MATHEMATICAL THEORY OF COMPUTATION.

Ref: Kfoury, Moll & Arbib: "A Programming Approach to Computability", Springer - Verlag.

① INTRODUCTION.

Computability theory examines the functions computed by algorithms. We regard a program as a function mapping a set of input data onto a set of output data.

We say that a function is computable if it is determined by some algorithm (ie, some well specified set of rules) We first investigate the relationship between algorithms and functions

② FUNCTIONS

$f: A \to B$ is a rule (function) according to which each elt $x \in A$ is associated with some elt $f(x) \in B$. We call A the domain of arguments, and B the co-domain of values.

The rule may be a precise algorithm, or it may be a hypothetical rule of correspondence, eg:

$$f(x) = \begin{cases} 1 \text{ if Fermat's last theorem is true} \\ 0 \text{ otherwise.} \end{cases}$$

The status of f is not questionable - it is a constant 0 or 1, although we do not know if F.L.T. is true.

All programmers should be familiar with the phenomenon of the non-terminating algorithm, such as:

③ while $x \bmod 2 = 0$ do $x := x + 2$

This is a function:

$$\theta(x) = \begin{cases} x & \text{if } x \text{ is odd} \\ \perp & \text{undefined otherwise} \end{cases}$$

This phenomenon of divergent computations is a fundamental feature of computability theory. θ is called a partial function.

DEFN: A partial function is a function which may or may not be defined on all its arguments.

Whereas an ordinary function $f: A \to B$ assigns a value $g(x) \in B \quad \forall x \in A$, a partial function $\theta: A \to B$ assigns a value $\theta(x)$ only to some $x \in \text{DOM}(\theta) \subseteq A$.

④ $\text{DOM}(\theta)$ is called the domain of definition of θ. If $x \notin \text{DOM}(\theta)$ we say that θ is undefined at x.

The set $\{x \mid x \in \text{DOM}(\theta)\}$ is called the range of θ, denoted $\text{RAN}(\theta)$.

We define a special partial function:

$$\perp(x) := \perp, \quad \forall x \in \mathbb{N}$$

called the empty function.

DEFN: A total function is one where $\text{DOM}(f) = A$ for $f: A \to B$

⑤ Unless stated otherwise, all functions will be of the form $\theta : \mathbb{N}^K \to \mathbb{N}^L$

ALGORITHMS.

Some conditions for the existence of an algorithm:

- an algorithm works on input data to produce output data.
- for any algorithm, there is a restricted set of valid input data for which an execution of the algorithm is meaningful.
- the input data must have a finite description (since the algorithm starts with a finite object and terminates (if it does) after a finite time, the output data also are finite)
  ⇒ the output data must have a finite description.
- an algorithm must always yield the same result when started with the same input data. In this way, an algorithm establishes a (partial) functional relationship between the sets of possible input and output data

⑥ How do we specify algorithms? There have been a number of attempts to provide a "most general" definition, these include:

- Turing machines
- the Kleene characterisation
- "program description"
- Markov algorithms
- Lambda calculus

etc.

For an algorithm specification method to be "most general" it must be <u>complete</u> in the sense that:

> "Every conceivable algorithm, regardless of how it is expressed, must be equivalent to some algorithm written in the specification methodology"

We say two algorithms are equivalent if they compute the same function. We have no way of proving completeness, but we do know that any algorithm that can be expressed in one form can be expressed in any of the others, and that they are as complete as any known specification method.

Any specification method has the following features in common:

- an algorithmic process is the process of applying an algorithm to appropriate input data, and can be decomposed into successive elementary steps
- each elementary step consists of replacing one state of the process by another, in a deterministic, mechanical manner, according to elementary rules of transformation
- the outcome of an algorithmic process either:
  (i) replacing one state by the next without termination
  <u>or</u> (ii) reaching a state for which there is no next state according to the transformation rules

(8)

- Only in the second case can we obtain output data from the algorithmic process.

CHURCH'S THESIS: The Turing and programming language specification methods both are complete

We said earlier that there is no way of proving this thesis; there is also no cause to disbelieve it, as no algorithm which cannot be defined has ever been found. Church's thesis is most useful, as we will see.

AN INFORMAL LOOK AT COMPUTABILITY THEORY

We will show that it is impossible for an algorithm to exist which can decide if any program halts for a certain input.

(9)

Assume a list of all valid programs:

$$P_0, P_1, P_2, \ldots, P_n, \ldots$$

Now any program can be represented by a natural number obtained from a mapping $\varphi: N \to N$, N defined as follows:

> "Using Ascii notation translate each character of the program to binary. The resulting (integer value of the) binary number is the index N of the program."

The translation φ is one-one, as is its inverse.

Some N will point to illegal programs (eg P_4) - we consider all these to be $\perp(N)$ (undefined).

Obviously, every possible program is represented, and the relationship φ is clear.

(10) How an arbitrary P_x halt when supplied its own index as input? We will see that the total function

$$F(x) = \begin{cases} 1 & \text{if } P_x \text{ halts on input } x \\ 0 & \text{if } P_x \text{ fails to halt on input } x \end{cases}$$

is not computable - a special case of the halting problem. We proceed by contradiction:

(1) suppose F is computable by some program HALT(n). If P_n halts on input n, HALT(n) returns 1, otherwise 0.

(2) Now suppose we construct a program CONFUSE which models the partial function

$$\varphi(n) = \begin{cases} 1 & \text{if HALT}(n) = 0 \\ \perp & \text{otherwise (infinite loop)} \end{cases}$$

(3) Suppose CONFUSE has the index e. What is $\varphi(e)$?

$$\varphi(e) = \begin{cases} 1 & \text{if halt}(e) = 0, \text{ ie } P_e \text{ does not halt on } e \\ \perp & \text{if halt}(e) = 1, \text{ ie } P_e \text{ does halt on } e \end{cases}$$

This is a contradiction. It is easy to write a short confuse program, so halt cannot exist.

Diagonalisation and the Halting Problem.

Recall that for a total function $f: A \to B$ is

- injective if all $x, y \in A : x \neq y \Rightarrow f(x) \neq f(y)$
- surjective if f maps onto all B i.e. $B = \text{range}(f)$.
- bijective if both injective & surjective.
- A is larger than B if $\exists$ an injection $f: B \to A$.
- If there are injections $A \to B$ and $B \to A$ then $\exists$ a bijection between A and B (i.e. cardinality the same).

Theorem: There is no bijection $h : \mathbb{N} \to \mathbb{R}$.

Proof: By contradiction. We construct a list of all the reals $h(0), h(1), \ldots, h(n), \ldots$ without repetitions. We can construct an r not in the list as follows

(i) r begins with a decimal point

(ii) The k^{th} digit to the right of the decimal point is a five, unless the k^{th} digit of $h(k)$ is a five, in which case we use some other digit.

(iii) r will then differ from each real by at least one decimal place.

Cantor's argument can be used to find non-computable functions. If we use a diagonal as follows.

