Mathematical Logic

Frank Stephan

Department of Computer Science and

Department of Mathematics

National University of Singapore

fstephan@comp.nus.edu.sg

Goals of these lectures

Part 1 (Tue): Computation and Decidability

Formalise notion of computation
Universal programs for computation
Limitations of computation
Coding undecidable problems into $(\mathbb{N}, +, *, 0, 1)$
Coding of computations into Natural Numbers

Part 2 (Fri): Search for proofs

Recursively enumerable systems of axioms
Theories provable from r.e. sets of axioms
Primitive recursive functions and sets
Solutions of Tutorial Questions

Part 1

Programs of register machines.
Machine uses finitely many variables, called registers.
Each variable can store any natural number but nothing else.
Programs consist of a numbered sequence of statements.
Statements are done in sequence unless otherwise specified.
Statements can change variable values by assigning additive combinations of variables and constants.
A statement like "Let y = y+x" assigns to y a new value; the new value is the sum of the old value and the current value of x.

Euclid's Algorithm

1. Function GCD(x,y);
2. If x > y then goto 5;
3. If x < y then goto 7;
4. Return(x);
5. Let x = x-y;
6. Goto 2;
7. Let y = y-x;
8. Goto 2;

Explanation

In the first line, the inputs are assigned to registers x and y. The statements in lines 2 and 3 are conditional jumps to lines 5 and 7, respectively, which are taken when the corresponding conditions are true.
The return-statement in line 4 defines the value of the function, provided that it is reached; it is not reached when one but not both variables are 0.
The statements in lines 5 and 7 replace the variable with the larger value that value by the difference of the two values in the variables.
In each round, the larger value in the variables is replaced by the difference until both variables are equal; that value is then returned as the greatest common divisor.

Multiplication

This and the following examples show that all functions from natural numbers to natural numbers which can be computed by computer programs can actually be computed by register programs. Here the multiplication.

1. Function Mult(x,y)
2. Let v = 0;
3. Let w = 0;
4. If w < y then goto 6;
5. Return(v);
6. Let v = v+x;
7. Let w = w+1;
8. Goto 4;

Multiplication can be carried out as a series of additions.

Exponentiation

1. Function Eponentiationwithbasetwo(x);
2. Let y = 1;
3. Let z = 0;
4. If z < x then goto 6;
5. Return(y);
6. Let y = y+y;
7. Let z = z+1;
8. Goto 4;

Exponentiation can also be mapped back to loops and addition.

Factorial

1. Function Factorial(x);
2. Let z = x;
3. Let y = 1;
4. If z > 0 then goto 6;
5. Return(y);
6. Let y = Mult(y,z);
7. Let z = z - 1;
8. Goto 4;

The factorial can be expressed using the function for multiplication.

Cantor Pairing Function

1. Function Cantor(x,y);
2. Let v = 0;
3. Let w = y;
4. If v < x+y then goto 6;
5. Return(w);
6. Let v = v+1;
7. Let w = w+v;
8. Goto 4;

The function Cantor(x,y) computes a number z which codes x,y according to the formula $z = (0+1+2+\ldots+(x+y))+y$; this formula can also be written as $z = (x+y)*(x+y+1)/2+y$. This function is a bijection from pairs to numbers and can be used to store data.

Inverse for x

1. Function Cantorx(z);
2. Let v = 0;
3. Let w = 0;
4. If z > v+w then goto 8;
5. Let y = z-v;
6. Let x = w-y;
7. Return(x);
8. Let w = w+1;
9. Let v = v+w;
10. Goto 4;

The function Cantorx(z) computes the values x,y with z = Cantor(x,y) and then takes x as the return value.

Inverse for y

1. Function Cantory(z);
2. Let v = 0;
3. Let w = 0;
4. If z > v+w then goto 8;
5. Let y = z-v;
6. Let x = w-y;
7. Return(y);
8. Let w = w+1;
9. Let v = v+w;
10. Goto 4;

The function Cantory(z) computes the values x,y with z = Cantor(x,y) and then takes y as the return value. The only difference to the program of the previous slide is in line 7 when y is returned in place of x.

Subprograms

Some of the above slides and many slides below will use subprograms. This is for the reader's convenience, as too lengthy programs might be too difficult to understand. