\ P\ \}\ S\ \{\ Q_1 \wedge Q_2\ \}$
  - reasoning: if the state after execution of S fulfills Q_1 and Q_2, then it fulfills $(Q_1 \wedge Q_2)$

- if $\langle\ P\ \rangle\ S\ \langle\ Q_1\ \rangle$ and $\langle\ P\ \rangle\ S\ \langle\ Q_2\ \rangle$
  then $\langle\ P\ \rangle\ S\ \langle\ Q_1 \wedge Q_2\ \rangle$

- even

  if $\langle\ P\ \rangle\ S\ \langle\ Q_1\ \rangle$ and $\{\ P\ \}\ S\ \{\ Q_2\ \}$
  then $\langle\ P\ \rangle\ S\ \langle\ Q_1 \wedge Q_2\ \rangle$

  - it suffices that reaching the destination is guaranteed once

3.2 Semantics of Statements

How can we show that a predicate holds in some location of a program?

- the instruction was: "using the previous predicate and the statements between them"
- how the statements between predicates can be used?
- we need rules for reasoning in the form

  $\{ \ P \ \} \ \mathit{statement} \ \{ \ Q \ \}$ and / or
  $\langle \ P \ \rangle \ \mathit{statement} \ \langle \ Q \ \rangle$

  for every type of statement

- these kind of rules for reasoning must somehow lean on the meaning of statements

It is often thought that the *meaning*, that is, the *semantics* of a statement, is the way how it changes the state of the program

- state of a program =
  - information of the location of execution
  - list of values of variables

- e.g. if in the beginning $A = \langle 3,1,6,2,1,1,1 \rangle$ (thus $n = 7$), $x = 2$, and $i = -95$, then the state of the searching program after different numbers of steps is

0: loc=1, $A = \langle 3,1,6,2,1,1,1 \rangle$, $n = 7, x = 2, i = -95$
1: loc=2, $A = \langle 3,1,6,2,1,1,1 \rangle$, $n = 7, x = 2, i = 1$
2: loc=3, $A = \langle 3,1,6,2,1,1,1 \rangle$, $n = 7, x = 2, i = 1$
3: loc=2, $A = \langle 3,1,6,2,1,1,1 \rangle$, $n = 7, x = 2, i = 2$
...
7: loc=2, $A = \langle 3,1,6,2,1,1,1 \rangle$, $n = 7, x = 2, i = 4$
8: loc=4, $A = \langle 3,1,6,2,1,1,1 \rangle$, $n = 7, x = 2, i = 4$

```
1: i := 1
2: while i < n ∧ A[i] ≠ x do
3:     i := i + 1
   endwhile
4: ...
```

This kind of way to express the meaning of a statement fits poorly in to the reasoning needed for program verification

- if all variables have been given a concrete value, then only one case can be handled at a time
- if the values of variables are expressed symbolically then it is symbolic execution
  $\Rightarrow$ does not work

For program verification there has been developed another way to represent the meaning of a statement:

weakest precondition (heikoin esiehto)

- let S be a statement (or a part of a program) and Q a predicate concerning a state
- $wp(\ S, Q\ )$ (the weakest precondition of Q with respect to S, Q:n heikoin esiehto S:n suhteen) is the weakest condition concerning a state that suffices to guarantee that an execution that is started in the state terminates and in the end state Q holds
- in another words:
  $P \;\Rightarrow\; wp(\ S, Q\ )$
  if and only if
  $\langle\ P\ \rangle\ S\ \langle\ Q\ \rangle$
- $wp(\ S, Q\ )$ is therefore a function that takes a statement and a predicate that describes the next state, and produces a predicate that describes the state that precedes the statement
  – the so called *predicate transformer (predikaattimuunnin)*

Examples ($x \in \mathbb{R}, n \in \mathbb{Z}$)

- $wp(\ x := x - 2,\ x > 0\ ) \;\Leftrightarrow\; x > 2$

- $wp(\,\textbf{if } x < 0 \textbf{ then } x := -x \textbf{ endif},\ x > 0\,) \Leftrightarrow x \neq 0$
- $wp(\,\textbf{while } n > 0 \textbf{ do } n := n - 1 \textbf{ endwhile},\ n = 0\,) \Leftrightarrow n \geq 0$
- $wp(\,x := 0,\ x = 0\,) \Leftrightarrow \mathsf{True}$
- $wp(\,x := 1,\ x = 0\,) \Leftrightarrow \mathsf{False}$
- $wp(\,\textbf{while } n \neq 0 \textbf{ do } n := n + 1 \textbf{ endwhile},\ \mathsf{True}\,) \Leftrightarrow n \leq 0$

Soon we give the *wp*-meaning for a set of most common statements

- statements of real programming languages have often complicated side effects
  - e.g. C: `x = ++i;`
  - e.g. Pascal:

```
function plusplus(var y:  integer):
integer;
begin plusplus := y; y := y+1 end;
...
x := plusplus(i)
```

$\Rightarrow$ verification theories for true programming languages becomes unclear

$\Rightarrow$ we restrict to simplified statements which do not have side effects

- the applier of the theory must be able to recognise side effects etc. exceptions using the assumptions of the theory and she must take them in account!

The *wp*-semantics for the simple assignment statement

- let there be given a predicate Q and an assignment statement S of the form

  $x := \mathit{expression}$

  where

  - computation of *expression* does not cause side effects
  - x is a single variable (not e.g. not an element of an array)

- let $\Uparrow\mathit{expression}$ be a predicate concerning a state so that it holds exactly when computation of *expression* is definitely possible

  - in other words, there is no division by zero, or the like

  $\Rightarrow$ $\langle\ P\ \rangle\ x := \mathit{expression}\ \langle\ \mathsf{True}\ \rangle$ if and only if
  $P\ \Rightarrow\ \Uparrow\mathit{expression}$

- the state after execution of S is otherwise the same as before but the location and the value of x have been changed

$\Rightarrow$ if in the state after there holds predicate Q then in the state before there held Q but modified so that every occurrence of x has been substituted by *expression.*

- in this course the notation $Q[x \leftarrow \mathit{expression}]$ denotes a predicate that is made by substituting every free occurrence of x in Q by *expression.*
  - the so called *textual substitution (tekstikorvaus)*
  - (other writers usually use the notation $Q^x_{\mathit{expression}}$, G-S also $Q[x := \mathit{expression}]$ and Backhouse $Q(x)$ vs. $Q(\mathit{expression})$).
- thus

  if $\langle\ P\ \rangle\ x := \mathit{expression}\ \langle\ Q\ \rangle$
  then $P \Rightarrow\ \Uparrow\!\mathit{expression}$ and
  $P \Rightarrow Q[x \leftarrow \mathit{expression}]$
- on the other hand
  $\{\ Q[x \leftarrow \mathit{expression}]\ \}\ x := \mathit{expression}\ \{\ Q\ \}$
- therefore
  $\langle\ P\ \rangle\ x := \mathit{expression}\ \langle\ Q\ \rangle$ if and only if
  $P \Rightarrow\ \Uparrow\!\mathit{expression}$ and $P \Rightarrow Q[x \leftarrow \mathit{expression}]$,
  in other words
  $P \Rightarrow\ \Uparrow\!\mathit{expression} \wedge Q[x \leftarrow \mathit{expression}]$

- therefore

$$wp(\ x := \mathit{expression},\ Q\ ) \ \Leftrightarrow$$
$$\Uparrow\!\mathit{expression} \wedge Q[x \leftarrow \mathit{expression}]$$

Examples

- $wp(\ x := x - 2,\ x > 0\ ) \ \Leftrightarrow$
  $\mathsf{True} \wedge (x > 0)[x \leftarrow x-2] \Leftrightarrow x-2 > 0 \Leftrightarrow x > 2$
- $wp(\ y := x - y,\ x = x_0\ ) \ \Leftrightarrow$
  $\mathsf{True} \wedge (x = x_0)[y \leftarrow x - y] \ \Leftrightarrow \ x = x_0$
- $wp(\ y := x - y,\ y = x_0\ ) \ \Leftrightarrow$
  $\mathsf{True} \wedge (y = x_0)[y \leftarrow x - y] \ \Leftrightarrow \ x - y = x_0$
- $wp(\ y := 8/x,\ y > 4\ ) \ \Leftrightarrow$
  $x \neq 0 \wedge (y > 4)[y \leftarrow 8/x] \ \Leftrightarrow$
  $x \neq 0 \wedge 8/x > 4 \ \Leftrightarrow$
  $x \neq 0 \wedge (0 \leq x < 2 \vee 0 \geq x > 2) \ \Leftrightarrow \ 0 < x < 2$
- $wp(\ x := 0,\ x = 0\ ) \ \Leftrightarrow$
  $\mathsf{True} \wedge (x = 0)[x \leftarrow 0] \ \Leftrightarrow \ 0 = 0 \ \Leftrightarrow \ \mathsf{True}$
- $wp(\ x := 1,\ x = 0\ ) \ \Leftrightarrow$
  $\mathsf{True} \wedge (x = 0)[x \leftarrow 1] \ \Leftrightarrow \ 1 = 0 \ \Leftrightarrow \ \mathsf{False}$

According to the definition of *wp* we get two deduction rules for the simple assignment statement

- $\{\ Q[x \leftarrow \mathit{expression}]\ \}\ x := \mathit{expression}\ \{\ Q\ \}$
- $\langle\ \Uparrow\!\mathit{expression} \wedge Q[x \leftarrow \mathit{expression}]\ \rangle$
  $x := \mathit{expression}\ \langle\ Q\ \rangle$

The previous examples in deduction form:

- $\langle\ x > 2\ \rangle \quad x := x - 2 \quad \langle\ x > 0\ \rangle$
- $\langle\ x = x_0\ \rangle \quad y := x - y \quad \langle\ x = x_0\ \rangle$
- $\langle\ x - y = x_0\ \rangle \quad y := x - y \quad \langle\ y = x_0\ \rangle$
- $\langle\ 0 < x < 2\ \rangle \quad y := 8/x \quad \langle\ y > 4\ \rangle$
- $\langle\ \mathsf{True}\ \rangle \quad x := 0 \quad \langle\ x = 0\ \rangle$
- $\langle\ \mathsf{False}\ \rangle \quad x := 1 \quad \langle\ x = 0\ \rangle$

Warning! The previous hold only if the target of assignment is a single variable

- e.g. the formula would give
  $wp(\ A[i] := 5,\ A[i] = 5 \wedge A[j] = 0 \wedge i = j\ ) \Leftrightarrow$
  $5 = 5 \wedge A[j] = 0 \wedge i = j \Leftrightarrow$
  $A[j] = 0 \wedge i = j$,
  so we would erronously get
  $\langle\ A[j] = 0 \wedge i = j\ \rangle$
  $A[i] := 5$
  $\langle\ A[i] = 5 \wedge A[j] = 0 \wedge i = j\ \rangle$

The *wp*-semantics and deduction rules for **skip**-statement

- sometimes it is handy to use a statement that does nothing
  - **skip**-statement

- since **skip** does not change the state, thus
  - $wp(\ \mathbf{skip},\ Q\ ) \Leftrightarrow Q$
  - $\langle\ P\ \rangle\ \mathbf{skip}\ \langle\ P\ \rangle$

The *wp*-semantics for consecutively coupled statements

- if $\langle\ P\ \rangle\ S_1; S_2\ \langle\ Q\ \rangle$
  then $\langle\ P\ \rangle\ S_1\ \langle\ wp(\ S_2,\ Q\ )\ \rangle$,
  thus $P \Rightarrow wp(\ S_1,\ wp(\ S_2,\ Q\ )\ )$
- if $P \Rightarrow wp(\ S_1,\ wp(\ S_2,\ Q\ )\ )$
  then $\langle\ P\ \rangle\ S_1\ \langle\ wp(\ S_2,\ Q\ )\ \rangle\ S_2\ \langle\ Q\ \rangle$,
  thus $\langle\ P\ \rangle\ S_1; S_2\ \langle\ Q\ \rangle$
- therefore $wp(S_1; S_2, Q) \Leftrightarrow wp(S_1, wp(S_2, Q))$

Example: compute $wp(\, S,\ y = x_0 \wedge x = y_0\,)$, when S is

$$x := x + y;\ y := x - y;\ x := x - y$$

- proceed backwards
- the last statement:
  $wp(\, x := x - y,\ y = x_0 \wedge x = y_0\,) \;\Leftrightarrow$
  $y = x_0 \wedge x - y = y_0$
- two last statements:
  $wp(\, y := x - y;\, x := x - y,\, y = x_0 \wedge x = y_0\,) \;\Leftrightarrow$
  $wp(\, y := x - y,\ wp(\, x := x - y,\ y = x_0 \wedge x = y_0\,)\,) \;\Leftrightarrow$
  $wp(\, y := x - y,\ y = x_0 \wedge x - y = y_0\,) \;\Leftrightarrow$
  $x - y = x_0 \wedge x - (x - y) = y_0 \;\Leftrightarrow$
  $x - y = x_0 \wedge y = y_0$
- the whole program:
  $wp(\, x := x + y;\ y := x - y;\ x := x - y,\ y = x_0 \wedge x = y_0\,) \;\Leftrightarrow$
  $wp(\, x := x + y,\, wp(\, y := x - y;\ x := x - y,\ y = x_0 \wedge x = y_0\,)\,) \;\Leftrightarrow$
  $wp(\, x := x + y,\ x - y = x_0 \wedge y = y_0\,) \;\Leftrightarrow$
  $x + y - y = x_0 \wedge y = y_0 \;\Leftrightarrow$
  $x = x_0 \wedge y = y_0$
- the answer is $x = x_0 \wedge y = y_0$
  - thus the program swaps the values of x and y

The corresponding deduction rules

- if $\{ P \} S_1 \{ Q \}$ and $\{ Q \} S_2 \{ R \}$
  then $\{ P \} S_1; S_2 \{ R \}$
- if $\langle P \rangle S_1 \langle Q \rangle$ and $\langle Q \rangle S_2 \langle R \rangle$
  then $\langle P \rangle S_1; S_2 \langle R \rangle$

The previous example with deduction rules

$\langle x = x_0 \wedge y = y_0 \rangle$ (* last phase *)
$\quad x := x + y;$ (* notice! $(x + y) - x = x$ *)
$\langle x - y = x_0 \wedge y = x_0 \rangle$ (* third phase *)
$\quad y := x - y;$ (* notice! $x - (x - y) = y$ *)
$\langle y = x_0 \wedge x - y = y_0 \rangle$ (* second phase *)
$\quad x := x - y;$
$\langle y = x_0 \wedge x = y_0 \rangle$ (* calculations start here *)

The *wp*-semantics for the basic form of **if**-statement

- the basic form for **if**-statement is

  if B **then** S_1 **else** S_2 **endif**

- computation of B must succeed: $\Uparrow B$
- if **then**-branch is chosen then the precondition of S_1 must hold: $B \Rightarrow wp(S_1, Q)$
  - the same without the implication arrow: $\neg B \vee wp(S_1, Q)$
- if **else**-branch is chosen then the precondition of S_2 must hold: $\neg B \Rightarrow wp(S_2, Q)$

- the same without the implication arrow:
  $B \vee wp(\ S_2, Q\ )$

$\Rightarrow$ summa summarum:

$wp(\ \textbf{if}\, B\, \textbf{then}\, S_1\, \textbf{else}\, S_2\, \textbf{endif}\, ,\ Q\ )\ \Leftrightarrow$
$\Uparrow B \wedge (\neg B \vee wp(\ S_1, Q\ )) \wedge (B \vee wp(\ S_2,\ Q\ )\ )$

- this has also an alternative form that is sometimes more handy to be used in calculations:

$wp(\ \textbf{if}\, B\, \textbf{then}\, S_1\, \textbf{else}\, S_2\, \textbf{endif}\, ,\ Q\ )\ \Leftrightarrow$
$\Uparrow B \wedge (B \wedge wp(\ S_1,\ Q\ ) \vee \neg B \wedge wp(\ S_2,\ Q\ )\ )$

Example: compute
$wp(\, \textbf{if}\, x > y\ \ \textbf{then}\, d := x - y$
$\textbf{else}\, d := y - x\, \textbf{endif}\, ,\ d > 0)$

- $wp(\ d := x - y,\ d > 0\ )\ \Leftrightarrow\ x - y > 0\ \Leftrightarrow\ x > y$
- $wp(\ d := y - x,\ d > 0\ )\ \Leftrightarrow\ y - x > 0\ \Leftrightarrow\ x < y$
- thus the answer $\Leftrightarrow$
  $\mathsf{True} \wedge (\neg(x > y) \vee x > y) \wedge (x > y \vee x < y)\ \Leftrightarrow$
  $x \neq y$

The deduction rules for the basic **if** -statement

- if $\{ P \wedge B \}\ S_1\ \{ Q \}$ and $\{ P \wedge \neg B \}\ S_2\ \{ Q \}$
  then

  $\{ P \}$
  if B **then** S_1 **else** S_2 **endif**
  $\{ Q \}$

- if $P \Rightarrow \Uparrow B$, $\langle P \wedge B \rangle\ S_1\ \langle Q \rangle$, and
  $\langle P \wedge \neg B \rangle\ S_2\ \langle Q \rangle$
  then

  $\langle P \rangle$
  if B **then** S_1 **else** S_2 **endif**
  $\langle Q \rangle$

The previous example using deduction rules

$\langle\ x \neq y\ \rangle$
if $x > y$ **then** $d := x - y$ **else** $d := y - x$ **endif**
$\langle\ d > 0\ \rangle$

- the same with all interphases

$\langle\ x \neq y\ \rangle$
if $x > y$ **then**
    $\langle\ x > y\ \rangle$
    $d := x - y$
    $\langle\ d > 0\ \rangle$
else
    $\langle\ x \leq y \wedge x \neq y\ \rangle$ (* same as $x < y$ *)
    $d := y - x$
    $\langle\ d > 0\ \rangle$
endif
$\langle\ d > 0\ \rangle$

The **if** -statement without **else** -branch

- **if** B **then** S **endif** is same as
  if B **then** S **else skip endif**

$\Rightarrow$ we can calculate
$wp($ **if** B **then** S **endif**, $Q\ )\ \Leftrightarrow$
$wp($ **if** B **then** S **else skip endif** , $Q\ )\ \Leftrightarrow$
$\Uparrow B\ \wedge(\neg B \vee wp(S, Q)) \wedge (B \vee wp(\mathbf{skip}, Q))\ \Leftrightarrow$
$\Uparrow B\ \wedge(\neg B \vee wp(\ S,\ Q\ )\ ) \wedge (B \vee Q)$

- therefore

  $$wp(\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{endif},\ Q\ ) \ \Leftrightarrow$$
  $$\Uparrow B\ \wedge(\neg B \vee wp(\ S,\ Q\ )) \wedge (B \vee Q)$$

- alternative form

  $$wp(\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{endif},\ Q\ ) \ \Leftrightarrow$$
  $$\Uparrow B\ \wedge(B \wedge wp(\ S,\ Q\ )) \vee \neg B \wedge Q)$$

- by the same way we get the deduction rules:

  - if $\{\ P \wedge B\ \}\ S\ \{\ Q\ \}$ and $P \wedge \neg B \ \Rightarrow\ Q$, then

    $\{\ P\ \}$
    if B **then** S **endif**
    $\{\ Q\ \}$

  - if $P \ \Rightarrow\ \Uparrow B$, $\langle\ P \wedge B\ \rangle\ S\ \langle\ Q\ \rangle$, and
    $P \wedge \neg B \ \Rightarrow\ Q$, then

    $\langle\ P\ \rangle$
    if B **then** S **endif**
    $\langle\ Q\ \rangle$

Example

- prove (A is indexed $1 \ldots n$)

  $$\langle\ (1 \leq i \leq n \wedge \forall j\ ;\ 1 \leq j \leq i-1 : A[j] \neq x)$$
  $$\wedge\ (i \geq n \vee A[i] = x)\ \rangle$$
  $$\textbf{if}\ A[i] \neq x\ \ \textbf{then}\ i := 0\ \ \textbf{endif}$$
  $$\langle\ (1 \leq i \leq n \wedge A[i] = x\ \wedge$$
  $$\forall j\ ;\ 1 \leq j \leq i-1 : A[i] \neq x)$$
  $$\vee\ i = 0 \wedge \forall j\ ;\ 1 \leq j \leq n : A[j] \neq x\ \rangle$$

- the state predicates are:
  - $P \Leftrightarrow (1 \leq i \leq n \wedge$
    $\forall j \ ; \ 1 \leq j \leq i-1 : A[j] \neq x)$
    $\wedge \ (i \geq n \vee A[i] = x)$
  - $Q \Leftrightarrow (1 \leq i \leq n \wedge A[i] = x \wedge$
    $\forall j \ ; \ 1 \leq j \leq i-1 : A[i] \neq x)$
    $\vee \ i = 0 \wedge \forall j \ ; \ 1 \leq j \leq n : A[j] \neq x$
  - $B \Leftrightarrow A[i] \neq x$
- $P \Rightarrow 1 \leq i \leq n \Rightarrow \ \Uparrow B$ •/•
- $wp(\ i := 0,\ Q\ ) \ \Leftrightarrow$
  $(1 \leq 0 \leq n \wedge \ldots) \vee$
  $0 = 0 \wedge \forall j \ ; \ 1 \leq j \leq n : A[j] \neq x$
  $\Leftrightarrow \ \forall j \ ; \ 1 \leq j \leq n : A[j] \neq x$

  $P \wedge B \Rightarrow (1 \leq i \leq n \wedge$
  $\forall j \ ; \ 1 \leq j \leq i-1 : A[j] \neq x) \wedge$
  $i \geq n \wedge A[i] \neq x$
  $\Rightarrow \ i = n \wedge \forall j \ ; \ 1 \leq j \leq n : A[j] \neq x$
  $\Rightarrow \ wp(\ i := 0,\ Q\ )$
- $P \wedge \neg B \ \Leftrightarrow$
  $(1 \leq i \leq n \wedge$
  $\forall j \ ; \ 1 \leq j \leq i-1 : A[j] \neq x) \wedge A[i] = x$
  $\Rightarrow \ Q$ •/•

The nondeterministic (epädeterministinen) **if**-statement

- programming language theorists use often the nondeterministic **if** -statement

  if $B_1 \rightarrow S_1$
  [] $B_2 \rightarrow S_2$
  ...
  [] $B_n \rightarrow S_n$
  fi

- action
  - in the beginning of the execution of the statement every B_i must be computable
  - if no B_i does become true, then an error situation arises
  - otherwise some branch is arbitrarily chosen so that its B_i is true, and the corresponding S_i is executed

- example

  if $x \geq 0 \rightarrow S_1$
  [] $x = 0 \rightarrow S_2$
  [] $x \leq 0 \rightarrow S_3$
  fi

- an ordinary **if**-statement can be implemented by nondeterministic **if**-statement:

  if B then S_1 **else** S_2 **endif** is same as
  if $B \rightarrow S_1$ [] $\neg B \rightarrow S_2$ **fi**

- *wp*-semantics
  $wp(\ \mathbf{if}\, B_1 \rightarrow S_1\ [] \cdots []\ B_n \rightarrow S_n\, \mathbf{fi},\ Q\ )\ \Leftrightarrow$
  $\Uparrow B_1\ \wedge \cdots \wedge \Uparrow B_n\ \wedge (B_1 \vee \cdots \vee B_n) \wedge$
  $(\neg B_1 \vee wp(\ S_1,\ Q\ )\ ) \wedge\ \cdots$
  $\wedge\, (\neg B_n \vee wp(\ S_n,\ Q\ )\ )$

- deduction rules

  - if $\{\ P \wedge B_i\ \}\ S_i\ \{\ Q\ \}$ for all $i \in \{1 \ldots, n\}$, then
    $\{\ P\ \}$
    $\mathbf{if}\, B_1 \rightarrow S_1\ [] \cdots []\ B_n \rightarrow S_n\, \mathbf{fi}$
    $\{\ Q\ \}$
  - if $P\ \Rightarrow\ \Uparrow B_i$, $P\ \Rightarrow\ B_1 \vee \ldots \vee B_n$, and $\langle\ P \wedge B_i\ \rangle\ S_i\ \langle\ Q\ \rangle$, for all $i \in \{1 \ldots, n\}$, then
    $\langle\ P\ \rangle$
    $\mathbf{if}\, B_1 \rightarrow S_1\ [] \cdots []\ B_n \rightarrow S_n\, \mathbf{fi}$
    $\langle\ Q\ \rangle$

Assignment into an array

- we saw previously that the formula

  $$wp(\ x := \mathit{expression},\ Q\ )\ \Leftrightarrow$$
  $$\Uparrow \mathit{expression} \wedge Q[x \leftarrow \mathit{expression}]$$

  holds only if x is a single variable (not e.g. an element of an array)

- reason: textual substitution does not take into account that the element of an array $A[i]$ can be referred also by other name
  - e.g. $A[1]$ (if $i = 1$) or $A[j-3]$ (if $j = i+3$)
- solution: let us think that
  - the target of the assignment is the **whole array**, e.g.
    $A[2] := 3 \rightsquigarrow A[1\ldots 3] = \langle\, A[1], 3, A[3]\,\rangle$
  - textual substitution directs to the **name of the array variable without index**

$\Rightarrow$ we need a notation for the modified array that has been modified by assignment
$A[i] := e$, where i is in legal range:

$A([i] \leftarrow e)$ is an array such that

$$A([i] \leftarrow e)[k] = \begin{cases} e & \text{if } k = i, \\ A[k] & \text{otherwise.} \end{cases}$$

- the weakest precondition for assignment to an array can now be calculated by dividing – if necessary – the formula in to parts, similarly as in the following example:
  $wp(\, A[i] := e,\ Q(A[i], A[j]\,)\ \Leftrightarrow$
  $\Uparrow e \wedge 1 \leq i \leq n \wedge Q(A([i] \leftarrow e)[i], A([i] \leftarrow e)[j])\ \Leftrightarrow$
  $\Uparrow e \wedge 1 \leq i \leq n \wedge (\quad j = i \wedge Q(e, e)$
  $\qquad\qquad\qquad\qquad \vee\, j \neq i \wedge Q(e, A[j])\,)$

Examples (we assume that indexes are in legal ranges)

- $wp(\ A[i] := 2,\ A[j] = 0\ ) \ \Leftrightarrow$
  $A([i] \leftarrow 2)[j] = 0 \ \Leftrightarrow$
  $j = i \wedge 2 = 0 \vee j \neq i \wedge A[j] = 0 \ \Leftrightarrow$
  $j \neq i \wedge A[j] = 0$

- $wp(\ A[i] := 5,\ A[i] = 5 \wedge A[j] = 0 \wedge i = j\ ) \ \Leftrightarrow$
  $A([i] \leftarrow 5)[i] = 5 \wedge A([i] \leftarrow 5)[j] = 0 \wedge i = j \ \Leftrightarrow$
  $5 = 5 \wedge (j = i \wedge 5 = 0 \vee j \neq i \wedge A[j] = 0) \wedge i = j \Leftrightarrow$
  False

- $wp(\ A[A[i]+1] := 2,\ A[j] = 0\ ) \ \Leftrightarrow$
  $A([A[i]+1] \leftarrow 2)[j] = 0 \ \Leftrightarrow$
  $j = A[i]+1 \wedge 2 = 0 \vee j \neq A[i]+1 \wedge A[j] = 0 \ \Leftrightarrow$
  $j \neq A[i] + 1 \wedge A[j] = 0$

- $wp(\ A[i] := A[i] - A[j],\ A[i] = y_0 \wedge A[j] = x_0\ ) \ \Leftrightarrow$
  $\qquad j = i \wedge A[i] - A[j] = x_0 = y_0$
  $\vee \quad j \neq i \wedge A[i] - A[j] = y_0 \wedge A[j] = x_0 \ \Leftrightarrow$
  $\quad j = i \wedge x_0 = y_0 = 0$
  $\vee\ j \neq i \wedge A[i] = x_0 + y_0 \wedge A[j] = x_0$

- the whole program $A[i] := A[i] + A[j];\ A[j] := A[i] - A[j];\ A[i] := A[i] - A[j]$

$$
\begin{array}{l}
\langle\, j = i \wedge x_0 = y_0 = 0 \vee \\
\quad j \neq i \wedge A[i] = x_0 \wedge A[j] = y_0 \,\rangle \\
\qquad\qquad A[i] := A[i] + A[j]; \\
\langle\, j = i \wedge x_0 = y_0 = 0 \vee \\
\quad j \neq i \wedge A[i] = x_0 + y_0 \wedge A[j] = y_0 \,\rangle \\
\qquad\qquad A[j] := A[i] - A[j]; \\
\langle\, j = i \wedge x_0 = y_0 = 0 \vee \\
\quad j \neq i \wedge A[i] = x_0 + y_0 \wedge A[j] = x_0 \,\rangle \\
\qquad\qquad A[i] := A[i] - A[j]; \\
\langle\, A[i] = y_0 \wedge A[j] = x_0 \,\rangle
\end{array}
$$

- notices from the last example
  - the program does not compute wrongly everytime when $j = i$
  - the case $j = i$ caused extra work only by the last assignment

$\Rightarrow$ the rule for assigment to an array

  - ensures that all special cases are taken into account
  - is usually not so laborious as it seems to

Useful laws for weakest preconditions:

- *law of the excluded miracle (pois suljetun ihmeen laki)*:

$$wp(\, S,\, \mathsf{False}\,) \Leftrightarrow \mathsf{False}$$

  - in other words, there does not exists a state from where started execution of S ends, and the end state is impossible

- usually $wp(\, S,\, \mathsf{True}\,) \neq \mathsf{True}$, since wp ensures that execution of S succeeds
  - e.g. if $i \in \mathbb{Z}$ and S is
    while $i \neq 0$ **do** $i := i - 1$ **endwhile**
    then $wp(\, S,\, \mathsf{True}\,) \Leftrightarrow i \geq 0$

- if $P \Rightarrow Q$, then $wp(\, S,\, P\,) \Rightarrow wp(\, S,\, Q\,)$
- $wp(\, S,\, P \wedge Q\,) \Leftrightarrow wp(\, S,\, P\,) \wedge wp(\, S,\, Q\,)$
- $wp(\, S,\, P \vee Q\,) \Leftarrow wp(\, S,\, P\,) \vee wp(\, S,\, Q\,)$
- $wp(\, S,\, P \vee Q\,) \not\Rightarrow wp(\, S,\, P\,) \vee wp(\, S,\, Q\,)$
  - a counterexample:
    if S assigns to x a random number 0 or 1
    $x := \mathit{random}(0, 1)$
    then
    $wp(\, S,\, x = 0 \vee x = 1\,) \Leftrightarrow \mathsf{True} \not\Rightarrow$
    $wp(\, S,\, x = 0\,) \vee wp(\, S,\, x = 1\,) \Leftrightarrow$
    $\mathsf{False} \vee \mathsf{False} \Leftrightarrow \mathsf{False}$

– another way to assign 0 or 1 to x:

$$\textbf{if}\ \textsf{True} \rightarrow x := 0\ []\ \textsf{True} \rightarrow x := 1\ \textbf{fi}$$

- S is *deterministic (deterministinen)* if and only if for all states t the following holds when S has been started in state t:

  if S can terminate and produce
  an end state t'
  then S definitely terminates and produces
  the end state t'

  – in other words, if S can do something, then it definitely does it

  $\Rightarrow$ for the action of S there is only one possible result

- if S is deterministic then
  $wp(\ S,\ P \vee Q\ )\ \Leftrightarrow\ wp(\ S,\ P\ ) \vee wp(\ S,\ Q\ )$

3.3 Proving Loop Statements

Consider the following:

- A coffee can contains black and white beans. The following process is repeated as long as possible:

  > Randomly select two beans from the can. If they are the same color, throw them away but place one black bean into the can. (Assume an adequate supply of black beans.) If they are different colors, throw the black one away and place the white bean back into the can.

- Determine the relation between the initial contents of the coffee can and the color of the final bean that remains.

The *wp*-semantics of a loop statement

- the base form of a loop statement is

  while B **do** S **endwhile**

  where

  - B is a condition
  - S is a statement (or a sequence of statements)

- for simplicity we assume that B is always computable (in other words, $\Uparrow B \;\Leftrightarrow\; \mathsf{True}$)
- the weakest precontion for that the loop statement is executed 0 time and the result satisfies Q is

$$P_0 \;\Leftrightarrow\; \neg B \wedge Q$$

- the weakest precontion for that the loop statement is executed 1 time and the result satisfies Q is

$$P_1 \Leftrightarrow B \wedge wp(S, \neg B \wedge Q) \Leftrightarrow B \wedge wp(S, P_0)$$

- similarly for two execution times we get

$$P_2 \;\Leftrightarrow\; B \wedge wp(\; S,\; P_1 \;)$$

- and for n execution times

$$P_n \;\Leftrightarrow\; B \wedge wp(\; S,\; P_{n-1} \;)$$

- if the execution of the loop statement succeeds and the result satisfies Q then the loop is circled n times for some finite n and in the beginning P_n holds; similarly if this kind of n exists then execution of the loop statement succeeds and the result satisfies Q.

$$\Longrightarrow\; wp(\; \textbf{while}\; B\; \textbf{do}\; S\; \textbf{endwhile}\,,\; Q\;) \Leftrightarrow \exists n : P_n$$

The *wp*-semantics for loop statement is difficult (or impossible) to use in calculations

- e.g. what is $wp(\ S,\ \mathsf{True}\ )$, when S is:

  while $x \neq 1$ **do**
  if $x \ \mathsf{mod}\ 2 = 0$ **then** $x := x \ \mathsf{div}\ 2$
  else $x := 3 \cdot x + 1$ **endif**
  endwhile

  The Collatz conjecture is an unsolved conjecture first proposed in 1937 by Lothar Collatz:

  > "Take any natural number n. If n is even, divide it by 2 to get $n/2$, if n is odd multiply it by 3 and add 1 to obtain $3n+1$. Repeat the process indefinitely. The conjecture is that no matter what number you start with, you will always eventually reach 1."

  Paul Erdus said about the Collatz conjecture: "Mathematics is not yet ready for such problems." He offered $500 for its solution.

  In 2006, researchers Kurtz and Simon, proved that a natural generalization of the Collatz problem is undecidable. However, as this proof depends upon the generalization, it cannot be applied to the original Collatz problem.

- theory of solvability

$\Rightarrow$ there are situations where it is not even in principle possible to find out whether the following holds:

$$P \Rightarrow wp(\ \textbf{while}\ ,\ \mathsf{True}\ )$$

$\Rightarrow$ for loop statements we use other means

- partial correctness: *invariant (invariantti)*
- termination: *bound function (ylärajafunktio)*

Proving partial correctness of a loop statement by invariants

- claim I is the *invariant (invariantti)* of the loop statement

  $$\textbf{while}\ B\ \ \textbf{do}\ S\ \ \textbf{endwhile}$$

  if and only if

  $$\{\ I \wedge B\ \}\ S\ \{\ I\ \}$$

- if I is an invariant for the loop statement then

  $$\{\ I\ \}\ \textbf{while}\ B\ \ \textbf{do}\ S\ \ \textbf{endwhile}\ \{\ I \wedge \neg B\ \}$$

- a claim of the form

  $$\{\ P\ \}\ \textbf{while}\ B\ \ \textbf{do}\ S\ \ \textbf{endwhile}\ \{\ Q\ \}$$

  can therefore be proved by the following:

  1. find a suitable candidate for invariant: I

2. prove that $P \Rightarrow I$
3. prove that $\{ I \wedge B \} \; S \; \{ I \}$
4. prove that $I \wedge \neg B \Rightarrow Q$

- it is not yet necessary to prove that $\Uparrow B$

Example: prove that ($n, i \in \mathbb{Z}, a, x \in \mathbb{R}$,
assume $a \neq 0$)

$\{ n \geq 0 \}$
$x := 1;\; i := 0;$
while $i < n$ **do**
  $x := x \cdot a;\; i := i + 1$
endwhile
$\{ x = a^n \}$

- beginning of the solution

  $\{ n \geq 0 \}$
  $x := 1;\; i := 0;$
  $\{ x = 1 \wedge i = 0 \wedge n \geq 0 \}$

- processing the loop

1. candidate for invariant: $x = a^i \wedge i \leq n$
2. $x = 1 \wedge i = 0 \wedge n \geq 0 \Rightarrow x = a^i \wedge i \leq n$
   •/•
3. $wp(\, x := x \cdot a;\; i := i + 1,\; x = a^i \wedge i \leq n \,)$
   $\Leftrightarrow$
   $wp(\, x := x \cdot a,\; x = a^{i+1} \wedge i + 1 \leq n \,)$

$\Leftrightarrow$
$x \cdot a = a^{i+1} \wedge i + 1 \leq n \;\; \Leftrightarrow$
$x = a^i \wedge i < n$

therefore

$\{\ x = a^i \wedge i \leq n \wedge i < n\ \}$
$x := x \cdot a;\ i := i + 1$
$\{\ x = a^i \wedge i \leq n\ \}$

4. $x = a^i \wedge i \leq n \wedge \neg(i < n) \;\; \Leftrightarrow$
   $x = a^i \wedge i = n \;\Rightarrow\; x = a^n$

- all together

$\{\ n \geq 0\ \}$
$x := 1;\ i := 0;$
$\{\ x = 1 \wedge i = 0 \wedge n \geq 0\ \}$
$\{$ **inv:** $x = a^i \wedge i \leq n\ \}$
while $i < n$ **do**
  $x := x \cdot a;\ i := i + 1$
endwhile
$\{\ x = a^n\ \}$

Proving termination of a loop statement by using bound function

- *bound function (variant, ylärajafunktio)* is a function (usually integer valued) which
  - has a bound (often 0), for which and its smaller values new cycles are not started

- value decreases in every cycle at least by one unit

⇒ the loop cannot circulate infinitely

- especially if
  - the bound for the bound function is 0 and
  - when entering the loop the value of bound function is n

  then the loop can circulate at most n rounds

- for proving properties of a bound function it is usually utilised the invariant chosen for proving partial correctness

  ⇒ it may be necessary to choose a stronger invariant than otherwise would be needed

- so: termination of the loop

  while B **do** S **endwhile**

  outfitted with invariant I can be proven in the following way:

  1. find a suitable bound function $bf : States \rightarrow \mathbb{Z}$
  2. prove that $I \Rightarrow \Uparrow B$
  3. prove that $I \wedge B \Rightarrow bf > 0$
  4. prove that $\langle\, I \wedge B \wedge bf = bf_o \,\rangle\; S \;\langle\, bf < bf_0 \,\rangle$, where bf_0 is an arbitrary integer

Example: prove termination of the previous program

- expressions of assignment statements are always computable

  $\Rightarrow$ they succeed and terminate

$\langle\, n \geq 0 \,\rangle$
$x := 1;\ i := 0;$
$\langle\, x = 1 \wedge i = 0 \wedge n \geq 0 \,\rangle$
$\{$ **inv:** $x = a^i \wedge i \leq n \}$
while $i < n$ **do**
  $x := x \cdot a;\ i := i + 1$
endwhile
$\{\, x = a^n \,\}$

- proving termination of the loop

  1. bound function: $n - i$
  2. $i < n$ is always computable
  3. $x = a^i \wedge i \leq n \wedge i < n \ \Rightarrow$
     $i < n \ \Leftrightarrow \ n - i > 0$
  4. $wp(\ x := x \cdot a;\ i := i + 1,\ n - i < bf_0\ ) \ \Leftrightarrow$
     $n - (i + 1) < bf_0 \ \Leftrightarrow \ n - i \leq bf_0 \ \Leftarrow$
     $n - i = bf_0$
     therefore
     $\langle\ x = a^i \wedge i \leq n \wedge i < n \wedge n - i = bf_0\ \rangle$
     $x := x \cdot a;\ i := i + 1$
     $\langle\ n - i < bf_0\ \rangle$

Example: proving the correctness of search program

- prove total correctness of the following program when $n \geq 1$, and A and x are fixed:

$\langle\ n \geq 1\ \rangle$
$i := 1;$
while $i < n \wedge A[i] \neq x$ **do** $i := i + 1$
endwhile ;
if $A[i] \neq x$ **then** $i := 0$ **endif**
$\langle\ 1 \leq i \leq n \wedge A[i] = x \wedge$
$\forall j\ ;\ 1 \leq j \leq i - 1 : A[j] \neq x$
$\vee\ i = 0 \wedge \forall j\ ;\ 1 \leq j \leq n : A[j] \neq x\ \rangle$

- symbolic execution of the ass. statement gives
  $\langle\ n \geq 1\ \rangle\ i := 1\ \langle\ n \geq 1 \wedge i = 1\ \rangle$

- loop invariant and its proofs:

  1. $I :\Leftrightarrow$
     $1 \leq i \leq n \wedge$
     $\forall j\ ;\ 1 \leq j \leq i-1 : A[j] \neq x$ (in other words, i is in legal range and its precedings do not contain the searched value)

  2. $n \geq 1 \wedge i = 1 \ \Rightarrow\ I$ •/•

  3. $wp(\ i := i+1,\ I\ ) \ \Leftrightarrow$
     $1 \leq i+1 \leq n \wedge$
     $\forall j\ ;\ 1 \leq j \leq i : A[j] \neq x \ \Leftrightarrow$
     $0 \leq i < n \wedge \forall j\ ;\ 1 \leq j \leq i : A[j] \neq x$
     on the other hand $I \wedge B \ \Leftrightarrow$
     $1 \leq i \leq n \wedge$
     $\forall j\ ;\ 1 \leq j \leq i-1 : A[j] \neq x \wedge i < n \wedge$
     $A[i] \neq x \ \Leftrightarrow$
     $1 \leq i < n \wedge \forall j\ ;\ 1 \leq j \leq i : A[j] \neq x \ \Rightarrow$
     $wp(\ i := i+1,\ I\ )$
     therefore
     $\{\ I \wedge i < n \wedge A[i] \neq x\ \}\ i := i+1\ \{\ I\ \}$

  4. $I \wedge \neg B \ \Leftrightarrow$
     $1 \leq i \leq n \wedge \forall j\ ;\ 1 \leq j \leq i-1 : A[j] \neq x \wedge$
     $(i \geq n \vee A[i] = x)$

$\Longrightarrow$ $\{\ n \geq 1 \wedge i = 1\ \}$
while $i < n \wedge A[i] \neq x$ **do** $i := i+1$
endwhile
$\{\ 1 \leq i \leq n \wedge \forall j\ ;\ 1 \leq j \leq i-1 : A[j] \neq x$
$\wedge\ (i \geq n \vee A[i] = x)\ \}$

- bound function and its proofs:

  1. $bf = n - i$
  2. $I \ \Rightarrow \ 1 \leq i \leq n$, thus
     $I \ \Rightarrow \ \Uparrow(i < n \wedge A[i] \neq x)$
  3. $I \wedge i < n \wedge A[i] \neq x \ \Rightarrow \ n - i > 0$ •/•
  4. $wp(\, i := i + 1,\ n - i < bf_0 \,) \ \Leftrightarrow$
     $n - (i + 1) < bf_0 \ \Leftrightarrow$
     $n - i \leq bf_0$, thus
     $\langle\, I \wedge B \wedge n - i = bf_0 \,\rangle\ i := i + 1\ \langle\, n - i < bf_0 \,\rangle$

  $\Rightarrow$ the loop terminates

- we have now proved that

$$
\begin{array}{l}
\langle\, n \geq 1 \,\rangle \\
i := 1; \\
\ \textbf{while}\ i < n \wedge A[j] \neq x\ \ \textbf{do}\ i := i + 1 \\
\ \textbf{endwhile} \\
\langle\, 1 \leq i \leq n \wedge \forall j\ ;\ 1 \leq j \leq i - 1 : A[j] \neq x \\
\quad \wedge\, (i \geq n \vee A[i] \neq x) \,\rangle
\end{array}
$$

- the rest was proved in the previous subsection

$$
\begin{array}{l}
\langle\, 1 \leq i \leq n \wedge \forall j\ ;\ 1 \leq j \leq i - 1 : A[j] \neq x \\
\quad \wedge\, (i \geq n \vee A[i] = x) \,\rangle \\
\ \textbf{if}\ A[i] \neq x\ \ \textbf{then}\ i := 0\ \ \textbf{endif} \\
\langle\, 1 \leq i \leq n \wedge A[i] = x \wedge \\
\quad \forall j\ ;\ 1 \leq j \leq i - 1 : A[j] \neq x \\
\quad \vee\, i = 0 \wedge \forall j\ ;\ 1 \leq j \leq n : A[j] \neq x \,\rangle
\end{array}
$$

Combining the phases

- in the above we proved separately the partial correctness and termination of a loop
- in practice it is often handy to do them together

⇒ a claim of the form

$$\langle\ P\ \rangle\ \textbf{while}\ B\ \ \textbf{do}\ S\ \ \textbf{endwhile}\ \langle\ Q\ \rangle$$

can be proven in the following way:

1. find a suitable candidate for invariant: I
2. find a suitable candidate for bound function: bf
3. prove that $P \Rightarrow I$
4. prove that $I \Rightarrow \Uparrow B$
5. prove that $I \wedge B \Rightarrow bf > 0$
6. prove that $\langle\ I \wedge B \wedge bf = bf_0\ \rangle\ S\ \langle\ I \wedge bf < bf_0\ \rangle$, where bf_0 is an arbitrary integer
7. prove that $I \wedge \neg B \Rightarrow Q$

Example: we prove that the binary search program terminates and in the end the width of search range $a, \ldots, y$ is 1.

$\langle$ True $\rangle$
$a := 1;\ y := n + 1;$
while $a < y$ **do**
  $v := (a + y)$ div $2;$
  if $A[v] < key$ **then** $a := v + 1$
  else $y := v$
  endif
endwhile
$\langle\ 1 \leq a = y \leq n + 1\ \rangle$

- $A[1 \ldots n]$ and *key* are fixed
- initialisations are easy to process:

$\langle$ True $\rangle$
$a := 1;\ y := n + 1;$
$\langle\ a = 1 \wedge y = n + 1\ \rangle$
while $a < y$ **do**
  $v := (a + y)$ div $2;$
  if $A[v] < key$ **then** $a := v + 1$
  else $y := v$
  endif
endwhile
$\langle\ 1 \leq a = y \leq n + 1\ \rangle$

- the state predicates needed for proving the loop:

- $P \Leftrightarrow a = 1 \wedge y = n + 1$
- $Q \Leftrightarrow 1 \leq a = y \leq n + 1$
- $B \Leftrightarrow a < y$

1. invariant:
   - in the beginning the search range is $1, \ldots, n+1$ and it should narrow until its width is 1

   $\Rightarrow$ we choose: $I \Leftrightarrow 1 \leq a \leq y \leq n + 1$

2. bound function:
   - the range $a, \ldots, y$ should narrow in every cycle

   $\Rightarrow$ we choose $bf = y - a$

3. $P \Rightarrow I$:
   - because of the general assumptions $n \geq 0$ and thus $n + 1 \geq 1$
   - therefore $P \Rightarrow 1 = a \leq y = n + 1 \Rightarrow I$ •/•

4. $I \Rightarrow \Uparrow B$:
   - the condition $a < y$ is always computable

5. $I \wedge B \Rightarrow bf_0 > 0$:
   - $I \wedge B \Rightarrow B \Leftrightarrow a < y \Rightarrow bf = y - a > 0$ •/•

6. $\langle\ I \wedge B \wedge bf = bf_0\ \rangle\ S\ \langle\ I \wedge bf < bf_0\ \rangle$
   - laborious, we shall prove it soon
7. $I \wedge \neg B\ \Rightarrow\ Q$:
   - $I \wedge \neg B\ \Rightarrow\ 1 \leq a \leq y \leq n+1 \wedge a \geq y\ \Rightarrow$ $1 \leq a = y \leq n+1\ \Leftrightarrow\ Q$ •/•

Proof for item 6

- the branches of **if** -statement give us
  - $wp(\ a := v+1,\ I \wedge bf < bf_0\ )\ \Leftrightarrow$ $1 \leq v+1 \leq y \leq n+1 \wedge y-(v+1) < bf_0$
  - $wp(\ y := v,\ I \wedge bf < bf_0\ )\ \Leftrightarrow$ $1 \leq a \leq v \leq n+1 \wedge v-a < bf_0$
- using this we could calculate $wp(\mathbf{if}, I \wedge bf < bf_0)$ but it seems to produce a big expression (remember that $wp(\mathbf{if}B\mathbf{then}S_1\mathbf{else}S_2\mathbf{endif}, Q) \Leftrightarrow$ $\Uparrow B\ \wedge(\neg B \vee wp(\ S_1,\ Q\ )) \wedge (B \vee wp(\ S_2,\ Q\ ))\ \Leftrightarrow$ $\Uparrow B\ \wedge(B \wedge wp(\ S_1,\ Q\ ) \vee \neg B \wedge wp(\ S_2,\ Q\ ))$)
- the beginning of the body of the loop seems easy

$\Rightarrow$ we simulate the beginning of the body of the loop and use the deduction rule for **if** -statement

- the above *wp*-formulas contains v only in comparisons to a, $(y-1)$, etc.

⇒ it seems to be that essential is only the value of v in comparisons to a and y

⇒ for the value of v we consider only it

- The beginning of body of the loop:
  - $I \wedge B \wedge bf = bf_0 \Leftrightarrow$
    $1 \leq a < y \leq n+1 \wedge y - a = bf_0$
  - $a < y \Rightarrow$
    $a < (a+y)/2 < y$ and
    $(a+y)/2 - \frac{1}{2} \leq (a+y) \text{ div } 2 \leq (a+y)/2$
    $\Rightarrow a \leq (a+y) \text{ div } 2 < y$

$$\langle\ 1 \leq a < y \leq n+1 \wedge y - a = bf_0\ \rangle$$
$$v := (a+y) \text{ div } 2;$$
$$\langle\ 1 \leq a \leq v < y \leq n+1 \wedge y - a = bf_0\ \rangle$$

- now we can prove the *wp*-conditions for the branches of the **if** -statement
  - $1 \leq a \leq v < y \leq n+1 \wedge$
    $y - a = bf_0 \wedge A[v] < key \Rightarrow$
    $1 \leq v+1 \leq y \leq n+1 \wedge$
    $y - (v+1) < y - a = bf_0 \Rightarrow$
    $wp(\ a := v+1,\ I \wedge bf < bf_0\ )$
  - $1 \leq a \leq v < y \leq n+1 \wedge$
    $y - a = bf_0 \wedge A[v] \geq key \Rightarrow$
    $1 \leq a \leq v < y \leq n+1 \wedge$
    $v - a < y - a = bf_0 \Rightarrow$
    $wp(\ y := v,\ I \wedge bf < bf_0\ )$

$\Longrightarrow$ $\langle\ I \wedge B \wedge bf = bf_0\ \rangle$
$v := (a + y)\ \mathsf{div}\ 2$:
$\langle\ 1 \leq a \leq v < y = n + 1 \wedge y - a = bf_0\ \rangle$
if $A[v] < key$ **then** $a := v + 1$
else $y := v$
endif
$\langle\ I \wedge bf < bf_0\ \rangle$

Notices

- we proceeded forwards (using symbolic execution) or backwards (wp) depending on which was the easiest way
- first the easy calculations were made

$\Rightarrow$ as much as possible support for difficult parts

- while simulating the statement $v := (a+y)\ \mathsf{div}\ 2$ we did not save the exact value of v
  - essential was only to prove that $y - a$ decreases
  - the exact value would have complicated later calculations
  - the appropriate weakening of the state predicate!
- the ordinary condition of binary search
  $\forall i\ ;\ 1 \leq i \leq n-1 : A[i] \leq A[i+1]$
  was not needed at all

  - the algorithm terminates and in the end
    $1 \leq a = y \leq n+1$
    even the array would not be in order
  - fault-tolerance!

Partial orders (osittaisjärjestykset)

- *partial order (osittaisjärjestys)* for a set A is any relation "$\leq$" between the elements of A, if it holds that $\forall a, b, c \in A$:
  - $a \leq a$ (*reflexivity*)
  - $a \leq b \wedge b \leq c \Rightarrow a \leq c$ (*transitivity*)
  - $a \leq b \wedge b \leq a \Rightarrow a = b$ (*antisymmetry*)
- examples
  - "$\leq$" is a partial order for real numbers
  - "$\subseteq$" is a partial order for sets
  - "fits in" is a partial order for the following boxes if the sides are infinite narrow

- partial order differs from *total order (täysi järjestys)* in only that for partial order it is not required that $\forall a, b \in A : a \leq b \vee b \leq a$

  - "$\leq$" is a total order for real numbers
  - "$\subseteq$" is not a total order for sets, since $\neg(\{1\} \subseteq \{2\}) \wedge \neg(\{2\} \subseteq \{1\})$
  - "fits in" is not a total order for boxes (why not?)

- total order is also called as *linear order (lineaarinen järjestys)*
- if "$\leq$" is a partial order for A then we define $\forall a, b \in A:$
  - $a < b \Leftrightarrow a \leq b \wedge a \neq b$
  - $a \geq b \Leftrightarrow b \leq a$
  - $a > b \Leftrightarrow b < a$
  - $a \not\geq b \Leftrightarrow \neg(a \geq b)$ (**notice!** it is not same as $a < b$)
  - similarly for other negated relation symbols

Generalised bound function

- let A be a set and "$\leq$" a partial order for it
- a pair $(A,$"$\leq$"$)$ is *well-founded (hyvin perustettu)* if and only if there does not exist an infinite sequence $a_1, a_2, a_3, \ldots \in A$ such that $a_1 > a_2 > a_3 > \cdots$
- examples

- $(\mathbb{N}, \text{“}\leq\text{”})$ is well-founded: if $a_1 = n$ then in the descending chain there is $\leq n+1$ elements
- $(\mathbb{Q}^+, \text{“}\leq\text{”})$ is not well-defined:

  $$\frac{1}{1} > \frac{1}{2} > \frac{1}{3} > \cdots$$

- $(\mathbb{N} \times \mathbb{N}, \text{“}\leq\text{”})$ is well-founded when

  $$(n_1, m_1) > (n_2, m_2) \Leftrightarrow$$
  $$n_1 > n_2 \vee (n_1 = n_2 \wedge m_1 > m_2)$$

  try it!

- for a bound function any function of a state suits if its value has the following properties:
  - it belongs to some well-founded set (=: the set of a well-founded pair)
  - it decreases in every cycle
- knowing the initial value does not necessary guarantee that the number of cycles can be estimated
  - if the value of a bound function $\in \mathbb{N} \times \mathbb{N}$ then the following holds for any n:

    $$(1,0) > (0,n) > (0,n-1) > \cdots >$$
    $$(0,1) > (0,0)$$

  $\Rightarrow$ there exist descending chains of arbitrary lengths that begin $(1,0)$

  - anyway, an infinite descending chain is not possible

About chosing an invariant and a bound function

- there is not (or not even in most cases) a working automatic method for choosing an invariant and a bound function

  ⇒ in the chosing process human creativity must be used

- the invariant reflects the “basic idea” of the loop

⇒ if creation of an invariant does not in any way success then

  - maybe even the programmer himself does not quite understand why his loop should work

  ⇒ probably the loop is incorrect

- the bound function represents some quantity by which measured the execution proceeds all the time

- there is a lot of freedom in choosing the bound function

  - the decreasing in a cycle can be anything as long as it is at least 1 (or 1 unit)

  - the lower bound needs not be 0 — it suffices that it is some integer ($-\infty$ is not acceptable)
  - the bound function needs not to produce integers but any well-founded set suffices for its image (then the lower bound is not an integer, neither)

⇒ if the creation of bound function does not in any way success then

  - maybe even the programmer himself does not quite understand why his loop should terminate

  ⇒ probably the loop is incorrect

Nondeterministic loop statement

- programming language theorists use often nondeterministic loop statement

$$
\begin{array}{ll}
\textbf{do} & B_1 \rightarrow S_1 \\
[] & B_2 \rightarrow S_2 \\
\ldots & \\
[] & B_n \rightarrow S_n \\
\textbf{od} &
\end{array}
$$

- its action
  - in the beginning of every cycle every B_i must be computable
  - if no B_i comes true then the loop is exited
  - in other case some branch is arbitrarily chosen so that its B_i comes true, the corresponding S_i is executed, and return to the beginning of the loop
- a nondeterministic loop statement can be build using an ordinary loop statement and a nondeterministic **if** -statement:

$$\textbf{do}\ B_1 \rightarrow S_1\ [] \cdots\ []\ B_n \rightarrow S_n\ \ \textbf{od}$$

  is same as

$$\begin{array}{l} \textbf{while}\ B_1 \vee \cdots \vee B_n\ \ \textbf{do} \\ \quad \textbf{if}\ B_1 \rightarrow S_1\ [] \cdots\ []\ B_n \rightarrow S_n\ \ \textbf{fi} \\ \textbf{endwhile} \end{array}$$

- an example:

$$\begin{array}{ll} \textbf{do} & A[1] > A[2] \rightarrow \mathit{swap}(A[1], A[2]) \\ {[]} & A[1] > A[3] \rightarrow \mathit{swap}(A[1], A[3]) \\ {[]} & A[2] > A[3] \rightarrow \mathit{swap}(A[2], A[3]) \\ \textbf{od} & \end{array}$$

  - if it terminates then $A[1] \leq A[2] \leq A[3]$

- bound function: the amount of pairs (i, j) such that $i < j$ but $A[i] > A[j]$ (why it suits?)

  $\Rightarrow$ yes, it terminates

- the loop arranges the array

3.4 Proof Examples

In the previous sections basic techniques for proving the correctness of a program were introduced

- in practical proofs they can be applied and combined in many ways
- in this section we go trough a set of examples

An example: removing a element from an array preserving the ordering

- the task of the program is to remove the element i from the array $A[1 \ldots n]$ by moving the other elements one step backwards
- the original program

```
for j := i  to n − 1  do
    A[j] := A[j + 1]
endfor ;
n := n − 1
```

- for the end state it is required that
  - i is fixed, this time n is not fixed
  - the beginning is unchanged: $\forall k \ ; \ 1 \leq k < i : A[k] = A_0[k]$
  - the ending is moved: $\forall k \ ; \ i \leq k \leq n : A[k] = A_0[k+1]$

- the size of the array has decreased: $n = n_0 - 1$
- for the name of the initial state of the array is given A_0 and for its size n_0

- there is going to become long formulas which are difficult to read

⇒ let us define auxiliary predicates:

- unchanged: $Unchd(a, b) :\Leftrightarrow \forall k \ ; \ a \le k < b : A[k] = A_0[k]$
- moved: $Movd(a, b) :\Leftrightarrow \forall k \ ; \ a \le k < b : A[k] = A_0[k+1]$
- a clarifying convention: a range is announced from the beginning to after its end

- the requirement concerning the end state changes to form:

$$n = n_0 - 1 \wedge Unchd(1, i) \wedge Movd(i, n+1)$$

- the specification

$\langle\ n = n_0 \wedge A[1 \ldots n] = A_0\ \rangle$
for $j := i$ **to** $n - 1$ **do**
  $A[j] := A[j+1]$
endfor ;
$n := n - 1$
$\langle\ n = n_0 - 1 \wedge Unchd(1, i) \wedge Movd(i, n+1)\ \rangle$

- soon we are going to notice that it has an error (which?)

Beginning of the solution

- let us modify the loop to **while** -loop

$\langle\ A = A_0\ \rangle$
$j := i$
while $j < n$ **do**
  $A[j] := A[j+1];$
  $j := j + 1$
endwhile ;
$n := n - 1$
$\langle\ n = n_0 - 1 \wedge \mathit{Unchd}(1, i) \wedge \mathit{Movd}(i, n+1)\ \rangle$

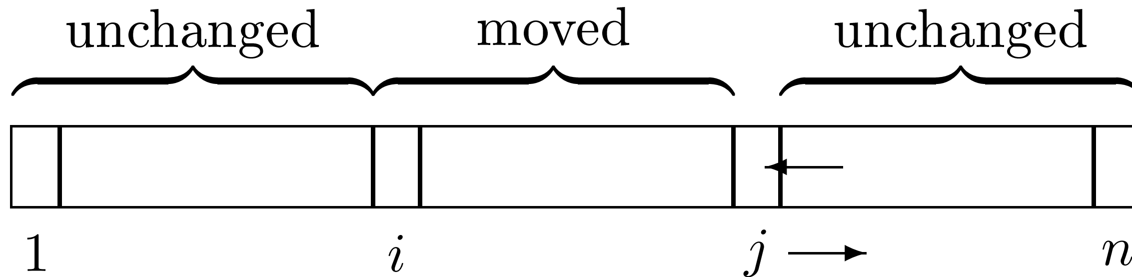
- let's calculate the easy assignment statements (using symbolic execution and *wp*-technique)

$\Rightarrow$ the task reduces to proving the loop

$\langle\ A = A_0\ \rangle$
$j := i$
$\langle\ A = A_0 \wedge j = i\ \rangle$
while $j < n$ **do**
  $A[j] := A[j+1];$
  $j := j + 1$
endwhile ;
$\langle\ n = n_0 \wedge \mathit{Unchd}(1, i) \wedge \mathit{Movd}(i, n)\ \rangle$
$n := n - 1$
$\langle\ n = n_0 - 1 \wedge \mathit{Unchd}(1, i) \wedge \mathit{Movd}(i, n+1)\ \rangle$

Designing the loop invariant

- the action of the program can be illustrated as a picture



- for this base let us make the loop invariant
  - n has not changed: $n = n_0$
  - the beginning is unchanged: $Unchd(1, i)$
  - the middle part is moved: $Movd(i, j)$
  - the ending is unchanged: $Unchd(j + 1, n + 1)$

$\Longrightarrow$ $Inv_1 :\Leftrightarrow n = n_0 \land$
$Unchd(1, i) \land Movd(i, j) \land Unchd(j + 1, n + 1)$

Proof for the loop invariant

2. the loop invariant must hold in the beginning of the loop

- by assigning $j = i$ we get for the target

$$n = n_0 \land$$
$$Unchd(1, i) \land Movd(i, i) \land Unchd(i + 1, n + 1)$$

$\Leftrightarrow \; n = n_0 \wedge Unchd(1,i) \wedge \; Unchd(i+1, n+1)$

- $A = A_0$ guarantees that $n = n_0 \wedge Unchd(1, n+1)$

$\Rightarrow$ question: does this imply $Unchd(1,i)$ and $Unchd(i+1, n+1)$?

- answer: only if $i \leq n+1$ and $i+1 \geq 1$

$\Rightarrow$ precondition $0 \leq i \leq n+1$ must be added to the specification

$\langle \; A = A_0 \wedge 0 \leq i \leq n+1 \; \rangle$
for $j := i$ **to** $n-1$ **do**
$\ldots$

  - what does the program do if in the beginning $i > n+1$? What if $i = n+1$?
  - is it correct?

4. the loop invariant and the exit condition must guarantee the condition after the loop

- the exit condition is $j \geq n$

$\Rightarrow$ we get

$j \geq n \wedge n = n_0 \wedge Unchd(1,i) \wedge Movd(i,j) \wedge Unchd(j+1, n+1)$

- but we need $n = n_0 \wedge Unchd(1,i) \wedge Movd(i,n)$

$\Rightarrow$ it is worth to compare formulas $Movd(i,j)$ and $Movd(i,n)$

- $j \geq n \wedge Movd(i, j)$ guarantees $Movd(i, n)$, but speaks about elements that are outside the array when $j > n$

⇒ probably in the end there should hold $j = n$

- question: but does not the loop guarantee that in the end $j = n$?
  answer: only if in the beginning $j \leq n$

⇒ to the specification of the program the precondition $i \leq n$ must be added

$\langle\ A = A_0 \wedge 0 \leq i \leq n\ \rangle$
for $j := i$ to $n - 1$ **do**
$\quad\quad \ldots$

- what does the program do if in the beginning $i = n + 1$? What if $i = n$?
- is it correct?

- now it is easy to see that if the end of the loop is reached then there holds $j = n$

$$
\begin{array}{l}
\langle\ i \leq n\ \rangle \\
\ j := i \\
\langle\ j \leq n\ \rangle \\
\ \textbf{while}\ j < n\ \ \textbf{do} \\
\qquad \{\ j < n\ \} \\
\qquad A[j] := A[j+1];\ j := j+1 \\
\qquad \{\ j \leq n\ \} \\
\ \textbf{endwhile}\ ; \\
\{\ j \leq n \wedge j \geq n\ \} \\
\{\ j = n\ \}
\end{array}
$$

- now we get from the exit condition and the invariant

$$
\begin{array}{ll}
& j = n \wedge n = n_0 \wedge Unchd(1, i) \wedge Movd(i, j) \wedge \\
& Unchd(j+1, n+1) \\
\Rightarrow & n = n_0 \wedge Unchd(1, i) \wedge Movd(i, n)
\end{array}
$$

- the addition for specification does not break the proof for part 2
  - we proved that the original precondition $\Rightarrow \cdots \Rightarrow Inv$
  - so of course the original precondition $\wedge i \leq n \Rightarrow \cdots \Rightarrow Inv$

3. the invariant and the loop condition must guarantee the invariant in the end of the loop

- let us calculate the weakest precondition

$$wp(\ A[j] := A[j+1]; j := j+1,\ Inv\ )$$

- $Inv_1 \ \Leftrightarrow$
  $n = n_0 \wedge Unchd(1, i) \wedge Movd(i, j) \wedge Unchd(j+1, n+1)$

  $wp(\ A[j] := A[j+1]; j := j+1,\ Inv\ ) \ \Leftrightarrow$
  $\quad n = n_0 \wedge Unchd(1, i) \wedge Movd(i, j+1) \wedge Unchd(j+2, n+1)$

  $wp(\ A[j] := A[j+1]; j := j+1,\ Inv\ ) \Leftrightarrow$??

- wp requires that the indexings are legal

  $\Rightarrow$ it must be $1 \leq j \leq n$ and $1 \leq j+1 \leq n$, in other words, $1 \leq j \leq n$

  - $j < n$ holds
  - $1 \leq j$ is needed $\Rightarrow \cdots$

  $\Rightarrow$ for the invariant $1 \leq j$ and for the precondition $1 \leq i$ must be added

$\Longrightarrow$ new invariant $Inv_2 :\Leftrightarrow$
$n = n_0 \wedge 1 \leq j \leq n \wedge Unchd(1, i) \wedge Movd(i, j) \wedge Unchd(j+1, n+1)$

- strenghtening of invariant does not break the proof for part 4
- the proof for part 2 must be redone but it is easy

- so, now we must calculate

$$\begin{aligned}&wp(\ A[j] := A[j+1],\\&n = n_0 \wedge 1 \leq j+1 \leq n \wedge \mathit{Unchd}(1,i) \wedge\\&\mathit{Movd}(i,j+1) \wedge \mathit{Unchd}(j+2,n+1)\ )\end{aligned}$$

- *Unchd* and *Movd* speak about A
  - we need the information to which of them "$A[j] :=$" affects
  - it affects part $\mathit{Unchd}(1,i)$ iff $1 \leq j < i$
  - it affects part $\mathit{Movd}(i,j+1)$ iff $i \leq j$
  - it does not affect part $\mathit{Unchd}(j+2,n+1)$
- it is easy to see that all the time holds $i \leq j$
  - in the beginning of the loop $i = j$
  - i does not change, j increases
  - (this can be proven by adding $i \leq j$ into the invariant)

$\Longrightarrow$ $wp(\ A[j] := A[j+1];\ j := j+1,\ \mathit{Inv}_2\ )\ \Leftrightarrow$
$n = n_0 \wedge 1 \leq j+1 \leq n \wedge \mathit{Unchd}(1,i) \wedge$
$X \wedge \mathit{Unchd}(j+2,n+1)$,
where
$X\ \Leftrightarrow\ \mathit{Movd}(i,j) \wedge A[j+1] = A_0[j+1]$

- $Inv_2 \wedge j < n$, in other words,
  $j < n \wedge n = n_0 \wedge 1 \leq j \leq n \wedge$
  $Unchd(1, i) \wedge Movd(i, j) \wedge Unchd(j+1, n+1)$
  guarantees

  - $n = n_0$ $\quad (\Leftarrow n = n_0)$
  - $1 \leq j+1 \leq n$ $\quad (\Leftarrow j < n \wedge 1 \leq j)$
  - $Unchd(1, i)$ $\quad (\Leftarrow Unchd(1, i))$
  - $Movd(i, j)$ $\quad (\Leftarrow Movd(i, j))$
  - $A[j+1] = A_0[j+1]$ $\quad (\Leftarrow Unchd(j+1, n+1) \wedge j < n)$
  - $Unchd(j+2, n+1)$ $\quad (\Leftarrow Unchd(j+1, n+1))$

$\Longrightarrow$ •/•

Proving termination

- the only loop in the program is a true **for**-loop
  $\Rightarrow$ termination is automatically guaranteed
- $n - j$ could be used as bound function

We have now proved

$\langle\ A[1 \ldots n] = A_0 \wedge 1 \leq i \leq n\ \rangle$
for $j := i$ **to** $n - 1$ **do**
  $A[j] := A[j+1]$
endfor ;
$n := n - 1$
$\langle\ n = n_0 \wedge Unchd(1, i) \wedge Movd(i, n+1)\ \rangle$

Notices

- the missing precondition $1 \leq i \leq n$ was revealed

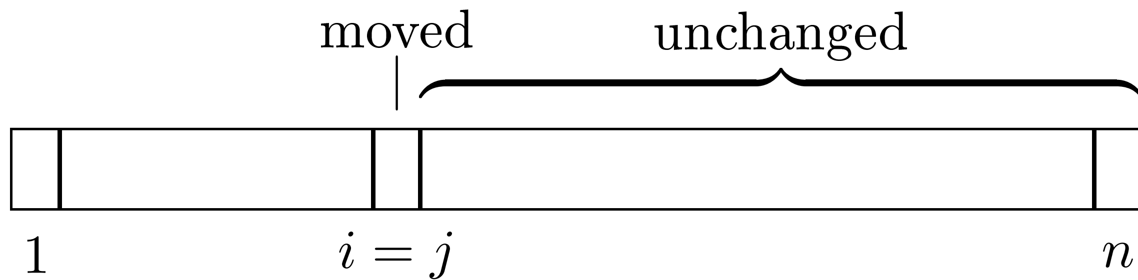
- the actual invariant was

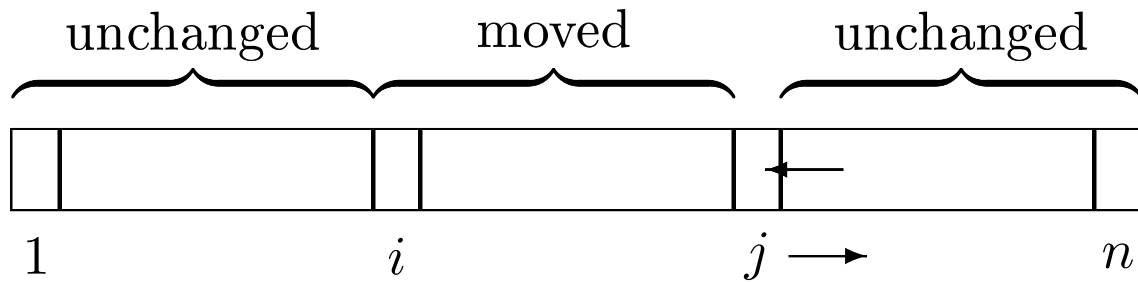
  $$n = n_0 \wedge 1 \leq i \leq j \leq n \wedge \mathit{Unchd}(1, i)$$
  $$\wedge\ \mathit{Movd}(i, j) \wedge \mathit{Unchd}(j + 1, n + 1)$$

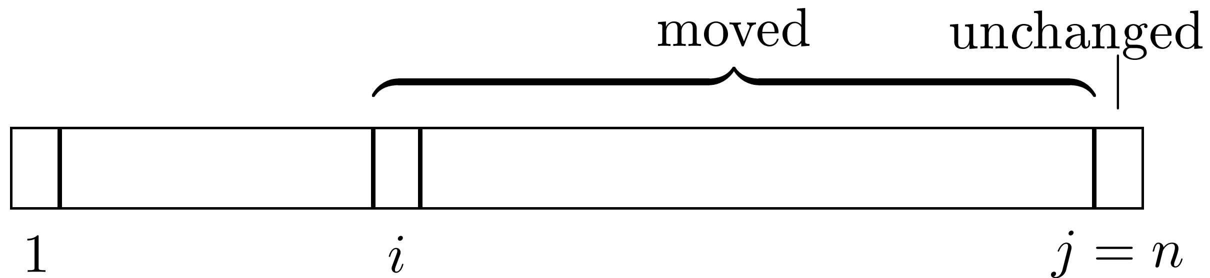
  - the part $1 \leq i \leq j$ was handled separately
  - the whole upper line could have been handled separetely

- the core of the invariant
  $\mathit{Movd}(i, j) \wedge \mathit{Unchd}(j + 1, n + 1)$
  is of the form
  $I_1 \wedge I_2$
  where

  - in the beginning of the loop I_1 speaks about an empty range, and the precondition for the loop guarantees I_2
  - during the loop the range of I_1 increases and the range of I_2 decreases
  - in the end of the loop I_2 speaks about an empty range and I_1 guarantees the state predicate after the loop







- this is quite usual
- the part $Unchd(j+1, n+1)$ could have been $Unchd(j, n+1)$
  - the calculations would still have succeeded
  - it is a matter of taste which one is chosen

Example: binary search

- we have already proved that

$$
\begin{array}{l}
\langle \text{ True } \rangle \\
a := 1;\ y := n+1; \\
\ \mathbf{while}\ a < y\ \ \mathbf{do} \\
\qquad v := (a+y)\ \mathsf{div}\ 2; \\
\qquad \mathbf{if}\ A[v] < \mathit{key}\ \ \mathbf{then}\ a := v+1 \\
\qquad \mathbf{else}\ y := v \\
\ \mathbf{endif} \\
\langle\ 1 \leq a = y \leq n+1\ \rangle
\end{array}
$$

- the whole specification for binary search is:

$$
\begin{array}{l}
\langle\ \forall i\ ;\ 1 \leq i \leq n-1 : A[i] \leq A[i-1]\ \rangle \\
\qquad\qquad \textit{binary search} \\
\langle\ (a = n+1 \vee A[a] \geq \mathit{key}) \wedge \\
\quad (a = 1 \vee A[a-1] < \mathit{key})\ \rangle
\end{array}
$$

- (why here it is not required $1 \leq a = y \leq n{+}1$?)
- we prove first that in the end

$$a = 1 \vee A[a-1] < \mathit{key}$$

- it is easy to check direct from the code that if the statement $a := v{+}1$ is never executed then in the end $a = 1$

- if the statemenet is executed then for the last execution it holds that (why?)

$$\langle\ A[v] < key\ \rangle\ a := v + 1$$

- symbolic execution gives

$$\langle\ A[v] < key\ \rangle\ a := v+1\ \langle\ A[a-1] < key\ \rangle$$

- since after this the value of a does not change, and A and key are fixed, then in the end it holds that

$$A[a-1] < key$$

⇒ it is proven that in the end $a = 1 \vee A[a-1] < key$

- since we already know that in the end $y = a$, we get

$$\begin{array}{l} \langle\ \mathsf{True}\ \rangle \\ \quad \textit{binary search} \\ \langle\ (a = n + 1 \vee A[a] \geq key) \wedge \\ \quad (a = 1 \vee A[a-1] < key)\ \rangle \end{array}$$

The proof of binary search did not use in any way the precondition $\forall i\ ;\ 1 \leq i \leq n-1 : A[i] \leq A[i-1]$!

⇒ **question:** is there an error in the proof?

- **answer:** no, the program really guarantees that in the end $a = n + 1 \vee A[a] \geq key$ etc. even the array would not be in order
- **question:** but binary search does not even work unless the array is ordered!
- **question:** let us see...
- the actual purpose of binary search is to find *key* in the array
  - if *key* is not in the array then of course binary search cannot find it

  $\Rightarrow$ the nonfunctioning of binary search can happen only so that *key* is not found but it is in the array
- we check does it hold that

  $\langle\ \exists x\ ;\ 1 \leq x \leq n : A[x] = key\ \rangle$
  binary search
  $\langle\ A[a] = key\ \rangle$

  - no: $A = [1, 0]$ and $key = 0$ produces $a = 1$
- what about if we add the assumption: A is in order?
  - so $\forall i\ ;\ 1 \leq i \leq n - 1 : A[i] \leq A[i - 1]$
  - the following form is in practice more handy:

$$\forall i, j \ ; \ 1 \leq i \leq j \leq n : A[i] \leq A[j]$$

- since A and *key* are fixed, in the end it holds that

$$\begin{aligned}
&\forall i, j \ ; \ 1 \leq i \leq j \leq n : A[i] \leq A[j] \wedge \\
&\exists x \ ; \ 1 \leq x \leq n : A[x] = key \wedge \\
&(a = n + 1 \vee A[a] \geq key) \wedge \\
&(a = 1 \vee A[a-1] < key)
\end{aligned}$$

- is it worth to try to prove that in the end $a = x$?

- since in the end $a = 1 \vee A[a-1] < key$ and A is in order, then $a \leq x$

  $\Rightarrow$ $A[a] \geq key$ (since $a = n+1$ is not possible since $x \leq n$) and $A[a] \leq A[x] = key$ (since $a \leq x$)

  $\Rightarrow$ $A[a] = key$

- so, the asumption of ordering is needed for ensuring that if *key* is in the array then at least one *key* is in the location indicated by the original postcondition

- if A is not in order, binary search indeed searches the location indicated by the original postcondition, but *key* may not be there

Conclusions

- everything was not proved using the techniques of the previous sections
  - termination and that a remains in the legal range were proved by them
  - after that that a hits “the correct slot” were deduced directly from the code
  - that key is found was reasoned separately using the postcondition and the assumption of ordering without looking the code
- everything could have been proved using the previous techniques but that would have been more difficult
  - often it is worth to prove “easy parts first” and then combine the final target little by little
- for binary search we had to form a complicated postcondition since it must “work” even when the searched key is not in the array
- surprisingly binary search guarantees the postcondition even when the array is not in order
- the assumption of ordering is needed only for that without it the postcondition does not guarantee that the key is found

  - this is not a surprise since it was in our mind when we were formulating the post-condition

- a well designed program guarantees sometimes (often?) surprising properties!

- it often holds that

  - a well designed program has logically fine properties
  - a poorly designed program is logically messy

For comparison, the proof is done by totally using invariants and bound functions

- for the original postcondition

  - maybe it becomes a little more complicated! $\ddot\smile$

    $\langle$ True $\rangle$
    $a := 1;\ y := n + 1;$
    $\langle\ a = 1 \land y = n + 1\ \rangle$
    $\langle$ **inv:** $1 \leq a \leq y \leq n + 1 \land$
    $(y = n + 1 \lor A[y] \geq key) \land$
    $(a = 1 \lor A[a - 1] < key)\ \rangle$
    $\langle$ ***bf*:** $y - a\ \rangle$

$$
\begin{array}{l}
\textbf{while}\ a < y\ \ \textbf{do} \\
\quad \langle\, 1 \le a < y \le n+1 \land \\
\quad\ \ (y = n+1 \lor A[y] \ge key) \land \\
\quad\ \ (a = 1 \lor A[a-1] < key) \land \\
\quad\ \ y - a = bf_0 > 0 \,\rangle \\
\quad v := (a+y)\ \mathsf{div}\ 2; \\
\quad \langle\, 1 \le a \le v < y \le n+1 \land \\
\quad\ \ (y = n+1 \lor A[y] \ge key) \land \\
\quad\ \ (a = 1 \lor A[a-1] < key) \land \\
\quad\ \ y - a = bf_0 \,\rangle \\
\quad \textbf{if}\ A[v] < key\ \ \textbf{then} \\
\quad\quad a := v+1 \\
\quad\quad \langle\, 1 < a \le y \le n+1 \land \\
\quad\quad\ \ (y = n+1 \lor A[y] \ge key) \land \\
\quad\quad\ \ A[a-1] < key \land \\
\quad\quad\ \ y - a < bf_0 \,\rangle \\
\quad \textbf{else} \\
\quad\quad y := v \\
\quad\quad \langle\, 1 \le a \le y \le n \land \\
\quad\quad\ \ A[y] \ge key \land \\
\quad\quad\ \ (a = 1 \lor A[a-1] < key) \land \\
\quad\quad\ \ y - a < bf_0 \,\rangle \\
\quad \textbf{endif} \\
\quad \langle\, 1 \le a \le y \le n+1 \land \\
\quad\ \ (y = n+1 \lor A[y] \ge key) \land \\
\quad\ \ (a = 1 \lor A[a-1] < key) \,\rangle \\
\textbf{endwhile}
\end{array}
$$

$$\langle\, 1 \le a = y \le n + 1 \wedge$$
$$(a = n + 1 \vee A[a] \ge \mathit{key}) \wedge$$
$$(a = 1 \vee A[a - 1] < \mathit{key})\,\rangle$$

- *key* is found if it is in the array
  - A is fixed
  - ⇒ the array is all the time in order
  - ⇒ we do not repeat the information of ordering in every predicate even we use it in reasoning

$\langle\, \exists x \;;\; 1 \le x \le n : A[x] = \mathit{key} \wedge$
$\quad \forall i, j \;;\; 1 \le i \le j \le n : A[i] \le A[j]\,\rangle$
$a := 1;\; y := n + 1;$
$\langle\, a = 1 \wedge y = n + 1 \wedge$
$\quad \exists x \;;\; 1 \le x \le n : A[x] = \mathit{key}\,\rangle$
$\langle\, \textbf{inv:}\ \exists x : 1 \le a \le x \le y \le n + 1 \wedge$
$\quad x \le n \wedge A[x] = \mathit{key}\,\rangle$
$\langle\, \textbf{\textit{bf}:}\ y - a\,\rangle$
while $a < y$ **do**
$\quad \langle\, \exists x : 1 \le a \le x \le y \le n + 1 \wedge x \le n \wedge$
$\quad\quad A[x] = \mathit{key} \wedge a < y \wedge y - a = \mathit{bf}_0 > 0\,\rangle$
$\quad v := (a + y)\ \mathsf{div}\ 2;$ (* now $a \le v < y$ *)
$\quad$ **if** $A[v] < \mathit{key}$ **then**
$\quad\quad \langle\, \exists x : 1 \le v < x \le y \le n + 1 \wedge x \le n \wedge$
$\quad\quad\quad A[x] = \mathit{key} \wedge y - v \le \mathit{bf}_0\,\rangle$
$\quad\quad a := v + 1$
$\quad$ **else**
$\quad\quad \langle\, (A[v] = \mathit{key} \wedge 1 \le a \le v \le n \vee$

$A[v] > key \land \exists x : 1 \leq a \leq x \leq v \leq n \land$
$A[x] = key) \land v - a < bf_0 \rangle$
$\langle \exists x : 1 \leq a \leq x \leq y \leq n \land x \leq n \land$
$A[x] = key \land v - a < bf_0 \rangle$
$y := v$
endif
endwhile
$\langle \exists x : 1 \leq a \leq x \leq y \leq n + 1 \land x \leq n \land$
$A[x] = key \land a \geq y \rangle$
$\langle 1 \leq a \leq n \land A[a] = key \rangle$

- the proof was surprisingly laborious!
- first, because of the initialisation $y := n + 1$ we had to often write separately $x \leq n$
  - this proof had been easier if it had been initialised $y := n$
- second, if $A[v] = key$, then
  - using the ordering assumption it is not possible to reason that $x \leq v$
  - then v can be chosen as the new x

  ⇒ the reasoning for **else**-branch had to be divided into two parts
- by proving first the original postcondition and then with its help the key being found, we avoided both these difficulties

   - we did not need to talk about the location of key while proving the program code

⇒ even the key being found was easier to prove using the original postcondition than directly

⇒ the original postcondition summarises excellently "the core" of binary search!

Another example: COUNTING-SORT: the task

- sorts an array by counting how many time each key exists
- the keys must be in range $0, \ldots, M$
- input: $A[1 \ldots n]$ and M are fixed
- output: $B[1 \ldots n]$
- essential additional property: *stable*
   - in other words, the mutual ordering of two records is not changed if they have same key
- stableness has meaning only if an element of an array contains more than just key

⇒ for an element we assume the following structure:

   - $A[i].key$ key
   - $A[i]$ the whole element

- the algorithm

```
Counting-Sort(A[1...n], B[1...n], M)

{ ∀i ; 1 ≤ i ≤ n : 0 ≤ A[i].key ≤ M }
 for k := 0 to M do C[k] := 0 endfor
 for i := 1 to n do
     C[A[i].key] := C[A[i].key] + 1
 endfor
(*  now C[k] knows how many element
    has key = k *)

 for k := 1 to M do
     C[k] := C[k] + C[k − 1]
 endfor
(*  now C[k] knows how many element
    has key ≤ k *)

 for i := n downto 1 do
     B[C[A[i].key]] := A[i];
     C[A[i].key] := C[A[i].key] − 1
 endfor
```

- domains of variables
  - k: at least $0, \ldots, M$
  - i: at least $1, \ldots, n$
  - $C[0 \ldots M]$: at least $0, \ldots, n$
  - $B[1 \ldots n]$: at least the same as $A[1 \ldots n]$

The specification of COUNTING-SORT

- the old and already familiar

  $$ordered(B) \land same\text{-}elems(A, B)$$

  does not contain the requirement of stableness

- to present the requirement of stableness we introduce a function f which tells into which location in array B the elements of array A are moved

  – thus in the end there must be

    $\forall i \; ; \; 1 \leq i \leq n :$
    $1 \leq f(i) \leq n \land B[f(i)] = A[i]$

- f must be a bijection:

  $$\forall i, j \; ; \; 1 \leq i < j \leq n : f(i) \neq f(j)$$

- requirement of stableness:

  $$\forall i, j \; ; \; 1 \leq i < j \leq n :$$
  $$A[i].key = A[j].key \rightarrow f(i) < f(j)$$

Beginning of the proof of Counting-Sort

- A is fixed

  $\Rightarrow$ condition
  $\forall i \; ; \; 1 \leq i \leq n : 0 \leq A[i].key \leq M$
  holds all the time

$\Rightarrow$ we do not repeat it in every predicate

- the task of the 1. **for**-loop is to zero C

$$\langle\ \forall i\ ;\ 1 \leq i \leq n : 0 \leq A[i].key \leq M\ \rangle$$
$$\textbf{for}\ k := 0\ \ \textbf{to}\ M\ \ \textbf{do}\ C[k] := 0\ \ \textbf{endfor}$$
$$\langle\ \forall h\ ;\ 0 \leq h \leq M : C[h] = 0\ \rangle$$

  - legality of assignments is obvious
  - invariant $I_1 \Leftrightarrow$
    $\forall h\ ;\ 0 \leq h \leq k - 1 : C[h] = 0$
  - $k = 0 \Rightarrow I_1$ obvious $^\bullet/_\bullet$
  - $\langle\ I_1 \wedge k \leq M\ \rangle\ C[k] := 0;\ k := k+1\ \langle\ I_1\ \rangle$,
    since
    $\forall h\ ;\ 0 \leq h \leq k : C([k] \leftarrow 0)[h] = 0 \Leftrightarrow$
    $I_1 \wedge\ 0 = 0$
  - $k = M + 1 \wedge I_1 \Rightarrow$
    $\forall h\ ;\ 0 \leq h \leq M : C[h] = 0$ $^\bullet/_\bullet$

- the task of the 2. **for**-loop is to count how many times different elements exist in A
  - invariant

    $\forall h \; ; \; 0 \leq h \leq M : C[h] = |\{ j \mid 1 \leq j \leq i-1 \wedge A[j].key = h \}|$
  - the proof is left for exercise

$$\langle \forall h \; ; \; 0 \leq h \leq m : C[h] = 0 \rangle$$

for $i := 1$ **to** n **do**
    $C[A[i].key] := C[A[i].key] + 1$
endfor

$$\langle \forall h \; ; \; 0 \leq h \leq M : C[h] = |\{ j \mid 1 \leq j \leq n \wedge A[j].key = h \}| \rangle$$

- the 3. **for**-loop counts how many elements of A is $\leq h$
  - the proof is left for exercise

$$\langle \forall h \; ; \; 0 \leq h \leq M : C[h] = |\{ j \mid 1 \leq j \leq n \wedge A[j].key = h \}| \rangle$$

for $k := 1$ **to** M **do**
    $C[k] := C[k] + C[k-1]$
endfor

$$\langle \forall h \; ; \; 0 \leq h \leq M : C[h] = |\{ j \mid 1 \leq j \leq n \wedge A[j].key \leq h \}| \rangle$$

- the proof of the 4. **for**-loop
  - so far nothing else has happened but information of the amount of keys has been counted into C
  - the task of 4. **for**-loop is to copy the elements of A to B using this information
  - according to the specification the dependency f between the old and the new elements is essential

  $\Rightarrow$ it is worth to first recognise f, and then prove that the program calculates correctly

$$\langle\, \forall h \;;\; 0 \leq h \leq M : C[h] = \\ |\left\{ j \;\middle|\; 1 \leq j \leq n \wedge A[j].key \leq h \right\}|\,\rangle$$

for $i := n$ **downto** 1 **do**
    $B[C[A[i].key] := A[i];$
    $C[A[i].key] := C[A[i].key] - 1$
endfor

$$\langle\, \forall j \;;\; 1 \leq j \leq n : B[f(j)] = A[j] \,\rangle$$

where

$$f(x) = |\left\{ k \;\middle|\; 1 \leq k \leq n \wedge A[k].key < A[x].key \right\}| \\ + |\left\{ k \;\middle|\; 1 \leq k \leq x \wedge A[k].key = A[x].key \right\}|$$

- invariant $I_4 \;\Leftrightarrow$
  $\forall h \;;\; : 0 \leq h \leq M :$
  $C[h] = \left| \left\{ k \;\middle|\; 1 \leq k \leq n \wedge A[k].key < h \right\} \right|$
  $\quad + \left| \left\{ k \;\middle|\; 1 \leq k \leq i \wedge A[k].key = h \right\} \right| \wedge$
  $\forall j \;;\; i+1 \leq j \leq n : B[(f(j)] = A[j]$

  - $P \;\Rightarrow\; I_4$ obvious •/•
  - preservation of I_4 is going to be proved soon
  - $I_4 \wedge i = 0 \;\Rightarrow\; Q$ obvious •/•

- for future use notice $C[A[i].key] = f(i)$

- the C-part of I_4 is preserved, since

$$\begin{aligned}
&\forall h \;;\; 0 \leq h \leq M : \\
&\quad C([A[i].key] \leftarrow C[A[i].key] - 1)[h] \;= \\
&\qquad \left| \left\{ k \;\middle|\; 1 \leq k \leq n \wedge A[k].key < h \right\} \right| + \\
&\qquad \left| \left\{ k \;\middle|\; 1 \leq k \leq i-1 \wedge A[k].key = h \right\} \right| \\
&\Leftrightarrow \\
&\forall h \;;\; 0 \leq h \leq M : \\
&\quad (\, h \neq A[i].key \rightarrow C[h] \;= \\
&\qquad \left| \left\{ k \;\middle|\; 1 \leq k \leq n \wedge A[k].key < h \right\} \right| + \\
&\qquad \left| \left\{ k \;\middle|\; 1 \leq k \leq i \wedge A[k].key = h \right\} \right| \\
&\quad \wedge (h = A[i].key \rightarrow C[A[i].key] - 1 \;= \\
&\qquad C[h] - 1 \;= \\
&\qquad \left| \left\{ k \;\middle|\; 1 \leq k \leq n \wedge A[k].key < h \right\} \right| + \\
&\qquad \left| \left\{ k \;\middle|\; 1 \leq k \leq i \wedge A[k].key = h \right\} \right| - 1) \\
&\Leftrightarrow\; C\text{-part}
\end{aligned}$$

- for proving the B-part of I_4 it is necessary to prove that if $i \neq j$ then $f(i) \neq f(j)$
  - if $A[i].key = A[j].key$ and $i < j$ then $f(j) = f(i) + |\{k \mid i+1 \leq k \leq j \wedge A[k].key = A[i].key\}| \geq f(i) + 1 > f(i)$
  - if $A[i].key < A[j].key$ then $f(j) > |\{k \mid 1 \leq k \leq n \wedge A[k].key < A[j].key\}| \geq |\{k \mid 1 \leq k \leq n \wedge A[k].key \leq A[i].key\}| \geq f(i)$

  $\Rightarrow$ if $A[i].key < A[j].key$ or $A[i].key = A[j].key \wedge i < j$ then $f(i) < f(j)$

  $\Rightarrow$ $i \neq j$ then $f(i) \neq f(j)$

- now $wp(\ S, \forall j\ ;\ i+1\ j \leq n : B[f(j)] = A[j]\ )$
  $\Leftrightarrow \forall j\ ;\ i \leq j \leq n :$
  $B([C[A[i].key]\ ] \leftarrow A[i])[f(j)] = A[j]$
  $\Leftrightarrow \forall j\ ;\ i \leq j \leq n :$
  $(\ f(j) = C[A[i].key] \rightarrow A[i] = A[j]\ ) \wedge$
  $(\ f(j) \neq C[A[i].key] \rightarrow B[f(j)] = A[j]\ )$
  (since $C[A[i].key] = f(i)$, and f is a bijection)
  $\Leftrightarrow \forall j\ ;\ i+1 \leq j \leq n : B[f(j)] = A[j]$,
  thus the B-part is preserved

- for proving the legality of assignments we have to prove that $\forall i \; ; \; 1 \leq i \leq n : 1 \leq f(i) \leq n$
  - $1 \leq$
    $|\, \{\, k \mid 1 \leq k \leq i \wedge A[k].key = A[i].key \,\} \,|$
    $\leq \; f(i) \; \leq$
    $|\, \{\, k \mid 1 \leq k \leq n \wedge A[k].key \leq A[i].key \,\} \,|$
    $\leq \; n$ •/•

The last phases of the proof of Counting-Sort

- the requirement of stableness has already been proved (when?)

  $\forall i, j \; ; \; 1 \leq i \leq n \wedge 1 \leq j \leq n :$
  $\quad (\, A[i].key = A[j].key \wedge i < j \rightarrow f(i) < f(j) \,)$

- to prove *ordered*(B) and *same-elems*(A, B) it is important to notice that
  - into array B there have been written n times
  - every time it was written into a different location (in other words, $i \neq j \rightarrow f(i) \neq f(j)$)

  $\Rightarrow$ it has been written into every element of B

  $\Rightarrow$ $\forall i \; ; \; 1 \leq i \leq n : \exists k \; ; \; 1 \leq k \leq n : i = f(k)$

- let $1 \leq i < j \leq n$, and let k and l be chosen so that $i = f(k)$ and $j = f(l)$

- if $A[l].key < A[k].key$ then
  $j = f(l) < f(k) < i$
  $\Rightarrow$ $B[i].key = B[f(k)].key = A[k].key \leq A[l].key = B[f(l)].key = B[j].key$

$\Rightarrow \forall i, j \; ; \; 1 \leq i \leq j \leq n : B[i].key \leq B[j].key$

- in other words, $ordered(B)$

- if $1 \leq i \leq n$ then $amount(A[i], B)$
  $= | \{ k \mid 1 \leq k \leq n \wedge B[k] = A[i] \} |$
  $= | \{ k \mid 1 \leq k \leq n \wedge B[f(k)] = A[i] \} |$ (why?)
  $= | \{ k \mid 1 \leq k \leq n \wedge A[k] = A[i] \} |$
  $= amount(A[i], A)$
  $\Rightarrow$ $samelems(A, B)$ $^\bullet/_\bullet$

Third example: the test program for Fermat's great theorem (or Fermat's last theorem): stating the problem

- the following question is famous
  - it was open from 17th century until it was solved at 1995

    Is there positive integers x, y, z, and n such that $n \geq 3$ and $x^n + y^n = z^n$?
  - About this, Pierre de Fermat wrote in 1637 in his copy of Diophantus's Arithmetica,
    "I have discovered a truly remarkable proof but this margin is too small to contain it."

Although the theorem was subsequently shown to be true for many specific values of n, leading to important mathematical advances in the process, the difficulty of the problem soon convinced mathematicians that Fermat never had a valid proof.

In 1995 the British mathematician Andrew Wiles (b. 1953) and his former student Richard Taylor (b. 1962) published a complete proof, finally solving one of the most famous of all mathematical problems.

- the task of the following program is to test all the possible x, y, z, and n until the required values are found

$x := 1;\ y := 1;\ z := 1;\ n := 3;$
while $x^n + y^n \neq z^n$ **do**
  if $y > 1$ **then** $x := x + 1;\ y := y - 1$
  elseif $z > 1$ **then** $z := z - 1;\ y := x + 1;\ x := 1$
  elseif $n > 3$ **then** $n := n - 1;\ z := x + 1;\ x := 1$
  else $n := x + 3;\ x := 1$
  endif
endwhile
$\langle\, x \geq 1 \wedge y \geq 1 \wedge z \geq 1 \wedge n \geq 3 \wedge x^n + y^n = z^n \,\rangle$

⇒ if the program works properly, it is not worth for **us** to even try to work out does it terminate

  - (from the result of Andrew Wiles's work it follows: it does not terminate)

- but, instead of that, it is possible and necessary to check
  - does the program test only legal x, y, z, and n? (otherwise it could terminate too early, e.g. $x = 3$, $y = 4$, $z = 5$, and $n = 2$)
  - if there exist no desired x, y, z, and n, does the program test all the legal possibilities?

$\Rightarrow$ the last example of this item

Does the program test only legal x, y, z, and n?

- let us denote ("L" $\sim$ legal)

$$L \Leftrightarrow x \geq 1 \wedge y \geq 1 \wedge z \geq 1 \wedge n \geq 3$$

- we have to prove that L holds in the beginning of every cycle of the loop
- we need an invariant I such that

$$I \Rightarrow L$$

- the next are easy to check:
  - the initialisation causes that I holds

- $\langle\ L \land y > 1\ \rangle$
  $x := x + 1;\ y := y - 1$
  $\langle\ L\ \rangle$
- $\langle\ L \land z > 1 \land \ldots\ \rangle$
  $z := z - 1;\ y := x + 1;\ x = 1$
  $\langle\ L\ \rangle$
- $\langle\ L \land n > 3 \land \ldots\ \rangle$
  $n := n - 1;\ z := x + 1;\ x = 1$
  $\langle\ L\ \rangle$
- $\langle\ L \land \ldots\ \rangle$
  $n := x + 3;\ x := 1$
  $\langle\ L\ \rangle$

therefore we can choose $I \Leftrightarrow L$

$\Rightarrow$ the program tests only legal x, y, z, and n

If there exist no desired x, y, z, and n, does the program test all the legal possibilities?

- this is a problem of totally different type than we have solved this far
- nevertheless, we do have sufficent means!
  - let us denote the target state by Q_0
  - let us build a sequence of legal states Q_1, Q_2, ... such that Q_j guarantees that after next cycle Q_{j-1} holds (yes, "$j - 1$"!)