Input → / Program ↓	0	1	2	3	...
P_0	1	⊥	⊥	⊥	...
P_1	⊥	1	⊥	1	...
P_2	1	1	⊥	1	...
P_3	1	⊥	⊥	1	...
⋮	⋮	⋮	⋮	⋮	

(16) Our program comparer would have ⊥⊥|⊥.... different to every P_n input n.

The unsolvability of the halting problem means that such a table is uncomputable..

---

(18) PROPOSITION: It is sufficient in computation theory work to consider natural number programs only - we suffer no loss of generality. We can justify this in that any countably infinite set is isomorphic to $\mathbb{N}$.

We can now evolve a programming language formalism. Our axioms are:

$0 : \to \mathbb{N}$ the zero function

$succ : \mathbb{N} \to \mathbb{N}$ the successor function

We can immediately construct the inverse of succ,

$pred : \mathbb{N} \to \mathbb{N}$

with definition: $\begin{cases} pred(0) = 0 \\ pred(succ(n)) = n \end{cases}$

(20) 0, succ and pred allow us to define numbers and arithmetic operations. We now introduce variables and the assignment statement.

variable := value-expression

We must also be able to compare values. We thus introduce tests. A <u>test</u> is a function which returns 0 or 1, also denoted <u>true</u> and <u>false</u>.

(21) The basic test checks equality. $= : \mathbb{N}^2 \to \{0,1\}$

semantics $0 = 0$

$x = y$ iff $pred(x) = pred(y)$

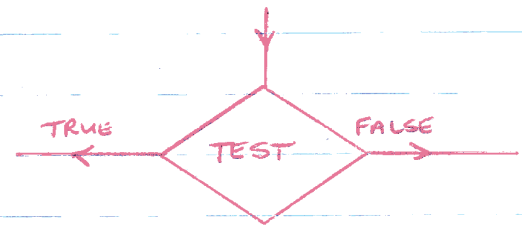
We can combine tests using and ($\wedge$) or ($\vee$) and not ($\neg$)
The second test is less than:

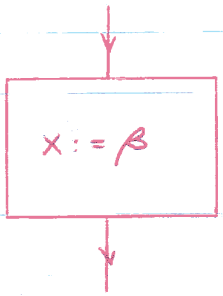
$< : \mathbb{N}^2 \to \{0,1\}$.

semantics $<(x,y) :=$ if $x = 0 \wedge y = 0$ then 0
else if $x = 0 \wedge \neg(y = 0)$ then 1
else if $\neg(x = 0) \wedge y = 0$ then 0
else $<(pred(x), pred(y))$

<u>Exercise</u> define $>$, $>=$, $<=$, $<>$ from the above set.

(22) We represent a test as: and an assignment as

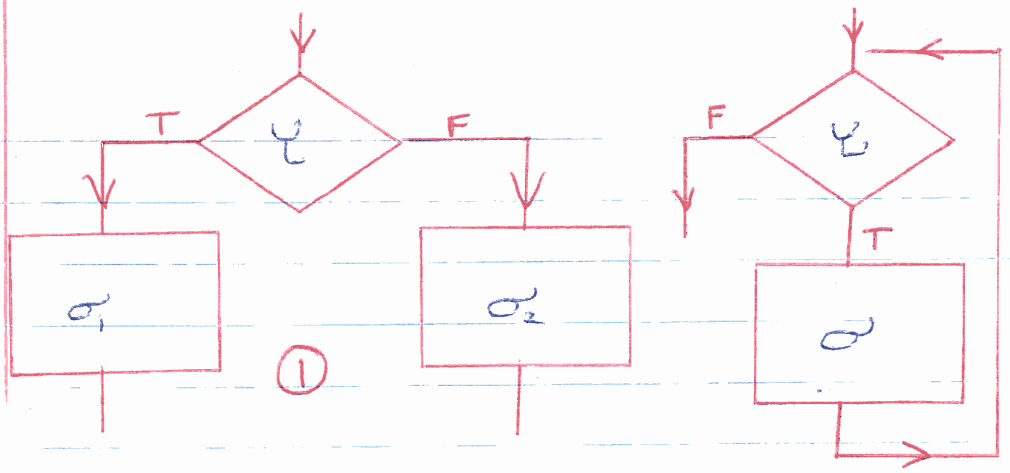




Where β is one of 0, succ(y), pred(y). An assignment is the most simple form of statement in our language.

We can combine tests and assignments together to form a statement with a conditional part. We can have different forms, eg:

(10)



(23) These structures are so useful that we provide syntax to represent them.

① if γ then σ_1 else σ_2

② while γ do σ

We can define compound statements:

begin σ_1 ; σ_2 ; ... ; σ_m end

and a null statement

Exercise Draw compound statement and null statement graphs

– Define repeat and for statements

DEFN A program is a compound statement.

(24) We have introduced high level control abstraction into our language, but we can perform only elementary operations. We turn now to defining arithmetic operations.

PROPOSITION: The statements:

(i) $x := y$

(ii) $x := n$, $n \in \mathbb{N}$

are valid in our language.

Proof: (i) We can get this by $x := succ^y(0)$

(ii) By analogy to (i)

(25) PROPOSITION: The statements

(i) $z := x + y$

(ii) $z := x - y$

(iii) $z := x * y$

(iv) $z := x \text{ mod } y$

(v) $z := x \text{ div } y$

(vi) $z := x ** y$

2 PROOF: (i) Equivalent to:

```
begin
    u := 0;
    z := x;
    while u <> y do
        begin
            z := succ(z);
            u := succ(u);
        end;
end;
```

(2) $\dot{-}$ is a monus operation - if $a > b$ then $a - b$ else 0.

a program for this is:

(2i)

```
begin
    z := x;
    u := y;
    while u <> 0 do
        begin
            z := pred(z);
            u := pred(u);
        end
end.
```

Note that these don't leave the operands unchanged.

(5) z := x div y defn by

```
begin
    z := 0;
    u := x;
    while u <> 0 do
        begin
            u := u - y;
            z := succ(z);
        end;
end;
```

Exercise: define (3), (4), (6)

(27) Any program can be given a name and called by giving the name in another program. Recursion is not allowed. Note that a program, being only a compound statement, may only be called by a call in a valid statement place.

COMPUTABLE FUNCTIONS.

Any statement σ (or program p) uses k variables, where $k \geq 0$. We call this a k-variable or k-ary statement, which means that σ uses at most k variables.

We will restrict our variable names to $x_1, x_2, x_3, \ldots, x_n, \ldots$ Then we can say that a k-ary statement uses a subset of $\{x_1, x_2, \ldots, x_k\}$.

DEFN: A state vector of a k-ary statement is a k-vector $(a_1, a_2, \ldots, a_k)$ where a_i is the value at some specific point in time of x_i.

29

For example, consider the program

```
begin
    x2 := 0
    while x2 ≠ x1 DO
        x2 := succ(x2)
    OD  (end do*)
end.
```

This computes $x_2 := x_1$.
We can describe the course of execution with initial data (2, 4) as:

(2,4)
$x_2 := 0$
(2,0)
$x_2 \neq x_1$?
(2,0)
$x_2 := succ(x_2)$
(2,1)
$x_2 \neq x_1$?
(2,1)
$x_2 := succ(x_2)$
(2,2)
$x_2 \neq x_1$?
(2,2)
end
(2,2)

30

Each state vector can be thought of as occurring at a particular location in the program. Given an SV. we can derive the next SV_2 by applying the intermediate instruct. to SV.

We generalise:

Defn. Let p be a k-variable program. A computation by p is a sequence

$$a_0 A_1 a_1 A_2 a_2 \ldots a_i A_{i+1} a_{i+1} \ldots$$

where the a's are k-vectors ($\leq$ V's) and the A's are instructions appearing in p which satisfies the consistency conditions below.

If the computation is finite it has the form $a_0 A_1 a_1 \ldots A_n a_n$ where n is called the length of the computation

(31)

The consistency conditions for a computation by p are:

(i) A_1 is the first instruction in p. If the path through p represented by the sequence of A's is finite then the sequence of A's has an upper limit A_n, $n \geq 0$ such that A_n is the last instruction executed before termination of p.

(ii) The first term a_0 in the sequence of a's is an arbitrary k-tuple of natural numbers.

$\forall i : i \geq 1$:
- If A_i is a test then $a_i = a_{i-1}$
- If A_i is an assignment $X_v = g(\bar{x})$ and $a_{i-1} = (u_0, u_1, \ldots)$ then $a_i = (u_0, u_1, \ldots, u_v = g(\bar{u}) \ldots)$

(32)

(iii)
- If A_i is an assignment statement then A_{i+1} is the next instruction in p after A_i if such an instruction exists, else the sequence A_n is finite with A_i the last term
- If A_i is a test which returns true, then the true "arm" is followed to find the next A_i, else the false "arm"

is followed. In both cases if no such subsequent A exists then the sequence is finite with A_i as the last member.

PROPOSITION: Let ρ be a k-variable program, and a_0 an arbitrary k-vector of N. Then there is exactly one computation by ρ which has a_0 as its initial (input) state vector.

DEFINITION: If the computation by ρ which starts with a_0 has an infinite sequence of A's we say that ρ is divergent on input a_0 and that its output is undefined. If the sequence of A's is finite then we say the computation converges on input a_0 or terminates on input a_0, with output a_n where n is the length of the computation.

(33)

Given a program ρ, we wish to interpret ρ as an agent for computing a j-ary function $\varphi_\rho : N^j \to N$ using the machinery developed above. We suppose ρ is a k-variable (compound) statement.

DEFN: The j-ary semantic function for a k-variable program ρ, $\varphi_\rho : N^j \to N$, is defined as follows:

Given input vector $(a_1, \ldots, a_j)$, $\varphi_\rho(a_1, \ldots, a_j)$ is evaluated according to the following rules, with two possible cases:

CASE I: $k \geq j$

$\varphi_\rho(a_1, \ldots, a_j)$ is evaluated by applying ρ to the initial state vector $(a_1, \ldots, a_j, \underbrace{0, \ldots, 0}_{k-j \text{ 0's}})$

If and when P halts on this state vector, the value of $\varphi_P(a_1, \ldots, a_j) = b$, where b = the value of x_1^* on termination of P

(34) CASE II: $k < j$ Here we compute $\varphi_P(a_1, \ldots, a_j)$ by applying P to the initial state vector $(a_1, \ldots, a_k)$ ignoring the last $(j-k)$ arguments. If and when P terminates, the value of $\varphi_P(a_1, \ldots, a_k) = b$ where b = the value upon termination of x_1.

In either case, if P fails to halt on the initial state vector, then we say that $\varphi_P(a_1, \ldots, a_j) = \perp$

See example

DEFN: A function $\psi : N^j \to N$ is effectively computable if $\psi = \varphi_P$ for some program P. If we need to stress the number of input variables we write $\varphi_P^{(j)} : N^j \to N$

(36) Any partial function $N \to N$ which can be algorithmically defined is also computed by some program in our formalism. Church's thesis means that if we can establish that something is algorithmic, then there exists a program to compute it* and we need not find the program explicitly.

(38) EFFECTIVE ENUMERATION OF PROGRAMS.

To develop an enumeration of the programs which could be written in our language, we once again need a code with the properties:

* We assume the result is always returned in x_1

- any program can be coded to a natural number and no two programs code to the same number
- given a number, the program which it codes can be "factored out"

Such a numbering system is called a GÖDEL NUMBERING, or ARITHMETISATION OF SYNTAX.

(40) Any program can be reduced to one using only the basic symbols of our language. We can equate each symbol with a binary number — 6 bits are more than adequate. A variable name such as X13 is composed of three symbols X, 1 and 3. Our enumeration of all programs is obtained from

$P_0, P_1, P_2, \ldots, P_n, \ldots$

To find P_n: translate n to binary, split into 6 bit groups and map the 6-bit number onto symbols. If the resulting program is syntactically incorrect, then the P_n is considered to be an instance of the empty function.

(42) To find n given P: reverse of above process. Our programs can now be effectively enumerated, ie can be obtained simply from 0 and the succ function.

We can associate with each P_n a function:

$$\varphi_{P_n}^{(k)} : N^k \to N$$

which is the function computed by P_n. We can derive an effective enumeration of the partial functions as well, and

relate each partial function to some P_n.

(43) THEOREM - THE UNSOLVEABILITY OF THE HALTING PROBLEM

There is no TOTAL computable function f such that:

$$f(P_n) = \begin{cases} 1 & \text{if } \varphi_{P_n}(P_n) \downarrow \quad (\text{i.e. halts}) \\ 0 & \text{if } \varphi_{P_n}(P_n) \uparrow \quad (\text{i.e. does not halt}) \end{cases}$$

PROOF: Suppose f is computable. Consider the function:

$$\psi \quad N \to N \qquad \psi(P_n) = \begin{cases} \perp & \text{if } f(P_n) = 1 \quad (\text{i.e. } P_n \text{ halts}) \\ 1 & \text{otherwise} \end{cases}$$

A program to compute ψ is easy

```
begin
    while halt(x1) = 1  do x1 = x1 od;
    x1 = 1
end
```

Let this program have index e. Then $\varphi_e(e)$ has contradictory behaviour, so f is not computable (as a total function)

See examples

(47) UNDECIDABLE PROBLEMS.

Until now, our functions have been two value functions onto $\{0,1\}$ and $\{1,\perp\}$. We will call these yes/no problems

A yes/no problem is DECIDABLE if $\exists$ an algorithm which converges with the correct yes/no answer. (The halting problem is such a yes/no function, but is not decidable)

PROPOSITION: There exists a total function $f: N \to N$ which is not computable

PROOF: Produce an enumeration of total functions:

$f_0, f_1, \ldots f_i, \ldots$

Define F by:

$$F(i) = f_i(i) + 1$$

clearly F is total. If F was computable, it would be in our list, say as f_j.