  - using the bound function technique we prove that the sequence ends to a state where $x = y = z = 1 \wedge n = 3$

$\Rightarrow$ from the initial state we get to the target state by a finite amount of loop cycles unless, before it, in some state it holds that $x^n + y^n = z^n$

- let us denote
  - S is the body of the **while**-loop
  - $Q \Leftrightarrow x = x_0 \wedge y = y_0 \wedge z = z_0 \wedge n = n_0$
- for the target state Q_0 only legal states are accepted

  $\Longrightarrow$ we define $Q_0 \Leftrightarrow L \wedge Q$
- $\langle\, Q_j \,\rangle\ S\ \langle\, Q_{j-1} \,\rangle$ holds if and only if $Q_j \Rightarrow wp(\, S,\, Q_{j-1}\,)$

  $\Longrightarrow$ we choose $Q_j \Leftrightarrow L \wedge wp(\, S,\, Q_{j-1}\,)$
- let us denote x_j = the value of x in state j, and correspondingly for y_j, z_j, and n_j

- the body of the loop, S, is of the form *COMP-IF* (or comp-structure):

  $$\textbf{if } B_1 \textbf{ then } S_1 \textbf{ elseif } B_2 \textbf{ then } S_2 \textbf{ elseif } \cdots \textbf{ elseif } B_{n-1} \textbf{ then } S_{n-1} \textbf{ else } S_n \textbf{ endif}$$

  - we can get the same functionality by nesting ordinary **if**—**then**—**else**—**endif**-statements
  - therefore the weakest precondition for Q with respect to the *COMB-IF* is

    $wp(\ COMB\text{-}IF, Q\ ) \Leftrightarrow$
    $\Uparrow B_1 \ \wedge B_1 \wedge wp(\ S_1, Q\ ) \vee$
    $\Uparrow B_1 \ \wedge \Uparrow B_2 \ \wedge \neg B_1 \wedge B_2 \ \wedge \ wp(\ S_2, Q\ ) \vee$
    $\cdots$
    $\Uparrow B_1 \ \wedge \cdots \wedge \Uparrow B_{n-1} \ \wedge \neg B_1 \wedge \cdots \wedge \neg B_{n-2} \wedge B_{n-1} \ \wedge \ wp(\ S_{n-1},\ Q\ ) \vee$
    $\Uparrow B_1 \ \wedge \cdots \wedge \Uparrow B_{n-1} \ \wedge \neg B_1 \wedge \cdots \wedge \neg B_{n-1} \wedge wp(\ S_n,\ Q\ )$

- $Q_j \;\Leftrightarrow\; L \wedge wp(\, S,\, Q_{j-1}\,) \;\;\Leftrightarrow$

$$
\begin{array}{ll}
\multicolumn{2}{l}{x_j \geq 1 \wedge y_j \geq 1 \wedge z_j \geq 1 \wedge n \geq 3 \wedge} \\
(& y_j > 1 \wedge \\
 & x_j + 1 = x_{j-1} \wedge y_j - 1 = y_{j-1} \wedge \\
 & z_j = z_{j-1} \wedge n_j = n_{j-1} \\
\vee & y_j \leq 1 \wedge z_j > 1 \wedge \\
 & 1 = x_{j-1} \wedge x_j + 1 = y_{j-1} \wedge \\
 & z_j - 1 = z_{j-1} \wedge n_j = n_j - 1 \\
\vee & y_j \leq 1 \wedge z_j \leq 1 \wedge n_j > 3 \wedge \\
 & 1 = x_{j-1} \wedge y_j = y_{j-1} \wedge \\
 & x_j + 1 = z_{j-1} \wedge n_j - 1 = n_{j-1} \\
\vee & y_j \leq 1 \wedge z_j \leq 1 \wedge n_j \leq 3 \wedge \\
 & 1 = x_{j-1} \wedge y_j = y_{j-1} \wedge z_j = z_{j-1} \wedge \\
 & x_j + 3 = n_{j-1} \\
) & \\
\Leftrightarrow & \\
 & x_{j-1} > 1 \wedge y_{j-1} \geq 1 \wedge z_{j-1} \geq 1 \wedge n_{j-1} \geq 3 \wedge \\
 & x_j = x_{j-1} - 1 \wedge y_j = y_{j-1} + 1 \wedge \\
 & z_j = z_{j-1} \wedge n_j = n_{j-1} \\
\vee & x_{j-1} = 1 \wedge y_{j-1} > 1 \wedge z_{j-1} \geq 1 \wedge n_{j-1} \geq 3 \wedge \\
 & x_j = y_{j-1} - 1 \wedge y_j = 1 \wedge \\
 & z_j = z_{j-1} + 1 \wedge n_j = n_{j-1} \\
\vee & x_{j-1} = 1 \wedge y_{j-1} = 1 \wedge z_{j-1} > 1 \wedge n_{j-1} \geq 3 \wedge \\
 & x_j = z_{j-1} \wedge y_j = 1 \wedge \\
 & z_j = 1 \wedge n_j = n_{j-1} + 1 \\
\vee & x_{j-1} = 1 \wedge y_{j-1} = 1 \wedge \\
 & z_{j-1} = 1 \wedge n_{j-1} > 3 \wedge \\
 & x_j = n_{j-1} - 3 \wedge y_j = 1 \wedge z_j = 1 \wedge n_j = 3
\end{array}
$$

- notices
  - if $(x_{j-1}, y_{j-1}, z_{j-1}, n_{j-1}) = (1, 1, 1, 3)$ then $Q_j \Leftrightarrow \mathsf{False}$
  - for other legal states $(x_{j-1}, y_{j-1}, z_{j-1}, n_{j-1})$ the formula defines at least one possible state (x_j, y_j, z_j, n_j)
  - (x_j, y_j, z_j, n_j) is even unique but that does not have any meaning in the sequel

⇒ the sequence Q_j can end only when $x = y = z = 1 \land n = 3$

- let us denote
  - $bf_4(j) = x_j + y_j + z_j + n_j - 6$
  - $bf_3(j) = x_j + y_j + z_j - 3$
  - $bf_2(j) = x_j + y_j - 2$
  - $bf_1(j) = x_j - 1$
- in legal states (therefore also in every Q_j)

$$bf_4(j) \geq 0 \land bf_3(j) \geq 0 \land bf_2(j) \geq 0 \land bf_1(j) \geq 0$$

holds

$$\begin{aligned} & bf_4(j) < bf_4(j-1) \\ \lor\ & bf_4(j) = bf_4(j-1) \land bf_3(j) < bf_3(j-1) \\ \lor\ & bf_4(j) = bf_4(j-1) \land bf_3(j) = bf_3(j-1) \land \\ & bf_2(j) < bf_2(j-1) \\ \lor\ & bf_4(j) = bf_4(j-1) \land bf_3(j) = bf_3(j-1) \land \\ & bf_2(j) = bf_2(j-1) \land bf_1(j) < bf_1(j-1) \end{aligned}$$

$\Rightarrow$ the vector ($bf_4(j), bf_3(j), bf_2(j), bf_1(j)$) is a bound function for the sequence Q_j

- the sequence Q_j ends in some state Q_m
- it is already proven that the sequence Q_j can end only in state
  $x = y = z = 1 \wedge n = 3$

$\Rightarrow$ the sequence Q_j ends in state
$x = y = z = 1 \wedge n = 3$

$\Rightarrow$ for every legal state Q_0 there exist a finite sequence of states $Q_0, Q_1, Q_2, \ldots, Q_m$ such that

- $\langle$ True $\rangle$
  $x := 1;\ y := 1;\ z := 1;\ n := 3$
  $\langle\ Q_m\ \rangle$
- $\langle\ Q_m\ \rangle\ S\ \langle\ Q_{m-1}\ \rangle\ S \ldots S\ \langle\ Q_1\ \rangle\ S\ \langle\ Q_0\ \rangle$

$\Longrightarrow$ if $L \Rightarrow x^n + y^n \neq z^n$ then eventually the program reaches state Q_0

4 Applications for Proving

In this section it is discussed what other benefits can be get than just proving final code

- usually final code is not even correct, thus proving cannot succeed!

Contents

- trying to prove as a means of review
- using proving as a program design method

4.1 Proving Correctness as a Means for Review

Proving an already final program does not usually succeed

- loop invariants, bound functions, suitable strengthenings and weakenings of predicates, etc. can be difficult to invent afterwards
- the program is hardly not even correct

$\Rightarrow$

- – it is worth to make the proof of a program together with the program (the next subsection)
- – a trial to prove a final program is a good means for review (this subsection)

Example: what happens if we try to prove the correctness of the erronous string equality comparator in Introduction?

```
1  ⟨ True ⟩
2  issame := (string1.length = string2.length);
3  if issame then
4      for i := 1 to string1.length do
5          issame :=
6              string1.char[i] = string2.char[i];
7  ⟨ issame ↔ string1 = string2 ⟩
```

- we introduce the following abbreviations
  - $s_1 = \textit{string1}$, $s_2 = \textit{string2}$
  - $.l = .\textit{length}$, $.c = .\textit{char}$
  - $\textit{is} = \textit{issame}$
  - $P_7 \Leftrightarrow (\textit{issame} \leftrightarrow \textit{string1} = \textit{string2})$ $\Leftrightarrow (\textit{is} \leftrightarrow s_1 = s_2)$

- **if**-rule

  $\Rightarrow$ in the beginning of line 3 P_3 must hold such that

  (a) $P_3 \wedge \textit{is} \Rightarrow \textit{wp}(\text{ lines } 4 \ldots 6, P_7\ )$

  (b) $P_3 \wedge \neg\textit{is} \Rightarrow P_7$

- let us outfold "$s_1 = s_2$": $s_1 = s_2 \Leftrightarrow$ $s_1.l = s_2.l \wedge \forall k\ ;\ 1 \leq k \leq s_1.l : s_1.c[k] = s_2.c[k]$

$\Longrightarrow P_7 \Leftrightarrow (\textit{is} \leftrightarrow s_1.l = s_2.l \wedge \forall k\ ;\ 1 \leq k \leq s_1.l : s_1.c[k] = s_2.c[k])$

- using symbolic execution we get the strongest P_3 such that $\langle$ True $\rangle$ line 2 $\langle\ P_3\ \rangle$:

$$P_3 \Leftrightarrow (\textit{is} \leftrightarrow s_1.l = s_2.l)$$

- now $P_3 \wedge \textit{is} \Rightarrow \neg\textit{is} \wedge s_1.l \neq s_2.l \Rightarrow \neg\textit{is} \wedge s_1 \neq s_2$

  $\Rightarrow$ P_7, so (b) holds

  - we have checked the case of strings of different lengths

- to prove (a) we need an invariant I for the **for**-loop such that

  (c) $P_3 \wedge is \wedge i = 1 \;\Rightarrow\; I$

  (d) $\langle\ I \wedge 1 \leq i \leq s_1.l\ \rangle$
  lines 5...6; $i := i + 1$
  $\langle\ I\ \rangle$

  (e1) $I \wedge i = s_1.l + 1 \;\Rightarrow\; P_7$

  (e2) $P_3 \wedge is \wedge s_1.l < 1 \;\Rightarrow\; P_7$

- I should probably be about of the form
  $I_1 \;\Leftrightarrow\; (is \leftrightarrow \forall k\ ;\ 1 \leq k \leq i - 1 : s_1.c[k] = s_2.c[k])$

  - (c) holds
  - (e) does not hold: P_7 tests also $s_1.l = s_2.l$

- a fix

  - $P_3 \wedge is \;\Rightarrow\; s_1.l = s_2.l$,
    and $s_1.l$ and $s_2.l$ are fixed

  $\Rightarrow$ $s_1.l = s_2.l$ can be added to the invariant without breaking (c) and (d)

$$I_2 \;\Leftrightarrow\; s_1.l = s_2.l \wedge I_1$$

- now
  - $P_3 \wedge is \wedge i = 1 \;\Rightarrow$
    $is \wedge s_1.l = s_2.l \wedge i = 1 \;\Rightarrow$
    I_2, thus (c) $^\bullet/_\bullet$
  - $I_2 \wedge i = s_1.l + 1$

    $\Rightarrow\; s_1.l = s_2.l \wedge (is \leftrightarrow$
    $\quad \forall k \;;\; 1 \leq k \leq s_1.l : s_1.c[k] = s_2.c[k])$

    (auxiliary rule: $P \wedge (Q \leftrightarrow R) \;\Rightarrow\; Q \leftrightarrow P \wedge R$, proof:

    * let $P \leftrightarrow$ True: LS $\Leftrightarrow\; Q \leftrightarrow R \;\Leftrightarrow$ RS
    * let $P \leftrightarrow$ False: LS $\Leftrightarrow$ False $\Rightarrow$ RS)

    $\Rightarrow\; is \leftrightarrow s_1.l = s_2.l \wedge$
    $\quad \forall k \;;\; 1 \leq k \leq s_1.l : s_1.c[k] = s_2.c[k]$
    (why this holds when $s_1.l \neq s_2.l$?)

    $\Leftrightarrow\; P_7$, thus (e) $^\bullet/_\bullet$
  - also (d): $wp($ lines 5...6; i:=i+1, $I_2\,)$
    $\Leftrightarrow\; I_2[i \leftarrow i+1][is \leftarrow (s_1.c[i] = s_2.c[i])]$
    $\Leftrightarrow\; (\; s_1.l = s_2.l \wedge (is \leftrightarrow$
    $\quad \forall k \;;\; 1 \leq k \leq i : s_1.c[i] = s_2.c[i])\;)$
    $\quad [is \leftarrow (s_1.c[i] = s_2.c[i])]$
    $\Leftrightarrow\; s_1.l = s_2.l \wedge (s_1.c[i] = s_2.c[i] \;\leftrightarrow$
    $\quad \forall k \;;\; 1 \leq k \leq i : s_1.c[k] = s_2.c[k])$
    $\Leftrightarrow\; s_1.l = s_2.l \wedge (s_1.c[i] \neq s_2.c[i] \vee$
    $\quad \forall k \;;\; 1 \leq k \leq i-1 : s_1.c[k] = s_2.c[k])$
    $\Leftrightarrow: X$

- now we have to prove that $I_2 \wedge 1 \leq i \leq s_1.l \Rightarrow X$
  - if *is* holds then $s_1.l = s_2.l \wedge I_1$
    $\Rightarrow s_1.l = s_2.l \wedge$
    $\forall k \; ; \; 1 \leq k \leq i-1 : s_1.c[k] = s_2.c[k]$
    $\Rightarrow X$
  - if *is* does not hold then $s_1.l = s_2.l \wedge I_1$
    $\Leftrightarrow s_1.l = s_2.l \wedge$
    $\neg\forall k \; ; \; 1 \leq k \leq i-1 : s_1.c[k] = s_2.c[k]$,
    so the only way to ensure X is to prove $s_1.c[i] \neq s_2.c[i]$
  - auxiliary result:
    $\neg\forall k \; ; \; 1 \leq k \leq i-1 : s_1.c[k] = s_2.c[k] \Leftrightarrow$
    $\exists k \; ; \; 1 \leq k \leq i-1 : s_1.c[k] \neq s_2.c[i]$
- problem: we have proved that $s_1.c[k] \neq s2.c[k]$ for some k, $1 \leq k \leq i-1$, but we should show that it holds when $k = i$

$\Rightarrow$ is the invariant too weak or is there an error in the program?

- test: what happens if $s_1.c[k] \neq s_2.c[k]$ for some k, $1 \leq k \leq i-1$, but $s_1.c[i] = s_2.c[i]$?

$\Rightarrow$ let us run the program with input `'aa' 'ba'`

  - the answer: "`true`" wrong!
  - actually the program calculates
    $is := (s_1.l = s_2.l) \wedge (s_1.c[s_1.l] = s_2.c[s_2.l])$

Notice

- the trial for proof encountered unexpected difficulties in certain situations

  $\Rightarrow$ is the error in the program or in the proof?

- the issue was tested with an input that leads to that situation

  $\Rightarrow$ an error was found in the program

$\Rightarrow$ the trial for proof helped us to find the input that revealed the error

Checklist for reviewing a part of program by proving

1. Does the part of program return the correct result, if it reaches the end?

   - the program is decorated by predicates
   - the predicates are tried to prove to be correct everytime when (and if) the execution of the program is in the location in question
   - if necessary, the predicates are clarified, until it can be proved to hold, or there reveals to be an error in the program
   - for simplicity we so far forget the possibility that the program executes any illegal operations

2. Does the part of program do anything that is denied?

    - are all statements computable?
    - do the indexes of arrays stay inside the legal range?
    - ...
    - check using the predicates derived from item 1.
    - if necessary clarify the predicates and return to item 1.

3. Does the part of program definitely terminate?

    - check that there are no assignments to the loop variable of **for**-loops
    - can you find a bound function for every **while**-, **repeat**-, etc. loop? Check that it is a bound function!
    - does every recursion have a bottom? Proof e.g. by giving a bound function for levels of recursion

- in practise most of these are easy to check
    - e.g. in Pascal the language itself prevents assignment to a loop variable of a **for**-loop

  - e.g. in the following the indexing of array $A[1 \ldots n]$ is always legal because of the beginning of the **for**-loop

    for $i := 1$ **to** n **do**
    $A[i] := 0$
    endfor

- it is not worth to always check the items 1–3 in this order, but you can do the easy ones first
  - otherwise the easy ones can remain totally unchecked, and there usually the despised errors are!
  - for choosing predicates it is good to know what kind of predicates the items 2 and 3 need
  - it is easier to check the difficult things if there exists a set of checked common knowledege for use

Example: check the indexing of array $A[1 \ldots n]$

$i := 1$
while $A[i] \neq x \wedge i < n$ **do**
$i := i + 1$
endwhile

- prove: every time when it is tested are we going to next cycle, it must hold that $1 \leq i \leq n$

⇒ let us take $1 \leq i \leq n$ as a part of the loop invariant

- let us try to check that $1 \leq i \leq n$ is an invariant:
  - $\langle \ \mathsf{True} \ \rangle \ i := 1 \ \langle \ i = 1 \ \rangle$
  - $i = 1 \ \Rightarrow \ 1 \leq i \leq n$ **only if** $n \geq 1$!!
- precondition $n \geq 1$ is needed
- proving is an efficient means to detect especially these kind of hidden preconditions!
  (if she who makes the calculations is careful ...)
- new try...
  - $\langle \ n \geq 1 \ \rangle \ i := 1 \ \langle \ i = 1 \land n \geq 1 \ \rangle$
  - $i = 1 \land n \geq 1 \ \Rightarrow \ 1 \leq i \leq n$
  - $wp(\ i := i + 1, \ 1 \leq i \leq n \ ) \ \Leftrightarrow$
    $1 \leq i + 1 \leq n \ \Leftrightarrow \ 0 \leq i \leq n - 1 \ \Leftarrow$
    $1 \leq i \leq n \land i < n$, thus

    $$\begin{array}{l} \langle \ 1 \leq i \leq n \land i < n \land A[i] \neq x \ \rangle \\ i := i + 1 \\ \langle \ 1 \leq i \leq n \ \rangle \end{array}$$

A different indexing of an array:

$i := 1$
while $i \leq n$ **andthen** $A[i] \neq x$ **do** $i := i + 1$
endwhile

- **andthen** computes the latter part only if the previous produced True
- **andthen** guarantees that always when indexing then $i \leq n$ holds
  - ⇒ it suffices to include $i \geq 1$ into the invariant
  - ⇒ the precondition $n \geq 1$ is not needed

It is worth to pay attention to followings:

- side effects: can a function or a subprogram modify a variable that is not intended to be modified?
- "aliasing": can two different names mean the same variable (then modification of the other modifies both)?
  - e.g. $A[i]$, $A[j]$ when $i = j$
  - e.g. reference parameters of subprograms or a reference parameter vs. a non-local variable
  - e.g. pointers

- macros: is the structure preserved as desired in the outfolding?
  - cf. `#define sqr(x) = x*x`
    `sqr(2+3)` $\leadsto$ `2+3*2+3` instead of `(2+3)*(2+3)`
- initialisation of variables: has the program unrealised preconditions of the form "$i = 0$"?
- pathological inputs: does e.g. the program that uses an array $A[1 \ldots n]$ assume that $n \geq 1$
  - in this kind of cases it is usually reasonable allow $n = 0$ but not $n < 0$
  - memory allocation and deallocation
    - is garbage memory generated?
    - has references to deallocated memory been generated?
  - has there been prepared to all possible results of the instruction?
    - e.g. are all different return codes of a service been processed in an appropriate way?
  - can the stack run out?
    - use of stack $\leadsto$ recursion $\leadsto$ number and size of local variables
    - size of arrays
    - amount of dynamically allocated memory

(these are difficult to check using proof techniques but sometimes there can be found suitable invariants which can be used to estimate the number of records)

- can the domains overflow?
  - especially the integers of 1 or 2 bytes

  (also this is usually difficult to check)

- problems of concurrency: can another program intrude into a critical section?

- ...

⇒ OHJ-2050 Ohjelmointikielten periaatteet (Principles of Programming Languages)

Example: a program for matching strings of text (text elements)

- there has been given:
  - a line of characters in array: $line[1 \ldots linelength]$
  - an array of text elements: $element[1 \ldots nofelements]$
    length of element i: $element[i].length]$
  - characters of element i: $element[i].c[1 \ldots element[i].length]$

– the following part of a program:

```
1.  k := 1; (* k = cursor *)
2.  while k ≤ linelength do
3.      if line[k] =' ' then k := k + 1
4.      else
5.          match := False; i := 0;
6.          while i < nofelements ∧ ¬match do
7.              i := i + 1; match := True;
8.              for j := 1 to element[i].length do
9.                  match := match ∧
10.                 element[i].c[j] = line[k + j − 1]
11.             endfor
12.         endwhile
13.         if match then
14.             write(i); k := k + element[i].length
15.         else
16.             write('virhe kohdassa ', k)
17.         endif
18.     endif
19. endif
```

- let us start from item 2: does it do anything illegal?
- by taking into account the use of th eprogram it is reasonable to do following assumptions:
  - $linelength \geq 0$, $nelements \geq 0$ (or ≥ 1?), $element[i].length \geq 1$

  - text elements have no spaces

- if there has not been given an exact specification for the program then
  - this kind of assumptions must be done
  - **these must be booked into sight!**

- notices (easy to check from the program)

  (a) $element[i].length \geq 1 \Rightarrow$ the value of k never decreases $\Rightarrow$ after line 1 always $k \geq 1$

  (b) in the lines 6,...,17 it holds $0 \leq i \leq nofelements$

- probably the following operations are illegal
  - risk for overflow in every assignement (lines 1, 3, 5ab, 7ab, 8, 9, and 14)
  - indexings of arrays (lines 3, 8, 10abc, and 14)

Checking the risk for overflow

- **assumptions:**
  - $\geq$ 16 bit integers
  - *linelength*, *nofelements*, and $element[i].length \leq 10{,}000$
- the above assumptions are done because
  - realistic: probably they hold all the time when the program is used (yet they must be booked into sight!)
  - makes the checkings easier (for comparison the situation is analysed also without them)
- lines 1, 5, 7b:
  a constant of legal size is assigned $^\bullet/_\bullet$
- line 3:
  - $k \leq linelength \leq 10{,}000 \;\Rightarrow$
    $k + 1 \leq 10{,}001 \;\Rightarrow$ no overflow
  - k increases $\Rightarrow$ no underflow $^\bullet/_\bullet$
  - (without:
    it must be $linelength <$ **maxint**)
- line 7a:
  - $i < nofelements \leq 10{,}000 \;\Rightarrow$
    $i + 1 \leq 10{,}000 \;\Rightarrow$ no overflow

  - i increases $\Rightarrow$ no underflow •/•
    (without: *nofelements* = **maxint** is allowed
    $\Rightarrow$ actualises automatically if i is of same or bigger type than *nofelements*)

- line 8; $1 \leq j \leq element[i].length \leq 10{,}000$ •/•
  - (without: $element[i].length =$ **maxint** is allowed)

- line 9: there cannot happen overflows for values of Boolean type values •/•

- line 14:
  - in line 4...13 there is not assigment into k

  $\Rightarrow$ in the beginning of this line
  $k \leq linelength \leq 10{,}000$

  $\Rightarrow$ $k + element[i].length \leq 20\ 000$ •/•
  - (without: it may be necessay to check that
    $linelength + element[i].length \leq$ **maxint**)

$\Rightarrow$ no risk for overflow **with the above assumptions!**

Indexings:

- line 3: note (a) $\Rightarrow 1 \leq k \leq linelength$ •/•

- line 8: note (b) $\wedge$ line 6,7 $\Rightarrow$
  $1 \leq i \leq nofelements$ $^{\bullet}/_{\bullet}$

- line 10a: $^{\bullet}/_{\bullet}$ because of line 8 $^{\bullet}/_{\bullet}$

- line 10b: because of line 8
  $1 \leq j \leq element[i].length$ $^{\bullet}/_{\bullet}$

- line 10c: $1 \leq j \leq element[i].length \Rightarrow$

  $$k \leq k + j - 1 \leq element[i].length + k - 1$$

  - we know about k only that
    $1 \leq k \leq linelength$

  $\Rightarrow$ nothing guarantees that
  $element[i].length + k - 1 \leq linelength$!