But then:

$$F_j(j) = f_j(j) + 1$$ which is impossible

$\Rightarrow$ F cannot be computable (or total)

(49) SOME STATED RESULTS

(i) There is no algorithm which, when presented with an arbitrary computable function $\varphi: N \to N$, can decide whether φ is total or not.

(ii) GENERAL UNSOLVEABILITY OF THE HALTING PROBLEM

There is no algorithm to decide if $\varphi_i(a)$, any i, a, is defined or not.

(iii) UNSOLVEABILITY OF THE EQUIVALENCE PROBLEM

Given two functions with indices i and j, there is no algorithm to determine whether they compute the same function

(20)

(50) # PROGRAM VERIFICATION.

(52) We now turn to verification ("proving programs correct").
There are two approaches the classical and temporal logic techniques.

The two issues we are concerned with are correctness and termination. In classical verification these are proved seperately, but more modern methods can prove both simultaneously

(53) ## CLASSICAL VERIFICATION.

There are many methods - we will consider the inductive assertions method. We consider a program as a function which accepts as input a certain machine state with associated environment, and outputs the final state and environment.

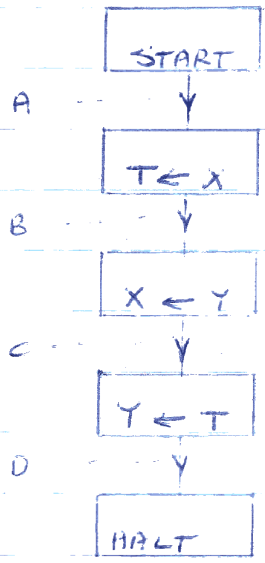
IF the compute language is well defined
AND the action of each statement is well defined in terms of the effect on the internal states of the machine
(54) THEN a computer can mechanically simulate the actions of a program by generating successive machine states and comparing the end results with the desired end state.

When working with the inductive assertions technique, we will always start with the last state and work backwards, because:-
- we regard computing as being goal orientated

- if the initial state satisfies the conditions we determine, then the final state will be true.
- we prove our input condition is the weakest precondition for the program to be correct

We specify our conditions in propositional calculus

(56) A simple example: Program to interchange two numbers



INPUT ASSERTION $X = X_0 \wedge Y = Y_0$

OUTPUT ASSERTION: $Y = X_0 \wedge X = Y_0$.

Working backwards D - $Y = X_0 \wedge X = Y_0$ (output assertion)

C - $T = X_0 \wedge X = Y_0$

B - $T = X_0 \wedge Y = Y_0$

A - $X = X_0 \wedge Y = Y_0$ (input assertion)

(55) At each point in the program where we encounter an assertion, we construct a compound proposition of the form:

assertion $\Rightarrow$ "ongoing assertion"

Where "ongoing assertion" is the assertion derived by starting with the output assertion and working backwards. In our example, we encounter only one such situation, when we reach A and meet our input assertion.

Here we have: assertion $X = X_0 \wedge Y = Y_0$ (input assertion)

ongoing assertion: $X = X_0 \wedge Y = Y_0$ (derived by us)

which gives: $X = X_0 \wedge Y = Y_0 \Rightarrow X = X_0 \wedge Y = Y_0$.

These new compound assertions are called verification conditions. To prove the program correct, we prove the verification conditions. In our example the verification condition is a tautology, so our program is correct (although we have not proven termination)

TERMINOLOGY: correctness ≡ partial correctness

total correctness ≡ partial correctness & termination.

(60) FORMALISATION:

We will use a type of program known as a while program. This has three sorts of variable grouped in vectors:

(a) input vector $\bar{x} = (x_1, x_2, \ldots, x_a)$ which never changes

(b) program vector $\bar{y} = (y_1, y_2, \ldots, y_b)$ consisting of the program/temporary variables which may change during execution

(c) output vector $\bar{z} = (z_1, z_2, \ldots, z_c)$ which yield output values when program execution terminates

We thus have three domains $D_{\bar{x}}$, $D_{\bar{y}}$, $D_{\bar{z}}$. For convenience, we will consider only integer values and variables. Note that we are concerned with run-time logic correctness, not syntactical or context-sensitive correctness.

A WHILE program is a sequence of statements B_i separated by semicolons $B_0; B_1; \ldots; B_n$.

B_0 is the unique start statement:

begin
$g \leftarrow f(\bar{x})$

where $f: D_{\bar{x}} \to D_{\bar{y}}$ is a total function.

There are 5 statement types:

- assignment

$\bar{y} = g(\bar{x}, \bar{y})$, $g: D_{\bar{x}} \times D_{\bar{y}} \to D_{\bar{y}}$ is a total fn

(62) - conditional

IF $t(\bar{x}, \bar{y})$ THEN B ELSE B'

IF $t(\bar{x}, \bar{y})$ THEN B

B, B' any statements, $t(\bar{x}, \bar{y}): D_{\bar{x}} \times D_{\bar{y}} \to \{0,1\}$ is a total predicate

- while statement

WHILE $t(\bar{x}, \bar{y})$ DO B

B, $t(\bar{x}, \bar{y})$ as above.

- compound statement

BEGIN
$B_1'; B_2'; B_3'; \ldots; B_k'$
END

where B_j' $1 \leq j \leq k$ are statements.

- halt statement

$\bar{z} \leftarrow h(\bar{x}, \bar{y})$
END

$h: D_{\bar{x}} \times D_{\bar{y}} \to D_{\bar{z}}$ is a total fn.

(24)

We now need to define the effect of each statement (trivial)
Given a while program P, and an input value $\bar{a} \in D_{\bar{x}}$ for $\bar{x}$, the program can be executed. Execution always begins at the start statement, initialising the value of $\bar{y}$ to $f(\bar{a})$ and then it proceeds through the sequence of statements

Whenever an assignment is reached, $\bar{y}$ is replaced by the current value of $g(\bar{x}, \bar{y})$. The execution of conditional and while statements can be represented by flowcharts
(64) (omitted).

If execution reaches a halt statement, then $\bar{z}$ is assigned the current value of $h(\bar{x}, \bar{y})$; say this value is $\bar{b}$. We then say that $P(\bar{a})$ is defined and $P(\bar{a}) = \bar{b}$. Otherwise, if execution never terminates, we say $P(\bar{a})$ is undefined. P thus represents a partial function mapping $D_{\bar{x}} \Rightarrow D_{\bar{z}}$

(68) Now, each program must have an input assertion $\varphi(\bar{x})$ which describes, bod, a condition which must be true at the start of the program, and also those $\bar{x} \in D_{\bar{x}}$ which may be used as input; and an output assertion $\psi(\bar{x}, \bar{z})$ which describes the relationship that must be satisfied between $\bar{x}$ and $\bar{z}$ at completion of program execution. Both φ and ψ are total predicates.

We can now say that:

- A program P terminates over φ if for every input $\bar{a}$ such that $\varphi(\bar{a})$ is true, the computation of the program terminates

- A program P is partially correct with respect to φ and ψ if, for every $\bar{a}$ such that $\varphi(\bar{a})$ is true, if the program terminates, then $\psi(\bar{a}, P(\bar{a}))$ is true

- A program P is totally correct w.r.t. φ and ψ if, for every $\bar{a}$ such that $\varphi(\bar{a})$ is true, the program terminates and $\psi(\bar{a}, P(\bar{a}))$ is true.

PATHS.

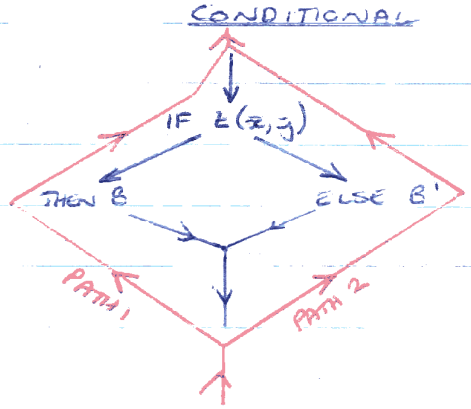
When we verify a program, we prove partial correctness by starting with the output assertion and working backwards through the program. Each route through the program is called a path.

Types of Paths

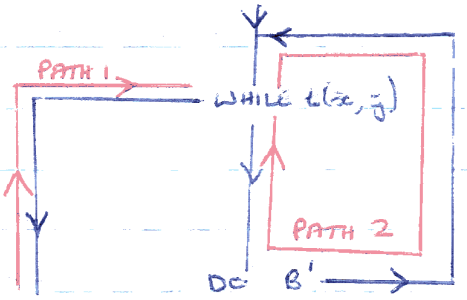
(70)

SEQUENTIAL

CONDITIONAL



ITERATIVE



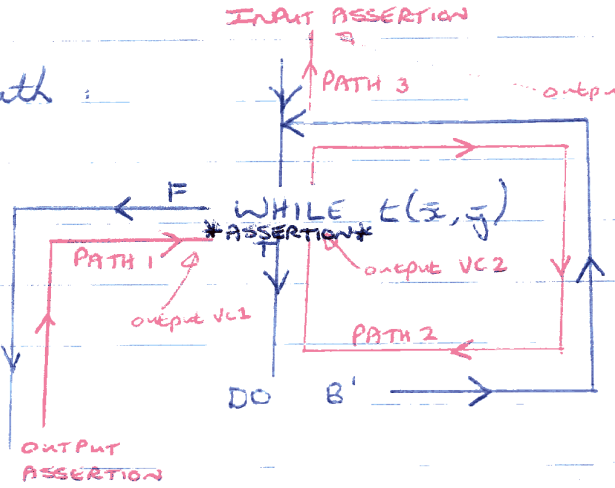
(72)

Path Verification

Similarly to our input and output assertions, we can supply a set of predicates P_i, each of which characterises the relationship between the variable at point i. Then $P_i(\bar{x}, \bar{y})$ must be true for the current values of $\bar{x}$ and $\bar{y}$ at i

We call these predicates INDUCTIVE ASSERTIONS. They are needed at the beginning and end of each path at least, although the predicate at the end of one path could be the predicate at the start of another path, etc. φ and ψ are thus special cases of inductive assertions. At each point where we encounter an inductive assertion, we output a verification condition which must then be checked by a theorem prover. Note that an iterative path must be finite, else no ver. con. will be generated

Eg iterative path:



There are three paths:

- from output assertion to F branch of while (ie to assertion), outputting VC1
- along T branch of while from inductive assertion back through statement to inductive assertion*
- from inductive assertion to input assertion, outputting VC3.

* This is called an INVARIANT ASSERTION - it is true when we enter and leave the loop, and true each and every time we go round the loop body.

PARTIAL CORRECTNESS.

Notation: $\{P_1\}\ B\ \{P_2\}$ is called an inductive expression.
P_1, P_2 are predicates, B a program segment. $\vdash$ represents a theorem
This means that if P_1 is true when B is entered,
then if B terminates P_2 will be true on exit.

We thus wish to prove $\{\varphi\}\ P\ \{\psi\}$. Our verification rules consist of the assignment axiom and several rules of inference

ASSIGNMENT AXIOM: $\vdash \{P(\bar{x}, g(\bar{x}, \bar{y}))\}\ y \leftarrow g(\bar{x}, \bar{y})\ \{P(\bar{x}, \bar{y})\}$

(78) RULES OF INFERENCE: The format for each rule is

$$\frac{A_1,\ A_2,\ A_3,\ \ldots,\ A_n}{B}$$

where $\bar{A}$ is the antecedent and B the consequent. This means that to infer B, it is sufficient to prove $\bar{A}$

Assignment Rule $\dfrac{P(\bar{x}, \bar{y}) \Rightarrow q(\bar{x}, g(\bar{x}, \bar{y}))}{\{P(\bar{x}, \bar{y})\}\ \bar{y} \leftarrow g(\bar{x}, \bar{y})\ \{q(\bar{x}, \bar{y})\}}$

ie to prove assignment it is sufficient to prove
$P(\bar{x}, \bar{y}) \Rightarrow q(\bar{x}, g(\bar{x}, \bar{y}))$

Conditional Rules ① $\dfrac{\{P\}\ A\ \{R\}\ B_1\ \{Q\},\ \{P\}\ A\ \{\neg R\}\ B_2\ \{Q\}}{\{P\}\ A;\ \text{if } R \text{ then } B_1 \text{ else } B_2\ \{Q\}}$

② $\dfrac{\{P\}\ A\ \{R\}\ B_1\ \{Q\},\ \{P\}\ A\ \{\neg R\}\ \{Q\}}{\{P\}\ A;\ \text{if } R \text{ then } B_1\ \{Q\}}$

A allows a set of statements between P and the beginning of the if, and may be the null set (probably added by Kaplan for extra boxishness)

eg if $x > y$ then $z \leftarrow x$ else $z \leftarrow y$

$Q = (z = max(x, y))$

antecedents: $\{P\}\ A\ \{x > y\}\ z \leftarrow x\ \{z = max(x, y)\}$
$\{P\}\ A\ \{x \leq y\}\ z \leftarrow y\ \{z = max(x, y)\}$.

(80) Iteration Rule: $\dfrac{\{P\}\ A\ \{I\},\ \{I \wedge R\}\ B\ \{I\},\ \{I \wedge \neg R\}\ \{Q\}}{\{P\}\ A;\ \text{while } R \text{ do } B\ \{Q\}}$

The first antecedent describes the path from invariant upwards into program, the second describes the inner loop, and the third describes the path from the output assertion to the invariant.

Note that each construct has as many antecedents in rule as paths in construct.

Composition Rule: $\dfrac{\{P\}\ B_1\ \{S\},\ \{S\}\ B_2\ \{Q\}}{\{P\}\ B_1; B_2;\ \{Q\}}$

Consequence Rule: $\dfrac{P \rightarrow Q}{\{P\}\ \text{null}\ \{Q\}}$.