- an error is found: if we are so near to the end of the line that the element to be tested does not anymore totally fit into the line, then the program tries to index pass the end of the line

4.2 Designing a Program and its Proof Together

It is often worth to design a difficult part of a program together with its proof

- it is worth to document the main ideas of the progression of the program by writing predicates that describe the interphases
- it is worth to design a loop, its invariant and bound function together

Example: computing the value of a polynom using Horner's Rule

- a polynom $P(x)$ of degree n is an expression of the form:

  $$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

  - agreement $\forall x : x^0 = 1$
- straightforward computation requires
  - $2n$ multiplications
    (or foolishly doing even $(\frac{1}{2})(n^2 + n)$)
- $P(x)$ can be computed by fewer multiplications using the Horner's Rule:

  $$P(x) = (\cdots (a_n x + a_{n-1}) + x \cdots)x + a_0$$

  - n multiplications

  - n additions

- how to make a program that utilises this?
- inputs and outputs:
  - $n \geq -1$
  - $a_0 \dots a_n$ is given in array $A[0 \dots n]$
  - $A[0 \dots n]$ is fixed
  - the results is composed into variable Px
- target: in the end $Px = \sum_{j=0}^{n} A[j]x^j$
- the form of Horner's Rule induces to progress from inside to outside, that is, to use a **for**-loop of the from

  for $i := n$ **downto** 0 **do**
  $\quad Px := f(i, Px)$
  endfor

  or a **while**-loop of the form

  $i := n$;
  while $i \geq 0$ **do**
  $\quad Px := f(i, Px);\ i := i + 1$
  endwhile

- for creating the invariant and finding f we divide the value of the polynom into two parts:

  - Px presents the already computed part
  - the not yet computed part is expressed as the formula

$$\begin{aligned} Px &= (\cdots((\cdots(a_n x + a_{n-1})x + \cdots)x + a_i)x + \cdots)x + a_0 \\ &= (\cdots((\qquad\qquad Px \qquad\qquad)x + a_i)x + \cdots)x + a_0 \end{aligned}$$

$\Rightarrow$ the invariant

$$P(x) = (\cdots((Px)x + a_i)x + \cdots)x + a_0$$

- in the beginning $i = n$ so it must be $Px \cdot x = 0$
  - can be set to hold by assigning $Px := 0$
- in the end $i = -1$ so $P(x) = Px$
- in every cycle of the loop Px extends to cover a new parenthesis expresion

  $\Rightarrow$ we choose $f(i, Px) = Px \cdot x + A[i]$

- therefore

```
Px := 0; i := n;
while i ≥ 0 do
    Px := Px · x + A[i]; i := i + 1
endwhile
```

or as a **for**-loop

```
Px := 0;
```

for $i := n$ **downto** 0 **do**
$Px := Px \cdot x + A[i]$
endfor

- just in case we prove this to be correct using the ordinary expression of polynom and the invariant written in sum-form
  - we have to add $i \geq -1$ into invariant

$\langle\ n \geq -1\ \rangle$
$Px := 0;\ i := n;$
$\langle\ Px = 0 \wedge i = n \geq -1\ \rangle$
$\langle$ **inv:** $Px \cdot x^{i+1} + \sum_{j=0}^{i} A[j]x^j\ =$
$\sum_{j=0}^{n} A[j]x^j \wedge i \geq -1 \rangle$
while $i \geq 0$ **do**
$Px := Px \cdot x + A[i];\ i := i - 1$
endwhile
$\langle\ Px = \sum_{j=0}^{n} A[j]x^j\ \rangle$

- checking the loop
  - $Px = 0 \wedge i = n \geq -1\ \Rightarrow$ $Px{\cdot}x^{i+1} + \sum_{j=0}^{n} A[j]x^j = 0 + \sum_{j=0}^{n} A[j]x^j \wedge$ $i \geq -1\ \Rightarrow\ I$ •/•
  - $wp(\ Px := Px \cdot x + A[i];\ i := i - 1,\ I\ )$
    $\Leftrightarrow\ (\ Px \cdot x + A[i]\ ) \cdot x^i + \sum_{j=0}^{i-1} x^j\ =$
    $\sum_{j=0}^{n} A[j]x^j \wedge i \geq 0$
    $\Leftrightarrow\ Px \cdot x^{i+1} + \sum_{j=0}^{i} A[j]x^j\ =$
    $\sum_{j=0}^{n} A[j]x^j \wedge i \geq 0,$

so
$\langle\ I \wedge i \geq 0\ \rangle$
$Px := Px \cdot x + A[i];\ i := i - 1$
$\langle\ I\ \rangle$ •/•

- $I \wedge i < 0 \ \Rightarrow\ I \wedge i = -1$
  $\Rightarrow\ Px = Px \cdot x^0 + 0 = \sum_{j=0}^{n} A[j]x^j$ •/•
- termination: $i + 1$ is a bound function

Example: the task is to compute the cube of a natural number to in linear time without operations "raising to a power" nor multiplication.

- specification: the input for the program must be a natural number (non-negative integer) that is not allowed to modify during the execution and the program returns it raised to power of three (3).

  n is fixed
  $\langle\ n \geq 0\ \rangle$
  Compute_cube
  $\langle\ r = n^3\ \rangle$

- it seems obvious that our program must have at least one loop structure
  $\Rightarrow$ we can think that we have a variable i whose cube is computed in every cycle

- an invariant candidate would thus be $r = n^3$

- the loop must terminate when $i = n$ and then it would be $r = i^3 \wedge i = n$, which implies the postcondition

⇒ now our Compute_cube looks like this:

$\langle\ n \geq 0\ \rangle$
$i :=$ *expression1*;
$r :=$ *expression2*;
$\langle$ **inv:** $r = i^3\ \rangle$
while $i < n$ **do**
    $i :=$ *expression3*;
    $r :=$ *expression4*
endwhile
$\langle\ r = i^3 \wedge i = n\ \rangle$
$\langle\ r = n^3\ \rangle$

- we must choose $i = 0$ for the initial value of i for that we can compute the cube also when $n = 0$
- for that also **inv** would hold also r must be initialised to 0.
- for that the loop would terminate i must be increased in every cycle of the loop

⇒ now our Compute_cube looks like this:

$\langle\ n \geq 0\ \rangle$
$i := 0;$
$r := 0;$
$\langle$ **inv:** $r = i^3\ \rangle$
while $i < n$ **do**
  $i := i + 1;$
  $r :=$ *expression4*
endwhile
$\langle\ r = i^3 \wedge i = n\ \rangle$
$\langle\ r = n^3\ \rangle$

- the next thing is to specify *expression4*
- it must satisfy the condition (why?):
  $\langle$ **inv** $\wedge\ i < n\ \rangle\ i := i + 1; r :=$ *expression4*
  $\langle$ **inv** $\rangle$
- this is equivalent to
  inv $\wedge\ i < n \Rightarrow$
  $wp(\ i := i + 1;\ r :=$ *expression4*, **inv** $)$
- let us assume that **inv** $\wedge\ i < n$ holds and calculate *expression4* so that
  $wp(\ i := i + 1;\ r :=$ *expression4*, $r = i^3\ ) \Leftrightarrow$
  expression4 $= (i + 1)^3 = i^3 + 3i^2 + 3i + 1$

⇒ we assumed $r = i^3$, so we get
expression4 $= r + 3i^2 + 3i + 1$

- we introduce a new variable s and strengthen the invariant by term $s = 3i^2 + 3i + 1$

$\Rightarrow$ we can write $expression4 = r + s$

- we must also extend our program so that it would initialise s correct and would increase it correct in every cycle

$\Rightarrow$ now our Compute_cube looks like this:

$\langle\ n \geq 0\ \rangle$
$i := 0;$
$r := 0;$
$s := expression5;$
$\langle$ **inv:** $r = i^3 \wedge s = 3i^2 + 3i + 1\ \rangle$
while $i < n$ **do**
  $i := i + 1;$
  $r := r + s;$
  $s := expression6$
endwhile
$\langle\ r = i^3 \wedge i = n\ \rangle$
$\langle\ r = n^3\ \rangle$

- clearly s must be initialised to 1

- similarly as *expression4*, *expression6* must be specified so that

  $\langle$ **inv** $\wedge\ i < n\ \rangle$
  $i := i + 1;\ r := r + s;\ s := \mathit{expression6}$
  $\langle$ **inv** $\rangle$

  holds:

  $$\begin{aligned}
  & wp(\ i := i + 1;\ r := r + s;\ s := \mathit{expression6}, \\
  & \quad r = i^3 \wedge s = 3i^2 + 3i + 1\ ) \ \Leftrightarrow \\
  & wp(\ i := i + 1;\ r := r + s, \\
  & \quad r = i^3 \wedge \mathit{expression6} = 3i^2 + 3i + 1\ ) \ \Leftrightarrow \\
  & wp(\ i := i + 1, \\
  & \quad r + s = i^3 \wedge \mathit{expression6} = 3i^2 + 3i + 1\ ) \ \Leftrightarrow \\
  & r + s = (i + 1)^3 \wedge \\
  & \quad \mathit{expression6} = 3(i + 1)^2 + 3(i + 1) + 1 \ \Rightarrow
  \end{aligned}$$

  since $r + s = (i + 1)^3$ follows from the invariant
  $(r + s = i^3 + 3i^2 + 3i + 1 = (i + 1)^3)$, we get

  $$\begin{aligned}
  \mathit{expression6} &= 3(i + 1)^2 + 3(i + 1) + 1 \\
  &= 3i^2 + 9i + 7
  \end{aligned}$$

  and since $s = 3i^2 + 3i + 1$, we get

  $$\begin{aligned}
  \mathit{expression6} &= s + 6i + 6 \\
  &= s + i + i + i + i + i + i + 6
  \end{aligned}$$

- we could now write the final program but let us make it more elegant by introducing a new variable t, and with it we can write

  $$\mathit{expression6} = s + t$$

- at same time we strengthen the invariant by term $t = 6i + 6$

⇒ we initialise t to 6 for that the new invariant would hold

⇒ now our Compute_cube looks like this:

$\langle\ n \geq 0\ \rangle$
$i := 0;\ r := 0;\ s := 1;\ t := 6;$
$\langle$ **inv:** $r = i^3 \wedge s = 3i^2 + 3i + 1 \wedge t = 6i + 6\ \rangle$
while $i < n$ **do**
  $i := i + 1;\ r := r + s;\ s := s + t;$
  $t :=$ *expression7*
endwhile
$\langle\ r = i^3 \wedge i = n\ \rangle$
$\langle\ r = n^3\ \rangle$

- we must still specify *expression7* so that

$\langle$ **inv:** $\wedge\ i < n\ \rangle$
$i := i{+}1;\ r := r{+}s;\ s := s{+}t;\ t :=$ *expression7*
$\langle$ **inv** $\rangle$

holds:

wp($i := i + 1;\ r := r + s;$
  $s := s + t;\ t :=$ *epression7*,
  $r = i^3 \wedge s = 3i^2 + 3i + 1 \wedge t = 6i + 6$) $\Leftrightarrow$
$r + s = (i + 1)^3 \wedge$
  $s + t = 3(i + 1)^2 + 3(i + 1) + 1 \wedge$
  expression7 $= 6(i + 1) + 6 \quad \Rightarrow$
since the first parts follow directly from the invariant, we get
expression7 $= 6(i + 1) + 6 = 6i + 12 = t + 6$

⇒ now final Compute_cube looks like this:

$\langle\ n \geq 0\ \rangle$
$i := 0;\ r := 0;\ s := 1;\ t := 6;$
$\langle\ \textbf{inv:}\ r = i^3 \land s = 3i^2 + 3i + 1 \land t = 6i + 6\ \rangle$
while $i < n$ **do**
  $i := i + 1;\ r := r + s;\ s := s + t;\ t := t + 6$
endwhile
$\langle\ r = i^3 \land i = n\ \rangle$
$\langle\ r = n^3\ \rangle$

- since i is used only for counting the cycles, increasing it can be done also at the end of the loop

⇒ therefore Compute_cube implemented using **for**-loop looks like this:

$r := 0;\ s := 1;\ t := 6;$
for $i := 0$ **to** $n - 1$ **do**
  $r := r + s;\ s := s + t;\ t := t + 6$
endfor

- the execution time with respect to n is clearly linear (why?)

Example: the longest ascending inner sequence: stating the task

- sequence β is *inside* sequence α or its *inner sequence*, if β can be constructed from α by removing 0 or more elements
  - e.g. the inner sequences of the sequence $\langle 1, 9, 2, 1 \rangle$ are $\langle\rangle$, $\langle 1 \rangle$, $\langle 9 \rangle$, $\langle 2 \rangle$, $\langle 1, 9 \rangle$, $\langle 1, 2 \rangle$, $\langle 1, 1 \rangle$, $\langle 9, 2 \rangle$, $\langle 9, 1 \rangle$, $\langle 2, 1 \rangle$, $\langle 1, 9, 2 \rangle$, $\langle 1, 9, 1 \rangle$, $\langle 1, 2, 1 \rangle$, $\langle 9, 2, 1 \rangle$, and $\langle 1, 9, 2, 1 \rangle$
- a sequence is
  - *ascending* if next element is always greater than the previous
  - *non-descending* if next element is always at least equal than previous (is greater than or equal to)
- the task is to make a program for which is given a sequence in array $A[1 \ldots n]$, and which produces a number k and array $B[1 \ldots k]$ such that B is as long as possible ascending inner sequence of A

Longest ascending inner sequence: specification

- input: fixed array $A[1 \ldots n]$
- to specify the output we need a predicate to express that $B[1 \ldots k]$ is an ascending inner sequence of A; let us define it in stages:

- $innersequence(B[1 \ldots k], A[1 \ldots n]) :\Leftrightarrow$
  $\exists j_1, j_2, \ldots, j_k :$
  $1 \leq j_1 < j_2 < \cdots < j_k \leq n \wedge$
  $\forall i \; ; \; 1 \leq i \leq k : B[i] = A[j_i]$
- $ascending(B[1 \ldots k]) :\Leftrightarrow$
  $\forall i \; ; \; 1 \leq i < k : B[i] < B[i+1]$
- $ais(B[1 \ldots k], A[1 \ldots n]) :\Leftrightarrow$
  $ascending(B) \wedge innersequence(B, A)$

- in the end B must be *maximal*, that is, as long as possible ascending inner sequence
  - can be expressed e.g. $ais(C[1 \ldots m], A) \rightarrow m \leq k$
  - it is easy to notice that

    $$ais(C[1 \ldots m], A) \wedge 0 \leq j < m \;\Rightarrow\; ais(C[1 \ldots j], A)$$

  $\Rightarrow$ an easier way: $\forall C[1 \ldots k+1] : \neg ais(C, A)$

$\Longrightarrow$ let us define the maximal ascending inner sequence:

$$mais(B[1 \ldots k], A[1 \ldots n]) :\Leftrightarrow ais(B, A) \wedge \forall C[1 \ldots k+1] : \neg ais(C, A)$$

- so the specification of the program will be

$\langle$ True $\rangle$
$\quad S$
$\langle\; mais(B[1 \ldots k], A[1 \ldots n]) \;\rangle$

Longest ascending inner sequence: beginning of the design of the program

- let us try "piece by piece" strategy: we try to collect the answer by browsing through A in a loop

$\Rightarrow$ a guess for the loop and its invariant I:

$\langle$ **inv** : $mais(B[1 \ldots k], A[1 \ldots i-1) \rangle$
for $i := 1$ **to** n **do**
  ??
endfor

  - I is trivially true when $i = 1$, if we initialise $k := 0$
  - $I \wedge i = n+1 \Rightarrow mais(B, A)$

- the program looks like this:

$\langle$ True $\rangle$
$k := 0;$
$\langle$ **inv:** $mais(B[1 \ldots k], A[1 \ldots i-1]) \rangle$
for $i := 1$ **to** n **do**
  $\langle mais(B[1 \ldots k], A[1 \ldots i-1]) \wedge 1 \leq i \leq n \rangle$
  ??
  $\langle mais(B[1 \ldots k], A[1 \ldots i]) \rangle$
endfor
$\langle mais(B[1 \ldots k], A[1 \ldots n]) \rangle$

- what must be done for that the invariant still holds when i increases by one?

- if $A[i] > B[k]$ it suffices to concatenate $A[i]$ to the end of B

  if $A[i] > B[k]$ **then**
  $\quad k := k + 1;\ B[k] := A[i]$
  else
  $\quad$??
  endif

- if $A[i] \leq B[k]$ then $B[1 \ldots k]$ is an ascending inner sequence of $A[1 \ldots i]$ but not necessary maximal!
  - e.g. $A = \langle 8, 1, 2, 4, 0 \rangle, k = 1, B = \langle 8 \rangle, i = 3$

$\Rightarrow$ it seems necessary to remember to log at least some alternatives for ascending inner sequences

Which inner sequences need to be remembered?

- even a very short inner sequences may grow the longest
  - e.g. $\langle 6, 7, \mathbf{1}, 8, 9, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5} \rangle$
- of sequences of same length the one which ends to smaller, cannot be worse than the one which ends to greater
  - e.g. $\langle 6, \mathbf{1}, 7, \mathbf{2}, 9, \mathbf{8}, \ldots \rangle$

$\Rightarrow$ it is worth to remember for each length of the inner sequences one inner sequence that ends to as small element as possible

- e.g. $\langle 4, 6, 1, 9, 2 \rangle$

	inners sequences as a function of i				
length	1	2	3	4	5
1	$\langle 4 \rangle$	$\langle 4 \rangle$	$\langle 1 \rangle$	$\langle 1 \rangle$	$\langle 1 \rangle$
2	—	$\langle 4, 6 \rangle$	$\langle 4, 6 \rangle$	$\langle 4, 6 \rangle$	$\langle 1, 2 \rangle$
3	—	—	—	$\langle 4, 6.9 \rangle$	$\langle 4, 6, 9 \rangle$
4	—	—	—	—	—

- let these sequences be as B_h, where h is the length
- let k = be the biggest used value of h
- to make the border cases easier let $B_0 = \langle\, \rangle$

- we need a predicate "the ascending inner sequence that ends to smallest element" which claims that $B[1 \ldots h]$ has the smallest last element of the h-length ascending inner sequences of A

$$
\begin{aligned}
&\mathit{seais}(B[1 \ldots h], A[1 \ldots i]) :\Leftrightarrow \\
&\quad \mathit{ais}(B, A[1 \ldots i]) \wedge \\
&\quad \forall C[1 \ldots h] : \neg \mathit{ais}(C, A[1 \ldots i]) \vee \\
&\qquad\qquad C[h] \geq B[h]
\end{aligned}
$$

⇒ the program looks now like this:

$\langle$ True $\rangle$
$k := 0;$
$\langle$ **inv:** $\forall h \; ; \; 1 \leq h \leq k :$
$\quad seais(B_h[1 \ldots h], A[1 \ldots i-1]) \land$
$\quad mais(B_k[1 \ldots k], A[1 \ldots i-1]) \rangle$
for $i := 1$ **to** n **do**
$\quad \langle \forall h \; ; \; 1 \leq h \leq k :$
$\quad seais(B_h[1 \ldots h], A[1 \ldots i-1]) \land$
$\quad mais(B_k[1 \ldots k], A[1 \ldots i-1]) \land 1 \leq i \leq n \rangle$
$\quad ??$
$\quad \langle \forall h \; ; \; 1 \leq h \leq k :$
$\quad seais(B_h[1 \ldots h], A[1 \ldots i-1]) \land$
$\quad mais(B_k[1 \ldots k], A[1 \ldots i-1]) \rangle$
endfor
$\langle \; mais(B_k[1 \ldots k], A[1 \ldots n]) \; \rangle$
$B := B_k$
$\langle \; mais(B[1 \ldots k], A[1 \ldots n]) \; \rangle$

Maintaining of inner sequences

- how the inner sequences B_h must be modified when $A[i]$ is handled?
- $A[i]$ can be put to the end of only the inner sequences for which last element it is greater
  - in other words, $A[i] > B_h[h]$

  - for this case anything can be put to the end of B_0

  $\Rightarrow$ we require $h = 0 \vee A[i] > B_h[h]$

- actually, making B_h longer does not change B_h but renews B_{h+1}
- on the other hand, it is worth to make B_h longer by $A[i]$ only if we thereby get a sequence that is longer than all the previous, or a better sequence of length $h + 1$
  - in other words, $h = k \vee A[i] < B_{h+1}[h+1]$
  - (when $A[i] = B_{h+1}[h+1]$ the lengthening causes no disbenefit nor benefit)

$\Rightarrow$ B_h is made longer (in other words, B_{h+1} is renewed) if and only if

$$(h = 0 \vee A[i] > B_h[h]) \wedge$$
$$(h = k \vee A[i] < B_{h+1}[h + 1])$$

⇒ the program

$\langle$ True $\rangle$
$k := 0;$
$\langle$ **inv:** $\forall h \; ; \; 1 \le h \le k :$
$\quad seais(B_h[1\ldots h], A[1\ldots i-1]) \wedge$
$\quad mais(B_k[1\ldots k], A[1\ldots i-1]) \rangle$
for $i := 1$ **to** n **do**
$\quad \langle \forall h \; ; \; 1 \le h \le k :$
$\quad seais(B_h[1\ldots h], A[1\ldots i-1]) \wedge$
$\quad mais(B_k[1\ldots k], A[1\ldots i-1]) \wedge 1 \le i \le n \rangle$
$\quad \langle$ **inv:** ?? $\rangle$
$\quad$ **for** $j := 0$ **to** k **do**
$\quad\quad$ **if** $(j = 0$ **orelse** $A[i] > B_j[j]) \wedge$
$\quad\quad\quad (j = k$ **orelse** $A[i] < B_{j+1}[j+1])$ **then**
$\quad\quad\quad B_{j+1}[1\ldots j] := B_j;\; B_{j+1}[j+1] := A[i]$
$\quad\quad\quad$ **if** $j = k$ **then** $k := k+1$ **endif**
$\quad\quad$ **endif**
$\quad$ **endfor**
$\quad \langle \forall h \; ; \; 1 \le h \le k :$
$\quad seais(B_h[1\ldots h], A[1\ldots i-1]) \wedge$
$\quad mais(B_k[1\ldots k], A[1\ldots i-1]) \rangle$
endfor
$\langle \; mais(B_k[1\ldots k], A[1\ldots n]) \; \rangle$
$B := B_k$
$\langle \; mais(B[1\ldots k], A[1\ldots n]) \; \rangle$

- three nested **for**-loops (where is the third?)
  ⇒ seems quite unefficient!

Enhancing the book-keeping of inner sequences 1

- at worst $k = n$

  ⇒ in arrays $B_0, \ldots, B_k$ there are $\Theta(n^2)$ elements in all

  – but sequences are composed only on n elements

  ⇒ could we avoid having every sequence in its own B array?

⇒ questions

  – on which different ways the same element can be in several sequences?

  – to the end of which sequences $A[i]$ can be put?