By the composition rule, if the output assertion of one path is the input assertion of the next throughout the program, then if we can prove every path correct, the program is partially correct. More formally:

THEOREM: VERIFICATION RULES METHOD

Given a while program P, an input predicate $\varphi(\bar{x})$ and an output predicate $\psi(\bar{x}, \bar{z})$. If, by successively applying the verification rules described above, we can deduce $\{\varphi\} P \{\psi\}$, then P is <u>partially correct w.r.t φ and ψ</u>

(82) ALGORITHM FOR PRODUCING VERIFICATION CONDITIONS

① Start with the output assertion, working backwards through the program. Call this assertion Q

② Apply the inference rule applicable to the last statement encountered. If this rule results in more than one antecedent, stack all the antecedents except for the last one, and save the current position in the program

③ Apply the change indicated by the unstacked antecedent to Q to produce a new Q.

④ If the rule was a consequence rule and we have reached an inductive assertion, then print out the verification condition resulting from the consequence rule.

⑤ If the stack is non-empty, unstack next antecedent and goto ②
If the stack is empty, goto ② and apply next rule.

See examples

TOTAL CORRECTNESS

To prove total correctness, we must show that a program terminates. We use induction, ie, show 0th case holds, and m^{th} case $\Rightarrow$ $m+1$ th case, or $m+1^{th}$ case $\rightarrow$ m^{th} case.

Usually we do it backwards, converging to the 0^{th} case.
We perform the inductive process on programs using well-founded sets ie sets in which any decreasing set of elts has a lower bound and is finite (For example, the set $\mathbb{N}$).

To prove a program totally correct:

1. Prove it is partially correct
2. Find a well-founded set such that when the loop terminates, we are on the 0^{th} case
3. Associate with the loop a BOUND FUNCTION b such that $b(0)$ implies termination
4. Show $b(0) \rightarrow$ termination
5. Show $b(m+1) \rightarrow b(m)$ v termination
6. Deduce termination

See example

EXAMPLES

① Pg (35)

```
begin
    while x1 <> x2 + x3 do
        begin
            x1 := succ(x1)
            x2 := pred(x2)
        end
end
```

On input: (3,2,1) then $\varphi_P^{(3)} : N^3 \to N = 3$

(1,2,3) then $\varphi_P^{(3)} : N^3 \to N = 3$

(1,2) then $\varphi_P^{(2)} : N^2 \to N = \perp$

(1,3) then $\varphi_P^{(2)} : N^2 \to N = 2$.

---

② Pgs 45-6

$$\psi_1(x,y) = \begin{cases} y & \text{if } \varphi_x(x)\downarrow \\ \perp & \text{otherwise} \end{cases}$$ is computable

$$\psi_2(x,y) = \begin{cases} y & \text{if } \varphi_x(x)\downarrow \\ 0 & \text{otherwise} \end{cases}$$ is not computable

$$\psi_3(x,y,z) = \begin{cases} y & \text{if } \varphi_x(x)\downarrow \text{ in } z \text{ steps} \\ 0 & \text{otherwise} \end{cases}$$ is computable.

---

③ Page 83

```
begin                                   ← φ
  (y1, y2) ← (0, x1)
  while y2 ≥ x2 do                      ← invariant
    begin
      y1 ← succ(y1);
      y2 ← y2 − x2;
    end
  (z1, z2) ← (y1, y2)
end                                     ← ψ
```

This program divides x_2 into x_1, producing quotient z_1 and remainder z_2

Our assertions are:

φ : $x_1 \geq 0 \wedge x_2 > 0$

invariant : $x_1 = y_1 * x_2 + y_2 \wedge y_2 \geq 0$

ψ : $x_1 = z_1 * x_2 + z_2 \wedge 0 \leq z_2 < x_2$

Following the algorithm (my pg 30) we get

$Q = \{ x_1 = z_1 * x_2 + z_2 \wedge 0 \leq z_2 < x_2 \}$

Working backwards through the program we reach the assignment statement $(z_1, z_2) = (y_1, y_2)$. Applying the assignment rule, we get

$Q = \{ x_1 = y_1 * x_2 + y_2 \wedge 0 \leq y_2 \leq x_2 \}$

The next statement is a while statement. We check $\{P\}\ A\ \{I\}$ and $\{I \wedge R\}\ B\ \{I\}$, and use $\{I \wedge \neg R\}$ null $\{Q\}$, applying the consequence rule

$(I \wedge \neg R) \Rightarrow Q$

so $\{x_1 = y_1 * x_2 + y_2 \wedge y_2 \geq 0\} \wedge \{y_2 < x_2\} \Rightarrow \{x_1 = y_1 * x_2 + y_2 \wedge 0 \leq y_2 < x_2\}$ (1)

As the rule was a consequence rule and we have reached our invariant assertion, we output this as our first verification condition

Unstack $\{I \wedge R\}\ B\ \{I\}$

we have $Q = \{x_1 = y_1 * x_2 + y_2 \wedge y_2 \geq 0\}$

Apply assignment rule: $(y_1, y_2) \leftarrow (y_1 + 1, y_2 - x_2)$

gives $Q = \{x_1 = (y_1 + 1) * x_2 + (y_2 - x_2) \wedge (y_2 - x_2) \geq 0\}$

This leaves us with $\{I \wedge R\}$ null $\{Q\}$: applying consequence rule we get $\{I \wedge R\} \Rightarrow \{Q\}$

We have once again reached our inductive assertion, so we output our verification condition 2:

$\{x_1 = y_1 * x_2 + y_2 \wedge y_2 \geq 0\} \wedge \{y_2 \geq x_2\} \Rightarrow \{x_1 = (y_1 + 1) * x_2 + (y_2 - x_2) \wedge (y_2 - x_2) \geq 0\}$ (2)

Unstack $\{P\}\ A\ \{I\}$

we have $Q = \{x_1 = y_1 * x_2 + y_2 \wedge y_2 \geq 0\}$

Apply assignment rule for $(y_1, y_2) = (0, x_1)$

gives $Q = \{x_1 = 0 * x_2 + x_1 \wedge x_1 \geq 0\}$

This leaves us with $\{P\}$ null $\{Q\}$, i.e. $\{P\} \Rightarrow \{Q\}$ by conseq rule.

We generate our third verification condition:

$\{x_1 \geq 0 \wedge x_2 > 0\} \Rightarrow \{x_1 = x_1 \wedge x_1 \geq 0\}$ (3)

Notes: ① The verif. conds are not logically complex and rather than needing a complex theorem prover we need to simplify them algebraically

② Sometimes we find that information on the LHS of an implication is not needed to prove the RHS. (eg $x_2 \geq 0$ on VC3) We can often weaken our precondition to the weakest precondition (necessary and sufficient)

We have shown, in our eg:

$\{Q\}\ A\ \{I\}$
$\{I \wedge R\}\ B\ \{I\}$
$\{I \wedge \neg R\}$ null $\{Q'\}$
$\{Q'\}\ A'\ \{\psi\}$

By the composition rule, we may deduce $\{\varphi\}\ P\ \{\psi\}$, ie our program is partially correct w.r.t. $\{\varphi\}$ and $\{\psi\}$.

---

④ Page 88

```
begin
  (y1, y2) ← (x1, x2);
  while y1 ≠ 0 do          ← invariant
    if y2 ≥ y1 then
        y2 ← y2 - y1
    else begin
        y3 ← y1
        y1 ← y2
        y2 ← y3
      end
  z ← y2
end
```

This prog returns the GCD of x_1, x_2 in z.

(Note $\gcd(a,b) = \gcd(b,a)$

$\gcd(a,0) = \gcd(0,a) = a$

if $b > a$ then $\gcd(a,b) = \gcd(a, b-a)$

We must prove the program correct with :

φ : $\{x_1 > 0 \wedge x_2 > 0\}$

invariant : $\{y_1 \geq 0 \wedge y_2 \geq 0 \wedge \gcd(x_1,x_2) = \gcd(y_1,y_2)\}$

ψ : $\{z = \gcd(x_1, x_2)\}$

We start with $Q = \psi = \{z = \gcd(x_1, x_2)\}$

Apply assignment rule : $Q = \{y_2 = \gcd(x_1, x_2)\}$

Stack $\{P\}\ A\ \{I\}$ and $\{I \wedge R\}\ B\ \{I\}$

Use $\{I \wedge \neg R\}$ null $\{Q\}$, by consequence rule $\{I \wedge \neg R\} \Rightarrow \{Q\}$

We get our first VC :

$$\{y_1 \geq 0 \wedge y_2 \geq 0 \wedge \gcd(x_1,x_2) = \gcd(y_1,y_2) \wedge y_1 = 0\} \Rightarrow \{y_2 = \gcd(x_1, x_2)\}$$

Proof : Simplify $\{y_2 \geq 0 \wedge y_1 = 0 \wedge \gcd(x_1,x_2) = \gcd(0,y_2)\} \Rightarrow \{y_2 = \gcd(x_1,x_2)\}$ true.

Unstack $\{I \wedge R\}\ B\ \{I\}$ (*) B is any statement, so we stack $\{I \wedge R'\}\ B_1'\ \{I\}$ and use $\{I \wedge \neg R'\}\ B_2'\ \{I\}$

$Q = \{y_2 \geq 0 \wedge y_1 \geq 0 \wedge \gcd(x_1,x_2) = \gcd(y_1,y_2)\} = I$

B_2' is an assignment sequence. Using the assignment rule we get $Q = \{y_2 \geq 0 \wedge y_1 \geq 0 \wedge \gcd(x_1,x_2) = \gcd(y_2,y_1)\}$

We now have $\{I \wedge \neg R'\}$ null $\{Q\}$ which by consequence rule gives $\{I \wedge \neg R'\} \Rightarrow \{Q\}$

We have reached our invariant, so we can generate our second VC:

$$\{y_1 \geq 0 \wedge y_2 \geq 0 \wedge gcd(x_1, x_2) = gcd(y_1, y_2) \wedge y_1 < y_2\}$$

$$\Rightarrow \{gcd(x_1, x_2) = gcd(y_1, y_2) \wedge y_1 \geq 0 \wedge y_2 \geq 0\}$$

Proof obvious and omitted

Unstack $\{I \wedge R'\}\ B'\ \{I\}$

where B' is $y_2 = y_2 - y_1$, $Q = I$

Use assignment rule:

$$Q = \{y_1 \geq 0 \wedge (y_2 - y_1) \geq 0 \wedge gcd(x_1, x_2) = gcd(y_1, y_2 - y_1)\}$$

We now have $\{I \wedge R'\}$ null $\{I\}$ and are once again at our invariant, giving the third VC:

$$\{y_1 \geq 0 \wedge y_2 \geq 0 \wedge y_2 \geq y_1 \wedge gcd(x_1, x_2) = gcd(y_1, y_2)\}$$

$$\Rightarrow \{y_1 \geq 0 \wedge (y_2 - y_1) \geq 0 \wedge gcd(x_1, x_2) = gcd(y_1, y_2 - y_1)\}$$

Going back to ⊗, we now have $\{I \wedge R\}$ null $\{I\}$ so we can output a verification condition:

$$\{y_1 \geq 0 \wedge y_2 \geq 0 \wedge gcd(x_1, x_2) = gcd(y_1, y_2) \wedge y_1 \neq 0\}$$

$$\Rightarrow \{gcd(x_1, x_2) = gcd(y_1, y_2) \wedge y_1 \geq 0 \wedge y_2 \geq 0\}$$

We next unstack $\{P\}\ A\ \{I\}$, where A is the assignment $(y_1, y_2) = (x_1, x_2)$ and $Q = I$

Applying assignment rule, we get:

$Q = \{x_1 \geq 0 \wedge x_2 \geq 0 \wedge \gcd(x_1, x_2) = \gcd(x_1, x_2)\}$

We have left $\{P\}$ and $\{Q\}$ when $P = \varphi$

Generating the final VC:

$$\{x_1 \geq 0 \wedge x_2 \geq 0\} \Rightarrow \{\gcd(x_1, x_2) = \gcd(x_1, x_2) \wedge x_1 \geq 0 \wedge x_2 \geq 0\}$$

We have shown: $\{\varphi\}\ A\ \{I\}$

$\{I \wedge R\}\ B\ \{I\}$ / $\{I \wedge R'\}\ B_1\ \{I\}$

$\{I \wedge \neg R\}\ B'\ \{\psi\}$ \ $\{I \wedge \neg R'\}\ B_2\ \{I\}$

Therefore $\{\varphi\}\ P\ \{\psi\}$ and P is partially correct w.r.t φ and ψ.

---

⑤ Proving termination: Integer division with remainder

```
begin
    (y1, y2) ← (0, x1)
    while y2 ≥ x2 do
        begin
            y1 = y1 + 1
            y2 = y2 - x2
        end
    od
    (z1, z2) := (y1, y2)
end
```

φ: $\{x_1 \geq 0 \wedge x_2 > 0\}$

invariant $\{x_1 = y_1 * x_2 + y_2 \wedge y_2 \geq 0\}$

ψ: $\{x_1 = z_1 * x_2 + z_2 \wedge 0 \leq z_2 \leq x_2\}$

We need a bound function that will be 0 when $y_2 < x_2$ (ie loop termination), and have the correct inductive behaviour.

As the bound function uses predicate logic, we can use anything about the variables that have the correct inductive behaviour, ie, each time around the loop $b(n) \Rightarrow n$ iterations left to be done and n decreases uniformly on each iteration.

Initially $y_2 = x_1$, but y_2 decreases by x_2 each time as y_1 is incremented

Possible fns: $\left(\frac{x_1}{x_2} - y_1\right)$ or $\left(\frac{y_2}{x_2}\right)$

Both of these are 0 on termination, but the second is better as it decreases by 1 each time!.

Proof: ① Done previously (ex ③)

② Well founded set, as we use $\mathbb{N}$.