- except the borderline cases $h = 0$ and $h = k$, $A[i]$ is concatenated to the end of sequence B_h only if

$$B_h[h] < A[i] < B_{h+1}[h+1]$$

  but then

$$B_h[h] < B_{h+1}[h+1]$$

- question: when $B_h[h] < B_{h+1}[h+1]$?
- answer: if $B_h[h] \geq B_{h+1}[h+1]$ then $B_h[h] \geq B_{h+1}[h+1] > B_{h+1}[h]$

  - but then B_h is not an ascending inner sequence of length of h that ends to smallest character ↯
  - therefore $B_h[h] < B_{h+1}[h+1]$ holds always when $1 \leq h < k$

- case $h = 0$: $A[i]$ is concatenated to the end of sequence B_0 if and only if it creates a sequence B_1 or improves it
  - in other words, $k = 0 \vee A[i] < B_1[1]$
- case $h = k$: $A[i]$ is concatenated to the end of sequence B_k if and only if

$$k = 0 \vee A[j] > B_k[k]$$

- $A[i]$ is concatenated to the end of sequence B_h if and only if

$$(h = 0 \vee B_h[h] < A[i]) \wedge$$
$$(h = k \vee A[i] < B_{h+1}[h+1])$$

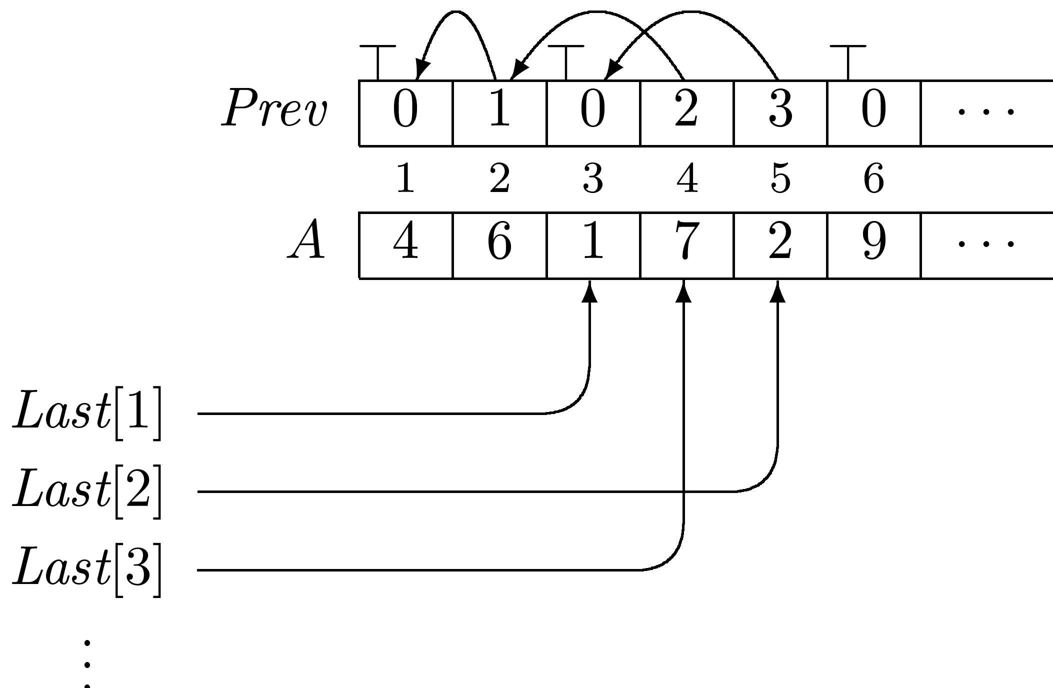
⇒ $B_1[1] < \cdots < B_k[k]$ define $k+1$ spaces and $A[i]$ is added into

  - one of them if $A[i] \neq B_h[h]$ for all $1 \leq h \leq k$
  - none if $A[i] = B_h[h]$ for some $1 \leq h \leq k$

⇒ $A[i]$ is concatenated to the end of at most one sequence

⇒ the sequences can be represented by remembering the location of the last element of each sequence, and linking into location i a pointer which tells the location of the previous element of the sequence

- let us introduce arrays $Last[0 \ldots n]$ and $Prev[1 \ldots n]$
  - $Last[h]$ the location of the last element of sequence of length h
  - $Prev[i]$ the location of previous element of the element in location i



- to make the program simpler we make an agreement that

  - $Prev[i] = 0$ when there is no previous element
  - $Last[0]$ exists ($\Rightarrow \ Last[0] = 0$)

(These could be avoided by suitable **if**-statements)

- for "reading" sequences B_h from *Last* and *Prev* we define
  - $Prev^0[x] = x$ if $1 \leq x \leq n$
  - $Prev^{i+1}[x] = Prev[Prev^i[x]\,]$ if $1 \leq Prev^i[x] \leq n$
  - $B_h = \langle\, A[Prev^{h-1}[Last[h]\,]\,], \ldots, A[Prev^1[Last[h]\,]\,], A[Prev^0[Last[h]\,]\,]\,\rangle$

- according to the previous

$\langle\, \forall h \ ; \ 1 \leq h \leq k : seais(B_h[1 \ldots h], A[1 \ldots i-1])$
$\ \wedge\ mais(B_k[1 \ldots k], A[1 \ldots, i-1]) \wedge 1 \leq i \leq n \,\rangle$
if for some h holds
$(h = 0 \vee A[Last[h]\,] < A[i]) \wedge$
$(h = k \vee A[i] < A[Last[h+1]\,])$
then execute
$Prev[i] := Last[h];\ Last[h+1] := i;$
if $h = k$ **then** $k := k + 1$ **endif**
$\langle\, \forall h \ ; \ 1 \leq h \leq k : seais(B_h[1 \ldots h], A[1 \ldots i])$
$\ \wedge\ mais(B_k[1 \ldots k], A[1 \ldots, i]) \,\rangle$

- this way the program transforms into the following form

```
⟨ True ⟩
k := 0; Last[0] := 0;
⟨ inv: ∀h ; 1 ≤ h ≤ k :
        seais(B_h[1 ... h], A[1 ... i − 1]) ∧
        mais(B_k[1 ... k], A[1 ... i − 1]) ⟩
for i := 1 to n do
    ⟨ inv: ?? ⟩
    k' := k;
    for j := 0 to k' do
        if ( j = 0 orelse A[i] > A[Last[j] ] ) ∧
           ( j = k orelse A[i] < A[Last[j + 1] ] ] )
        then
            Prev[i] := Last[j]; Last[j + 1] := i;
            if j = k then k := k + 1 endif
        endif
    endfor
    ⟨ ∀h ; 1 ≤ h ≤ k :
      seais(B_h[1 ... h], A[1 ... i − 1]) ∧
      mais(B_k[1 ... k], A[1 ... i − 1]) ⟩
endfor
⟨ mais(B_k[1 ... k], A[1 ... n]) ⟩
l := Last[k];
⟨ inv: mais(B_k[1 ... k], A[1 ... n]) ∧
        l = Prev^(k−i)[Last[k] ] ∧
        ∀h ; i + 1 ≤ h ≤ k : B[h] = B_k[h] ⟩
for i := k downto 1 do
    B[i] := A[l]; l := Prev[l]
endfor
⟨ mais(B[1 ... k], A[1 ... n]) ⟩
```

Enhancing the book-keeping of inner sequences 2

- we know that $A[i]$ is concatenated to the end of at most one sequence

⇒ the innermost **for**-loop is needed for searching the correct place — could it be done more efficiently?

- yes, it could, by binary search!

$\langle$ True $\rangle$
$k := 0;\ Last[0] := 0;$
$\langle$ **inv₁:** $\forall h\ ;\ 1 \le h \le k :$
$\quad seais(B_h[1 \ldots h], A[1 \ldots i-1]) \wedge$
$\quad mais(B_k[1 \ldots k], A[1 \ldots i-1]) \rangle$
for $i := 1$ **to** n **do**
$\quad a := 1;\ y := k+1;$
$\quad$ **while** $a < y$ **do**
$\quad\quad v := (a+y)$ div $2;$
$\quad\quad$ **if** $A[Last[v]\,] < A[i]$ **then** $a := v+1$
$\quad\quad$ **else** $y := v$
$\quad\quad$ **endif**
$\quad$ **endwhile**
$\quad \langle$ $\mathbf{inv_1} \wedge 1 \le a \le k+1 \wedge$
$\quad\quad (\, a = 1 \vee A[Last[a-1]\,] < A[i]\,) \wedge$
$\quad\quad (\, a = k+1 \vee A[Last[a]\,] \ge A[i]\,) \rangle$
$\quad \langle \forall h\ ;\ 1 \le h \le k :$
$\quad\quad seais(B_h[1 \ldots h], A[1 \ldots i-1]) \wedge$
$\quad\quad mais(B_k[1 \ldots k], A[1 \ldots i-1]) \rangle$

if $a = k + 1$ **orelse** $A[\mathit{Last}[a]\,] > A[i]$ **then**
  $\mathit{Prev}[i] := \mathit{Last}[a-1];\ \mathit{Last}[a] := i$
endif
if $a = k + 1$ **then** $k := a$ **endif**
$\langle\, \forall h \ ;\ 1 \le h \le k :$
  $\mathit{seais}(B_h[1 \ldots h], A[1 \ldots i]) \wedge$
  $\mathit{mais}(B_k[1 \ldots k], A[1 \ldots i])\, \rangle$
endfor
$\langle\ \mathit{mais}(B_k[1 \ldots k], A[1 \ldots n])\ \rangle$
$v := \mathit{Last}[k];$
$\langle$ **inv:** $\mathit{mais}(B_k[1 \ldots k], A[1 \ldots n]) \wedge$
  $v = \mathit{Prev}^{k-i}[\mathit{Last}[k]\,] \wedge$
  $\forall h \ ;\ i+1 \le h \le k : B[h] = B_k[h]\, \rangle$
for $i := k$ **downto** 1 **do**
  $B[i] := A[v];\ v := \mathit{Prev}[v]$
endfor
$\langle\ \mathit{mais}(B[1 \ldots k], A[1 \ldots n])\ \rangle$

execution time

- **while**-loop $O(\lg k) \le O(\lg n)$
- 1. **for**-loop $O(n)$ cycles
- 2. **for**-loop $O(k) \le O(n)$ cycles
- rest of the operations constant time

$\Rightarrow$ altogether $(O(n \lg n)$

- surprisingly fast!

About choosing the level for formality

- in the specification of the inner sequence program we needed quite long state predicates
  - *mais* and *seais*
  - the corresponding notions are not very complicated

⇒ the required reasonings were laborous to formalise, but not very complicated

  - we did not formalise very rigorously

- formalisation is a tool, not a purpose itself!
  - the derivation and proof of the program would not have significally worsened even we had partly left *mais* and *seais* without formalisation
  - formalisation guaranteed that their meaning is unique

Analysis of the design of the inner sequence program

- the development of the program was a combination of careful analysis of situations and properties of inner sequences and use of known algorithmic means
  - proof techniques of programs supported the analysis well
- for choosing the algorithmic means the previous experience helped a lot
  - presentation of sequences B_h using "backwards" pointers
  - use of binary search
- in the mathematical handling the previous experience helped a lot
  - making of state predicates and invariants
  - introducing function $Prev^l$
  - finding properties of inner sequences
- in some cases we proceeded by guessing
  - e.g. choosing "piece by piece" -strategy
  - the guesses may have also failed
  - then it had been necessary to try also other strategies, e.g. "divide and conquer"

  - strategy "dynamic programming" would have led to essentially same solution (dynamic programming is a method for solving complex problems by breaking them down into simpler steps)

- sometimes we used the non-realistic "theoretical" data structure
  - sequences B_h
  - a risk: in advance it was almost sure that they can be implemented somehow, but it was not sure that they can be implemented efficiently enough
  - although sequences B_h are missing from the final program they are essential for understanding how it works

    $\Rightarrow$ they were left into the final state predicates

    * state predicates are a good means to document "hidden" structures of a program
  - conclusions
    * nothing can replace experience
    * mathematical reasoning is a useful tool
    * reasoning requires familiarizing with the rules of the operation area of the program

- after thorough reasoning the effect of modifications of the program is easy to estimate
    - * exercises

5 Proving Algorithms

In proving algorithms the invariant and bound functions techniques are used versatilely

- ordinary loop invariants and bound functions
- invariants that are interpreted by the point of view of the data structure in question, e.g. a graph algorithm:
  - *the children of a black node are grey or black*
  - *the nodes in stack and only they are grey*
- criterians for progression based on a data structure
  - *the colour of a node can change only*

    $$\textit{white} \rightarrow \textit{grey} \rightarrow \textit{black}$$

  - *in every cycle the colour of at least one node changes*
- invariants between executions and which concern a data structure
  - e.g. the fundamental property of a binary search tree

    $$\begin{array}{ll} & \textit{the keys of nodes of the left sub-tree} \\ \leq & \textit{the key of the current node} \\ \leq & \textit{the keys of nodes of the right sub-tree} \end{array}$$

Often some application-specific knowledge is needed

- e.g.
  - properties of paths in a graph
  - the relation between the solution and a sub-solution (greedy algorithms)
  - number theory (hash functions, encryption)

$\Rightarrow$ it is needed to develop a tiny theory of how the objects of the application area behave

In this chapter we go through the topic by examples (depending on how much we have time).

5.1 Using an Abstract Data Structure

Example: going through all the nodes of a graph, to which there exist a path from node v_0

- in other words, which are *reachable* from v_0

The algorithm and its specification

- the presentation of the graph consists of two components
  - V: the set of nodes
  - E: the set of edges, $E \subseteq V \times V$
- since we need starting edges of nodes, we define – to make it easier to refer them in future – we define function

$$neighbours(u) = \{\, v \mid (u, v) \in E \,\}$$

- repetition
  - an abbreviation:
    $u \rightarrow v \;\Leftrightarrow\; v \in neighbours(u)$
  - $v_0 \rightarrow^* v$ if and only if
    $\exists n \in \mathbb{N} : \exists v_1, \ldots, v_n :$
    $v = v_n \wedge v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n$
- $V, E, neighbours$, and v_0 are fixed
- the type of variables u, v, and v_0 is V, and *unfinished* and *found* are of type 2^V

- *neighbours*: function $V \rightarrow 2^V$

```
1  unfinished := {v0}; found := {v0};
2  while unfinished ≠ ∅ do
3      choosesome u ∈ unfinished;
4      forall v ∈ neighbours(u) do
5          if v ∉ found then
6              found := found ∪ {v};
7              unfinished := unfinished ∪ {v}
8          endif
9      endfor;
10     unfinished := unfinished − {u}
11 endwhile
   ⟨ found = { v ∈ V | v0 →* v } ⟩
```

- important notice: after the initialisation no assignment decreases set *found*

Invariant of **while**-loop:

$$\begin{array}{ll} & unfinished \subseteq found \wedge v_0 \in found \\ \wedge & (\forall u \in found : v_0 \rightarrow^* u) \\ \wedge & (\forall u \in found - unfinished : \\ & \quad \forall v \in neighbours(u) : v \in found) \end{array}$$

- in other words,
  - the unfinished nodes are found nodes
  - the initial node v_0 is found
  - only reachable nodes are being found

  - the neighbours of a found, finished node are found

- when entering the loop
  $unfinished = found = \{v_0\}$

$\Rightarrow$ the invariant holds

Preservation of the invariant in a cycle of the loop

- is holding of $unfinished \subseteq found$ preserved?
  - the modifications concerning sets *unfinished* and *found* have to be checked
  - line 6 does not decrease *found* $\Rightarrow$ harmless
  - line 7 is harmless since at the same time also line 6 is executed
  - line 10 does not increase *found* $\Rightarrow$ harmless

  $\Rightarrow$ it is preserved

- is holding of $v_0 \in found$ preserved?
  - nothing is taken from set *found* $\Rightarrow$ yes

- is holding of $\forall u \in found : v_0 \rightarrow^* u$ preserved?
  - the modifications concerning set *found* and relation "$\rightarrow$" have to be checked

  - in the beginning of line 6 lines 3, 4, and the invariant give
    $u \in \mathit{unfinished} \subseteq \mathit{found} \wedge v \in \mathit{neighbours}(u)$
    $\Rightarrow v_0 \rightarrow^* u \rightarrow v \Rightarrow v_0 \rightarrow^* v$
    $\Rightarrow$ line 6 does not break the part in question of the invariant

  $\Rightarrow$ it is preserved

- is holding of the last part of the invariant preserved?

  - the modifications concerning sets *unfinished*, $found$, and *neighbours* have to be checked
  - the pair of lines 6,7 does not increase *found*–*unfinished* neither *neighbours*, neither decrease *found*
  - line 10 increases *found*–*unfinished* by node u, and, according to lines 4,. . . , 6 and because set *found* is never decreased, it holds for it that

$$\forall v \in \mathit{neighbours}(u) : v \in \mathit{found}$$

  $\Rightarrow$ it is preserved

$\Rightarrow$ the invariant is preserved in every cycle

Actualization of the end condition

- let $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n$ be a path

- in the end *unfinished* $= \emptyset$, therefore the invariant

$$\Rightarrow \quad v_0 \in \mathit{found} \\ \wedge \ \forall u \in \mathit{found} : \forall v : (u \rightarrow v \ \Rightarrow \ v \in \mathit{found})$$

$$\Longrightarrow v_1 \in \mathit{found} \Rightarrow v_2 \in \mathit{found} \Rightarrow \cdots \Rightarrow v_n \in \mathit{found}$$

$$\Longrightarrow \left\{ v \in V \,\middle|\, v_0 \rightarrow^* v \right\} \subseteq \mathit{found}$$

  - in other words, all the reachable nodes are found

- on the other side $\forall u \in \mathit{found} : v_0 \rightarrow^* u$ gives

$$\mathit{found} \subseteq \left\{ v \in V \,\middle|\, v_0 \rightarrow^* v \right\}$$

$\Rightarrow$ in the end $\mathit{found} = \left\{ v \in V \,\middle|\, v_0 \rightarrow^v \right\}$

Termination of **while**-loop

- there are no removals from set *found*
- in every cycle either *found* increases, or *found* remains the same and *unfinished* decreases

$\Rightarrow$ pair $(\,|\, V - \mathit{found} \,|, |\, \mathit{unfinished} \,|\,)$ is bound function

  - **assumption:** the graph is finite

Example of a concrete implementation: data structures

- set V of abstract code (=nodes)

- is expressed as numbers: $V = \{1, 2, \ldots, n\}$
- n is fixed, and $n \geq 1$
- the number of initial node v_0 in variable v_0

$\Rightarrow$ $1 \leq v_0 \leq n$

- set function *neighbours* (expresses the edges)
  - is presented as numbers 1, 2, ..., m, and as fixed arrays $\mathit{first}[1 \ldots n]$, $\mathit{edges}[1 \ldots n]$, and $\mathit{vertex}[1 \ldots m]$
  - the starting edges of node v are numbered by consecutive numbers

    $$\mathit{first}[v], \mathit{first}[v] + 1, \ldots, \\ \mathit{first}[v] + \mathit{edges}[v] - 1$$

  $\Rightarrow$ $\mathit{first}[v] \geq 1$ and $\mathit{first}[v] + \mathit{edges}[v] - 1 \leq m$
  - edge e ends to node $\mathit{vertex}[e]$

  $\Rightarrow$ $1 \leq \mathit{vertex}[e] \leq n$
  - the relation between the abstract and the concrete way of presentation

    $$\mathit{neighbours}(v) = \\ \{\, \mathit{vertex}[i] \mid \\ \mathit{first}[v] \leq i \leq \mathit{first}[v] + \mathit{edges}[v] - 1 \,\}$$

- set *unfinished*
  - array $\mathit{sequence}[1 \ldots n]$, values $1 \leq \mathit{sequence}[v] \leq n$

  - the limits for the used range l and h
  - **requirement:** always when the following formula is used, it must hold that $l \geq 1 \wedge h \leq n+1$

    $$\mathit{unfinished} = \{\, \mathit{sequence}[i] \mid l \leq i < h \,\}$$

- set *found*
  - Boolean array $\mathit{isfound}[1 \ldots n]$

    $$\mathit{found} = \{\, i \mid 1 \leq i \leq n \wedge \mathit{isfound}[i] = \mathsf{True} \,\}$$

Implementations of the abstract code with their justifications

	abstract operation	implementation
a	$\mathit{unfinished} := \{v_0\}$	$\mathit{sequence}[1] := v_0$; $l := 1;\ h := 2$

b	$\mathit{found} := \{v_0\}$	**for** $i := 1$ **to** n **do**
		$\mathit{isfound}[i] := \mathsf{False}$
		endfor;
		$\mathit{isfound}[v_0] := \mathsf{True}$

c	$\mathit{unfinished} \neq \emptyset$	$l < h$
d	choosesome $u \in \mathit{unfinished}$	$u := \mathit{sequence}[l]$
e	**forall** $v \in \mathit{neighbours}(u)$ **do**	**for** $i := \mathit{first}[u]$ **to** $\mathit{first}[u] + \mathit{edges}[u] - 1$ **do** $v = \mathit{vertex}[i]$
f	$v \notin \mathit{found}$	$\neg \mathit{isfound}[v]$
g	$\mathit{found} := \mathit{found} \cup \{v\}$	$\mathit{isfound}[v] := \mathsf{True}$
h	$\mathit{unfinished} :=$ $\mathit{unfinished} \cup \{v\}$	$\mathit{sequence}[h] := v$; $h := h + 1$
i	$\mathit{unfinished} :=$ $\mathit{unfinished} - \{v\}$	$l := l + 1$

- let us denote the value of an abstract or concrete variable x
  - before the investigated statement: x
  - after execution: x'

- in the following justifications there are assumptions that must be proven to hold before the operation is executed
  - they are proven from the concrete code

(a) $\mathit{unfinished}'$
$= \{\, \mathit{sequence}'[i] \mid l' \leq i < h' \,\}$
$= \{\, \mathit{sequence}'[i] \mid 1 \leq i < 2 \,\}$
$= \{\mathit{sequence}'[1]\}$
$= \{v_0\}$ •/•

  - requirement $l' \geq 1 \wedge h' \leq n+1$ holds, since $n \geq 1$

(b) found'
$= \{\, i \mid 1 \leq i \leq n \wedge \mathit{isfound}'[i] = \mathsf{True} \,\}$
$= \{\, i \mid 1 \leq i \leq n \wedge i = v_0 \,\} = \{v_0\}$ •/•

c $\mathit{unfinished} \neq \emptyset$
$\Leftrightarrow \{\, \mathit{sequence}[i] \mid l \leq i < h \,\} \neq \emptyset$
$\Leftrightarrow l < h$

  - **assumption:**
    while executed $l \geq 1 \wedge h \leq n+1$

(d) $u' = \mathit{sequence}[l] \in \mathit{unfinished}$

  - **assumption:** while executed, $\mathit{unfinished} \neq \emptyset$, in other words, $1 \leq l < h \leq n+1$

(e) $neighbours(u)$
$= \{\, vertex[i] \mid first[u] \leq i \leq first[u] + edges[u] - 1 \,\}$
$= \{\, v \mid \exists i : v = vertex[i] \land first[u] \leq i \leq first[u] + edges[u] - 1 \,\}$

therefore v browses set $neighbours(u)$

(f) $v \notin found$
$\Leftrightarrow\ v \notin \{\, i \mid 1 \leq i \leq n \land isfound[i] = \mathsf{True} \,\}$
$\Leftrightarrow\ v < 1 \lor v > n \lor \neg isfound[v]$
$\Leftrightarrow\ \neg isfound[v]$

- **assumption:** while executed, $1 \leq v \leq n$ (in other words, v is a legal node number)

(g) $found \cup \{v\}$
$= \{\, i \mid 1 \leq i \leq n \land isfound[i] = \mathsf{True} \,\} \cup \{v\}$
$= \{\, i \mid 1 \leq i \leq n \land isfound[i] = \mathsf{True} \lor i = v \,\}$
$= \{\, i \mid 1 \leq i \leq n \land (isfound[i] = \mathsf{True} \lor i = v) \,\}$
$= \{\, i \mid 1 \leq i \leq n \land isfound'[i] = \mathsf{True} \,\}$
$= found'$

- **assumption:** while executed, $1 \leq v \leq n$

(h) $unfinished \cup \{v\}$
$= \{\, sequence[i] \mid l \leq i < h \,\} \cup \{v\}$
$= \{\, sequence'[i] \mid l \leq i < h \,\} \cup \{sequence'[h]\}$
$= \{\, sequence'[i] \mid l \leq i < h + 1 \,\}$
$= \{\, sequence'[i] \mid l' \leq i < h' \,\}$
$= unfinished'$

- **assumption:** while executed,
  $l \geq 1 \land h' \leq n + 1$

(i) $\mathit{unfinished} - \{u\}$
$= \left\{\, \mathit{sequence}[i] \;\middle|\; l \leq i < h \,\right\} - \{\mathit{sequence}[l]\}$
$= \left\{\, \mathit{sequence}[i] \;\middle|\; l + 1 \leq i < h \,\right\}$
$= \left\{\, \mathit{sequence}'[i] \;\middle|\; l' \leq i < h' \,\right\}$
$= \mathit{unfinished}'$

- **assumption:** $u = \mathit{sequence}[l]$
- **assumption:** while executed,
  $l \geq 1 \land h \leq n + 1$

From the abstract we get concrete code by substituting the abstract operations by concrete

```
1  sequence[1] := v0; l := 1; h := 2;
   for i := 1 to n do
       isfound[i] := False
   endfor;
   isfound[v0] := True;
2  while l < h do
3      u := sequence[l];
4      for i := first[u] to first[u] + edges[u] − 1 do
           v := vertex[i];
5          if ¬isfound[v] then
6              isfound[v] := True;
7              sequence[h] := v; h := h + 1;
8          endif
9      endfor;
10     l := l + 1
11 endwhile
```

- It must also be proven that the assumptions we made are holding
  - line 2: $l \geq 1 \wedge h \leq n+1$
  - line 3: $1 \leq l < h \leq n+1$
  - line 5,6: $1 \leq v \leq n$
  - line 7: $l \geq 1 \wedge h' \leq n+1$
  - line 10: $u = sequence[l] \wedge l \geq 1 \wedge h \leq n+1$

- in the end of line 1 $l = 1$ and after that l does not decrease

  $\Rightarrow$ $l \geq 1$ in lines 2, 3, 7, and 10

- line 2 $\Rightarrow$ in line 3 $l < h$

- $\mathit{isfound}[v] := \mathsf{True}$
  is executed ≤ 1 times for a node
  $\Rightarrow$ h increases from the initial value 2
  atmost $n - 1$ times
  $\Rightarrow$ always $h \leq n + 1$

- v is got from array *vertex* by legal index i
  $\Rightarrow$ $1 \leq v \leq n$ on lines 5 and 6

- line 3 and on line 7 $h \neq l$
  $\Rightarrow$ $u = \mathit{sequence}[l]$ on line 10

Notices

- the invariant for abstract code was
  - easy to understand and natural
  - easy to prove to be an invariant
- termination of the abstract code was easy to prove
- that the end condition becomes true needed a little reasoning by induction based on the definition of path

  - "since v_0 is found and all neighbours of each found node are found, then all nodes that are reachable from v_0 are found"

- the concrete code was designed by making an array-based implementation for the operators used in the abstract code

  - the analogy between how the abstract and concrete data is expressed was expressed as three easy formulas

- the concrete code was proven to be correct by proving that

  - each concrete operation accomplishes the corresponding abstract operation (in other words, the operation preserves the analogue of code)
  - the concrete code guarantees that its own variables (e.g. l and h) have legal values when the value is used in the analogue

- the analogue between the abstract and concrete way of express is also a kind of invariant!