③ Use $b(k) = 0 \leq y_2/x_2 \leq k$

④ $b(0) \rightarrow 0 \leq y_2/x_2 \leq 0$, ie $y_2/x_2 = 0$ (integer divide)
ie $y_2 < x_2 \wedge 0 \leq y_2$ on termination

⑤ $b(m+1)$ is $0 \leq y_2/x_2 \leq m+1$

If $y_2/x_2 > 0$
then $y_2 \geq x_2$ so go round loop
which decreases y_2/x_2 by 1
ie $0 \leq y_2/x_2 \leq m$

ie $b(m+1) \rightarrow b(m)$

CSC3000W COMPUTABILITY EXCERCISE

PROVE THE FOLLOWING PROGRAM, WHICH COMPUTES FACTORIALS, TO BE TOTALLY CORRECT WITH RESPECT TO THE ASSERTIONS:

INPUT: $x_1 \geq 0$
INVARIANT: $y_2 = x_1! / y_1!$
OUTPUT: $z_1 = x_1!$ and $y_1 = 0$

USE THE VARIABLE y_1 AS THE BOUND FUNCTION.

```
begin
    (y1, y2) ← (x1, 1)
    while y1 ≥ 0 do
        y2 ← y2 * y1
        y1 ← y1 - 1
    od
    z1 ← y2
end.
```

SOLUTIONS WILL BE POSTED ON THE NOTICEBOARD AT THE END OF THE QUARTER. WARNING: THE EXAM INCLUDES A PARTIAL CORRECTNESS PROOF WHICH IS MORE COMPLEX THAN THIS ONE, IN THAT IT HAS A WHILE LOOP WHICH CONTAINS AN IF STATEMENT. MAKE SURE YOU ARE COMFORTABLE WITH THE IDEA OF STACKING AND UNSTACKING ANTECEDENTS. WHEN NECESSARY. GOOD LUCK!

Exercises Pages 12, 16-17, 27, 37, 44 46.

(12) Let α denote the set of Roman capitals $\{A..Z\}$ and x, y be arbitrary strings of finite length from α. x is called the text string, and y the pattern.

(a) Describe an algorithm to determine if y occurs in x

(b) Describe an algorithm to determine the number of times that y occurs in x.

(16) Show that the following sets have cardinality $\aleph_0$

(a) $\mathbb{N} \times \mathbb{N}$

(b) the set of finite strings from the alphabet $\{A..Z\}$

(c) the set of all finite PASCAL programs

(d) the set of all $x \in \mathbb{Q} : 0 \leq x \leq 1$

Show that the following sets have cardinality $> \aleph_0$

(a) the set of all fns $f: \mathbb{N} \to \{0,1\}$

(b) $2^{\mathbb{N}}$

Say a real number is computable if $\exists$ a program to output the first n decimal places of the number on input $n \in \mathbb{N}$.

(a) Show every rational number is computable

(b) show every square root of an integer is computable

(c) construct a diagonal argument to show that not every real number is computable.

(27) Express 2^z and $\log_2(z)$ in terms of our language.

(39)

(37) Show that the total function $f: \mathbb{N} \to \mathbb{N}$ which returns 0 for all but finitely many of its arguments, is effectively computable.

Show that the function $f: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is computable,

where $f(x, y) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ (y 2's, x 2's)

Show that the integer square root function $\sqrt{x}$ is effectively computable ($i.e.$ $\sqrt{x} = \lfloor \sqrt{x} \rfloor$)

The "excess-over-a-square" function defined by $excess(x) = x \dot{-} (\sqrt{x})^2$. Show that $\forall y$ there are infinitely many $x \in \mathbb{N}$ for which $excess(x) = y$.

Prove the following program, which computes factorials, to be totally correct w.r.t.:

INPUT $x_1 \geq 0$

INVARIANT $y_2 = x_1! / y_1!$

OUTPUT $z_1 = x_1! \wedge y_1 = 0$

```
begin
    (y1, y2) ← (x1, 1)
    while y1 ≠ 0 do
        y2 ← y2 * y1
        y1 ← y1 - 1
    od
    z1 ← y2
end
```

Begin with $Q = \Psi = \{z_1 = x_1! \wedge y_1 = \emptyset\}$

Apply assignment rule to $z_1 \leftarrow y_2$:

$Q = \{y_2 = x_1! \wedge y_1 = \emptyset\}$

We get to the 'while', stack $\{P\}\ A\ \{I\}$ and $\{I \wedge R\}\ B\ \{I\}$

Use $\{I \wedge \neg R\}$ null $\{Q\}$.

By consequence rule, $\{I \wedge \neg R\} \Rightarrow \{Q\}$, so we output VC1:

(VC1) $\{y_2 = x_1!/y_1! \wedge \neg(y_1 > \emptyset)\} \Rightarrow \{y_2 = x_1! \wedge y_1 = \emptyset\}$

Unstack $\{I \wedge R\}\ B\ \{I\}$, so $Q = \{I\}$

Apply assignment rule to $y_1 \leftarrow y_1 - 1$:

$Q = \{y_2 = x_1!/(y_1 - 1)!\}$

Apply assignment rule to $y_2 \leftarrow y_2 * y_1$:

$Q = \{y_2 * y_1 = x_1!/(y_1 - 1)!\}$

We are left with $\{I \wedge R\}$ null $\{Q\}$, so consequence rule gives $\{I \wedge R\} \Rightarrow \{Q\}$

(VC2) $\{y_2 = x_1!/y_1! \wedge y_1 > \emptyset\} \Rightarrow \{y_2 * y_1 = x_1!/(y_1 - 1)!\}$

Unstack $\{P\}\ A\ \{I\}$, so $Q = \{I\}$, $P = \varphi$

Apply assignment rule to $(y_1, y_2) \leftarrow (x_1, 1)$:

$Q = \{1 = x_1!/x_1!\}$

We are left with $\{\varphi\}$ null $\{Q\}$, so

(VC3) $\{x_1 \geq \emptyset\} \Rightarrow \{1 = 1\}$

Simplify VC1: $\{y_1 = \emptyset \wedge y_2 = x_1!\} \Rightarrow \{y_2 = x_1! \wedge y_1 = \emptyset\}$

Simplify VC2: $\{y_2 \ldots\} \Rightarrow \{y_2 = x_1!/(y_1 * (y_1 - 1)!)\}$

so $\{y_2 = x_1!/y_1! \wedge y_1 > \emptyset\} \Rightarrow \{y_2 = x_1!/y_1!\}$

All are tautologies, so program is partially correct →

Use y_1 as the bound function.

$b(\emptyset) \rightarrow \emptyset \leq y_1 \leq \emptyset$ ie $y_1 = \emptyset$ on termination

$b(m+1) \rightarrow \emptyset \leq y_1 \leq m+1$

If $y_1 > \emptyset$ go round loop and $y_1 \leftarrow y_1 - 1$

ie $\emptyset \leq y_1 \leq m$ ie $b(m)$

so $b(m+1) \rightarrow b(m)$ Totally correct.