- in the proof there were many details, but all easy, and many of them were done by same reasoning

- use of the abstract code made the operational principle of the concrete code easy to understand

- proving the concrete code "directly" had needed coding of the analogue in to the loop invariant

  $\Rightarrow$ that had been confusing

- the concrete implementation can (at least in some limits) be replaced by another without that the proof of abstract code is necessary to touch

A problem

- depth-first-search does not find all the neighbours of same node consecutively

$\Rightarrow$ even the abstract code is quite universal, it cannot present the depht-first-search method

- the problem can be solved by making the abstract code a little more universal

- an exercise

5.2 Example of a Hard Proof of an Algorithm

In this subsection

- is given an efficient stack-based non-recursively implementation for depth-first-search
- is proven that it has the basic properties of depth-first-search
- is given and proven correct one application for it

Depth-first-subsearch

- goes through all nodes that are reachable from the first node
- each node is associated with two important point of time: being found and being finished
- the state of a node is expressed by colours
  - *white*: not yet found
  - *grey*: found but not yet finished
  - *black*: finished

(a) when a node is finished, its all neighbours are found or finished

- in other words

$$\forall u \in V : (\, u{\uparrow}.\mathit{colour} = \mathit{black} \;\Rightarrow\; \forall v \in \mathit{neighbours}(u) : v{\uparrow}.\mathit{colour} \neq \mathit{white}\,)$$

(b) grey nodes form a loopless path from the initial node v_0 to the node that is being processed

$$\begin{array}{ll} \exists n \in \mathbb{N} : \exists v_1, \ldots, v_n : \forall i, j; 0 \leq i < j \leq n : & \\ & v_i \neq v_j \\ \wedge & v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n \\ \wedge & \{v_0, v_1, \ldots, v_n\} = \left\{\, v \in V \;\middle|\; v{\uparrow}.\mathit{colour} = \mathit{grey}\,\right\} \end{array}$$

(c) the path and the colour of nodes can change only in two ways:

- white node v that is a continuation of the path changes to grey ($\Rightarrow$ the path becomes longer and v gets name v_{n+1})

$$v_n \rightarrow v \wedge v{\uparrow}.\mathit{colour} = \mathit{white}$$

- the last node v_n of the path changes to black (the path becomes shorter)

Algorithm (acknowledgements to I. Kokkarinen for finding the algorithm)

- the presentation of the graph and the initial node are as in previous

$\langle\ \forall v \in V : v\!\uparrow\!.\,colour = white\ \rangle$

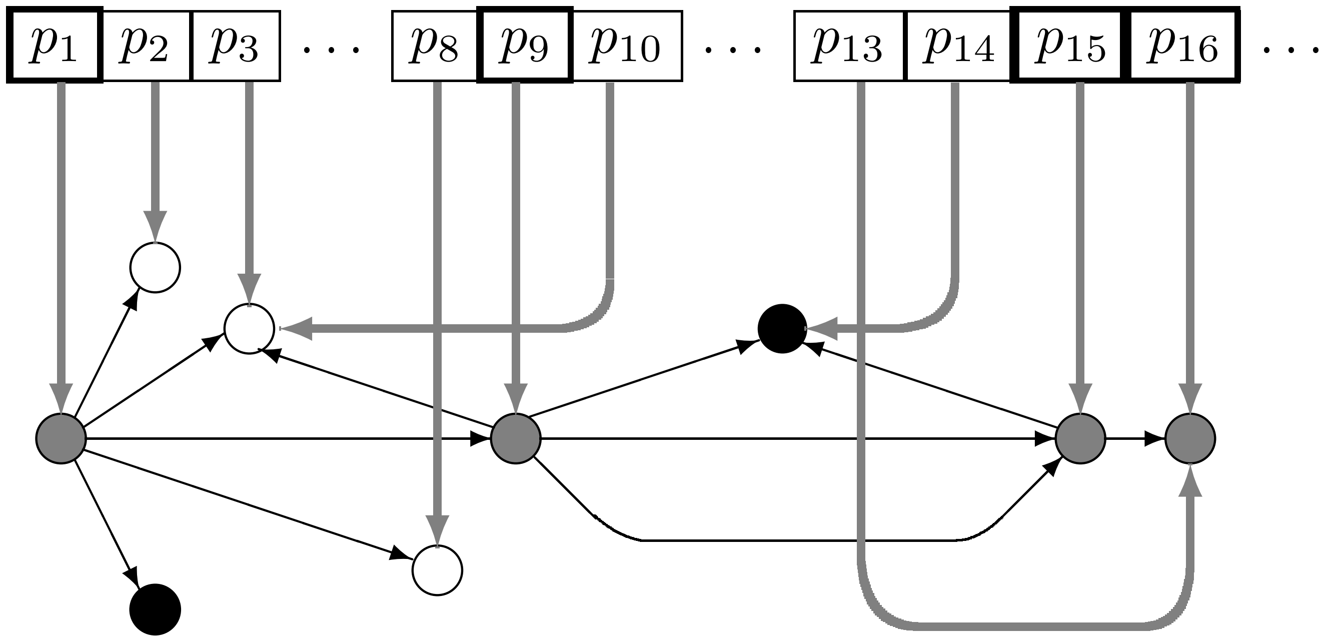
```
1  push(v0)
2  while ¬stackempty do
3      pop(u);
4      if u↑.colour = white then
5          u↑.colour := grey;  push(u);
6          forall v ∈ neighbours(u) do
7              if v↑.colour = white then
8                  push(v)
9              endif
10         endfor
11     else u↑.colour = grey then
12         u↑.colour := black
13     endif
14 endwhile
```

The operational principle informally

- in the stack there are pointers to nodes

- in the stack there can be many pointers to same node

- in the stack there can be pointers to nodes which are not yet "found" in the sence of depth-first-subsearch
  - the white nodes
- the correct finding- and finishing order is guaranteed by the colouring of nodes and handling of the stack
- a white node is found by popping from the stack a pointer to it
- when a node is found
  - it is coloured to grey
  - its still not yet unfounded neighbours are put to the stack above it to be found later
- if same node comes later back lower in the stack, it is black
  - ⇒ nothing is done for it
  - ⇒ only the top most occurrence of grey nodes is important
- in the stack the the top most occurrences of grey nodes define the depth-firts-path
- the other occurrences in stack are neighbours of the next lower of the top most grey node



- returning to the stack a just found node as grey and the later colouring to black are necessary for applications for depth-first-search

Notations

- n_S height of the stack
- $S[1], S[2], \ldots S[n_S]$ contents of the stack starting from the bottom
- $topmost(i) \Leftrightarrow \forall j \; ; \; i < j \leq n_S : S[j] \neq S[i]$
  - predicate which tells is the occurrence of a node in stack the top most occurrence of the node

- $GTM = \{\, i \mid topmost(i) \wedge S[i]\uparrow.colour = grey \,\}$
  - the set of the topmost occurencies of grey nodes
- Let
  - $G = \mid GTM \mid$ the number of grey nodes in the stack
  - $GTM = \{g_1, g_2, \ldots, g_G\}$ GTM in order

  $\Rightarrow$ $1 \leq g_1 < g_2 < \cdots g_G \leq n_S$

The invariant of the main loop in the algorithm consists on the following parts:

1. all grey nodes have a top most occurrence in the stack
   $\forall v \in V : (v\uparrow.colour = grey \Rightarrow \exists k \; ; \; 1 \leq k \leq G : v = S[g_G])$
   $\Leftrightarrow$ all grey nodes are in the stack since the stack is finite
2. the occurrences of nodes in the stack are neighbours of the top most of the next lower grey node
   - $\forall k \; ; \; 1 \leq k \leq G : \forall i \; ; \; g_k < i \leq g_{k+1} : (S[g_k] \rightarrow S[i])$
   - $G \geq 1 \Rightarrow \forall i \; ; \; g_G < i \leq n_S : (S[g_k] \rightarrow S[i])$

3. the white neighbours of grey nodes are in stack, and above the top most occurrence of the node in question

$$\forall k \ ; \ 1 \leq k \leq G : \forall v \in neighbours(S[g_k]) : \\ v\!\uparrow\!.\,colour \neq white \vee \exists i \ ; \ g_k < i \leq n_S : v = S[i]$$

4. the neighbours of black nodes are coloured

$$\forall u \in V : (\, u\!\uparrow\!.\,colour = black \Rightarrow \\ \forall v \in neighbours(u) : v\!\uparrow\!.\,colour \neq white \,)$$

5. if there is anything in the stack then the initial node is in the bottom of it

$$n_S \geq 1 \ \Rightarrow \ S[1] = v_0$$

6. if in the stack there are at least two nodes, then the initial node is grey and only in the bottom

$$n_S \geq 2 \ \Rightarrow \ g_1 = 1$$

Justifying the invariant

- when it is first come to line 2, in the stack there is only v_0, and all nodes (also v_0) are white

  $\Rightarrow$ $n_S = 1 \wedge G = 0$

  $\Rightarrow$ all parts of the invariant are holding

- item 1 is threatened only when
  - a node is coloured to grey
  - a grey node is removed from the stack
- when a node is coloured to grey, it is put to the stack
- when a grey node is removed from the stack, it is coloured to black

$\Rightarrow$ item 1 does not break

- the push operations on lines 5 and 8 preserve the item 2
- a node is changing to grey only on line 5
  - the top most occurrence of a grey node is all the time in the same position in the stack

$\Rightarrow$ item 2 does not break

- lines 5, . . . , 10 ensure that item 3 does not break when nodes are inserted in to the stack
- because of lines 4 and 5 a node that is removed from the stack is no longer white

$\Rightarrow$ item 3 does not break

- the colours on nodes can change only

  white $\rightarrow$ grey $\rightarrow$ black

- when a node is coloured to black, it was the top most grey in the stack

  $\Rightarrow$ because of item 3 it does not have white neighbours

$\Rightarrow$ item 4 does not break

- in the beginning the initial node is put to the bottom of the stack
- if nodes are inserted in to the stack on lines 5 and 8, it is first inserted the node that was just popped from the stack
- when there is nothing anymore in the stack, the algorithm terminates

$\Rightarrow$ item 5 does not break

- line 5 $\Rightarrow$ the node in the bottom changes to grey before new nodes come in to the stack
- item 5 $\Rightarrow$ if there is anything in the bottom, it is v_0
- lines 4 and 7 $\Rightarrow$ v_0 is not coming twice or more in to the stack

⇒ v_0 changes to black not until the stack becomes empty

⇒ item 6 does not break

Termination of the algorithm

- in every cycle either
  - the number of white nodes decreases, or
  - the number of white nodes remains the same and the height of the stack decreases

⇒ (n_w, n_S) is a bound function, where n_w is the number of white nodes

⇒ the algorithm terminates

Now it is possible to show the basic properties of depth-first-subsearch

- (a) follows directly form item 4 of the invariant
- the path $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n$ expressed by (b) is formed from the top most occurrences of grey nodes as following:
  - $n = G - 1$
  - $\forall k \ ; \ 2 \leq k \leq G : v_{k-1} = S[g_k]$
  - items 5 and 6 of the invariant ⇒ $v_0 = S[g_1]$

- g_G is determined by the top most occurrence of node $S[g_k]$

$$\Rightarrow \forall k, l \ ; \ 1 \leq k < l \leq G : \\ g_k < g_l \wedge S[g_k] \neq S[g_l]$$

$$\Rightarrow \forall i, j \ ; \ 0 \leq i < j \leq n : v_i \neq v_j$$

- item 2 of the invariant
  $\Rightarrow \ v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n$
- item 1 of the invariant and the definition of numbers g_k

$$\Rightarrow \{v_0, v_1, \ldots, v_n\} = \left\{ v \in V \;\middle|\; v\!\uparrow\!.\,\mathit{colour} = \mathit{grey} \right\}$$

Notices

- the operating principle is based on a huge set of detailed parts of the invariant
  - e.g.

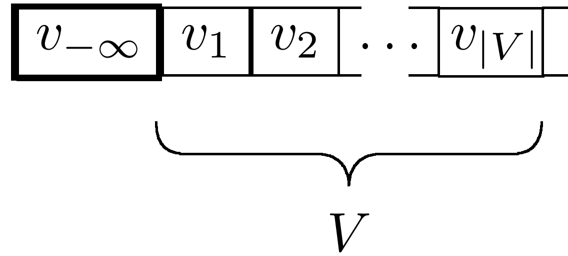
    "the other occurrences of nodes in stack are neighbours of the next lower of the top most grey node"
  - it is almots impossible to understand correctly without a formal formatting
- the proof simplified to a long sequence of checking the parts of the invariant and the properties of the depth-first path
- the single checkings are easy, but ...

- ... they are difficult to formalise properly
  - even in the algorithm books they are atmost at this level—unfortunately!
- the triple of algorithm, its invariants and proof is often like a house of cards, where everything affects to everything
  - if you fix one part then often another breaks
  - but designing algorithms is usually like this—it is not a fault of proof technique, if it reveals that the isuuse is difficult (or that the algorithm is broken)
- in the proofs and their checkings experience and caution is required

Full depht-first-search

- repeats depht-first-subsearch by new initial nodes until all nodes are black
- it could be done by one subsearch by adding in to the graph a new node $v_{-\infty}$ and edges $(v_{-\infty}, v)$ for all $v \in V$

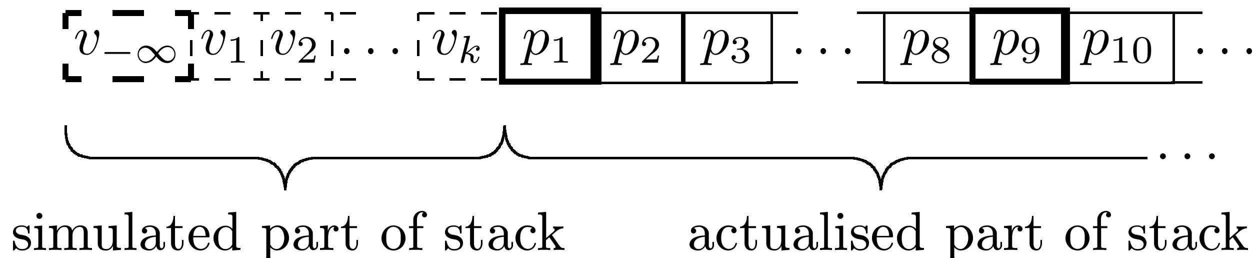
$\Rightarrow$ after the first cycle the stack would be



– unefficient since every node is in the stack

More efficient full depth-first-search

- the idea: let us simulate the lower part of the stack by putting nodes $v_1, v_2, \ldots, v_{|V|}$ to the stack of the implementation one by one not until they are needed



– the order for insertion is the investigation order of the starting edges of $v_{-\infty}$ backwards (why?)

– node $v_{-\infty}$ need not to be put in to the stack (why?)

⇒ algorithm

$\langle \forall v \in V : v{\uparrow}.\mathit{colour} = \mathit{white} \rangle$

0a **forall** $v \in V$ **do**
0b push(v)
2 **while** ¬stackempty **do**
... ... (* lines 2, ..., 14 remain *)
14 **endwhile**
15 **endfor**

- let
  - SA = simulated algorithm ($v_{-\infty}, v_1, v_2, \ldots,$ v_k are really in stack)
  - AA = actualised algorithm, where **forall** goes through the nodes in opposite order than where the first cycle of SA inserts them in to the stack
- description of simulation:

when the **forall**-loop has put the nodes

$v_{|V|}, v_{|V|-1}, \ldots, v_k, v_{k+1}$

in to the stack, then
the simulated part of the stack is

$v_{-\infty}, v_1, v_2, \ldots, v_k$

Proof of simulation

- we prove
  - on lines 2, . . . , 14 the state of AA included with the simulated part of the stack = the state of SA
  - AA does not imitate operations directed to $v_{-\infty}$
  - AA may execute line 2 some extra times when the stack is empty
  - otherwise executions of lines 2, . . . ,14 of AA imitate exactly and in same order the executions of lines 2, . . . , 14 of SA
- after 1. cycle of SA in the beginning of line 2
  - the stack of SA is $v_{-\infty}, v_1, v_2, \ldots, v_{|V|}$
  - $v_{-\infty}$ is grey
  - all the other nodes are white
- when AA first time enters line 2
  - only $v_{|V|}$ is in the stack of AA
  - all nodes are white

⇒ simulation holds

  - the simulated part of the stack is $v_{-\infty}, v_1, v_2, \ldots, v_{|V|-1}$

- if in the beginning of line 2 the stack of AA is not empty then
  - both SA and AA execute lines 2, ..., 14
  - both use only the part of AA of the stack

  $\Rightarrow$ the simulation holds

- if in the beginning of line 2 the stack of AA is empty but in the stack of SA there is also something else than $v_{-\infty}$, then
  - AA cycles the **forall**-loop only once
  - AA copies the top most element of SA in to its stack

  $\Rightarrow$ the simulation holds (the border between the simulated and the actualised goes one step to lower)

- if in the beginning of line 2 in the stack of SA there is only $v_{-\infty}$, then
  - SA pops $v_{-\infty}$ from the stack, colours it to black, and terminates
  - AA terminates

$\Rightarrow$ AA simulates SA, as was argued

$\Rightarrow$ AA finds and colours nodes in same ordering than SA, execept that AA does not handle the (hypothetical) node $v_{-\infty}$

Small optimisation

- if in the line 0b a balck node is put in to stack then lines 2, ..., 14 only remove it from the stack

⇒ it is not necesssary to put black nodes in stack on line 0b

- a node (except $v_{-\infty}$) changes to grey only if it has been in the stack of AA
  - it changes to grey before the stack of AA becomes empty

  ⇒ when the stack of AA is empty, only $v_{-\infty}$ can be grey

⇒ on line 0b it is sufficient to put only white nodes in to stack

⇒ the final algorithm

```
    ⟨ ∀v↑. colour = white ⟩
 1  forall v0 ∈ V
 2      if v0↑. colour = white then
 3          push(v0)
 4          while ¬stackempty do
 5              pop(u)
 6              if u↑. colour = white then
 7                  u↑. colour := grey; push(u)
 8                  forall v ∈ neighbours(u) do
 9                      if v↑. colour = white then
10                          push(v)
11                      endif
12                  endfor
13              elsif u↑. colour = grey then
14                  u↑. colour := black
15              endif
16          endwhile
17      endif
18  endfor
```

Application: recognition of cycles

- the task is to construct the set *Cyc* of nodes so that
  - each node of *Cyc* belongs to ≥ 1 cycles
  - ≥ 1 of the nodes of each cycle are in *Cyc*

- a cycle is a sequence of nodes $u_1, u_2, \ldots, u_n$ such that
  - $n \geq 1$
  - $u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_n$
  - $u_n \rightarrow u_1$
- node u belong to some cycle $\Leftrightarrow$ $\exists v \in V : u \rightarrow v \rightarrow^* u$
- proof
  - u belongs in to cycle $u_1, u_2, \ldots, u_n$

  $\Rightarrow$ $\exists i \ ; \ 1 \leq i \leq n : u = u_i$

  $\Rightarrow$ if $i < n$ then by choosing $v = u_{i+1}$ we get
  $$u \rightarrow v \rightarrow^* u_n \rightarrow u_1 \rightarrow u$$
  if $i = n$ then by choosing $v = u$ we get
  $$u = u_n \rightarrow v \rightarrow^* u_n = u$$

  - $\exists v \in V : u \rightarrow v \rightarrow^* u$

  $\Rightarrow$ $\exists n \ ; \ n \geq 1 : \exists u_1, u_2, \ldots, u_n :$
  $$n \geq 1 \wedge u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_n$$

  - $n \geq 1 \wedge u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_n \wedge u_n = u_1$

- the algorithm is got by full depth-first-search by small modifications

$\langle\ \forall v\!\uparrow\!.\,colour = white\ \rangle$

```
0    Cyc = ∅;
1    forall v0 ∈ V
2        if v0↑.colour = white then
3            push(v0)
4            while ¬stackempty do
5                pop(u)
6                if u↑.colour = white then
7                    u↑.colour := grey; push(u)
8                    forall v ∈ neighbours(u) do
9                        if v↑.colour = white then
10                           push(v)
10b                      elsif v↑.colour = grey then
10c                          Cyc := Cyc ∪ {u}
11                       endif
12                   endfor
13               elsif u↑.colour = grey then
14                   u↑.colour := black
15               endif
16           endwhile
17       endif
18   endfor
```

$$\langle\ (\forall u \in Cyc : \exists v \in V : u \rightarrow v \rightarrow^{*} u\,) \\ \wedge (\,\forall u_1, \ldots, u_n\ ; \\ n \geq 1 \wedge u_1 \rightarrow \cdots \rightarrow u_n \wedge u_n \rightarrow u_1 : \\ \exists i\ ;\ 1 \leq i \leq n : u_i \in Cyc\,)\ \rangle$$

Does the algorithm find the correct nodes?

- on line 10c it holds that
  - $u \rightarrow v$ because of line 8
  - $v \rightarrow^* u$ because of the properties (b) of depth-first-search

  $\Rightarrow$ to Cyc it is added only nodes that are in cycles

  $\Rightarrow$ in the end $\forall u \in Cyc : \exists v \in V : u \rightarrow v \rightarrow^* u$
- does the algorithm find at least one node from every cycle?
- let $u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_n$ be cycle
- let $1 \leq k \leq n$ be chosen so that u_k is the first node to be coloured black of nodes $u_1, u_2, \ldots, u_n$
- when $u_k$is coloured to black then $u_{k \bmod n+1}$ is not
  - white, because of depth-first (a)
  - black, because u_k is **the first** to be coloured to black

  $\Rightarrow$ $u_{k \bmod n+1}$ is grey
- depth-first (c)

$\Rightarrow$ $u_{k \bmod n+1}$ was lower in the stack than u_k or $= u_k$

$\Rightarrow$ $u_{k \bmod n+1}$ was grey when u_k was found

$\Rightarrow$ line 10c added u_k to Cyc

$\Rightarrow$ in the end

$$\forall u_1, \ldots, u_n \ ; \ n \geq 1 \wedge u_1 \rightarrow \cdots \rightarrow u_n \wedge u_n = u_1 : \exists i \ ; \ 1 \leq i \leq n : u_i \in Cyc$$

Termination

- the inserted lines do not affect the variables of the original program

  $\Rightarrow$ "invisible" from the point of view of execution of the original program

- the inserted lines terminate

$\Rightarrow$ the program terminates

6 Summary

The formal definition of a notion X tells exactly what object X is and what not

- also the borderline cases
- it is a different case how easy it is to see how the borders are specified
- it is a different case are the borders reasonable

Formal methods are usually bases on discrete mathematics

- especially notions of set theory and predicate logic of 1. degree are needed a lot
- the calculations needed for definitions and proofs become often long and laborious

  $\Rightarrow$ prone to errors

- definitions and proofs need often versatile datat types and arguments for reasoning

  $\Rightarrow$ difficult to automate

$\Rightarrow$ often you can get the best benefit from the method in a reasonable effort by following it informally, and processing formally only the most difficult and the most important parts

The benefit and limitations of prooving correctness

- eventually a programmer etc. wants an answer to the question

  "does this program work as I want?"

⇒ the functioning of the program must be compared to wishes of humans

  - *validation (kelpoistaminen eli validointi)*
  - inevitably informal

- means for validation: testing, reviewing, ordering the program from a reputable vendor,...
- the target of validation is

  to get a strong confidence that the program works as the validator wants

- programs would not need test at all if validation would succed in other means
- the final program may be very large, complex, and hard to understand

  ⇒ its validation can be a hopeless task

- the informal part of validation can be reduced in the following way:

- the requirements document of the program is made as easy to understand as possible
- prove that the program fulsills its requirements, that is, the program is *verified (verifioidaan)*
- the requirements documents is validated

- if te requirements document is easy to understand and the verification can be automatised, the human part in validation becomes crucially easier
- even the verification remains hand-craft, this kind of validitation can still give more reliable results than direct validation
- still always must be remembered that
  - in verification there can happen failures in calculations (was it done by human or machine)
  - the notion "fulfills the requirements" can be incorrectly specified (especially in concurrent systems)
  - the requirements documentation can be imcomplete even that nobody notices it
  - the reuirements documentation can be incorrect

⇒ *verification can never prove that a program is without errors*

- even in its best verification can only prove the program *fulfills* (whatever it may mean) *its formal requirement documentation* (whatever the requirements stated by it actually are)

Other means to use formal methods as an instrument of validation

- writing the specification and other documents formally
  - enforces to think the matter more rigorously
  - decreases risk for ambiguous
  - enforces to handle also the special cases

  ⇒ imprves the correspondence of implementaion and wishes
- verification againts incomplete or even very trivial requirements documentation can be used as a means of validation among other means
  - e.g. it is proved that the program terminates
  - e.g. it is proved that no overflows happen
- formal analysis can be used in same way

  - e.g. the data flows of the program are calculated and of which inputs a specified results depends is printed

- deriving the program formally from its requirements documentation is – from the point of view of validation – as good as proving that the program fullsfills its requirements documentation
  - direction of progress is in a way naclward
  - in practice deriving and proving often complete each other

Where formal methods gives the best benefit?

- when extremely great reliability is needed
  - safety-critical systems
- when an error can become exceptional expensive
  - e.g. protocols for telephone networks
  - e.g. universal program libraries
- when a system has to be made working as less prototype cycles as possible
  - e.g. designing silicon chips

- when the implementations are extremely difficult to understand compared to their specifications
    - e.g. programs for numerical analysis
    - reactive systems
    - complicated algorithms

⇒ the research of formal methods in Department of Software Systems is concentrated on reactive systems

About formal methods for reactive systems

- essentially different than the methods in this course but they utilise the same ideas
- several different theories and methods based on them
    - in the department there are studied action systems, temporary logic, process algebra, and in some sence Petri Nets

⇒ courses

- OHJ-2606 State Machines 5 cru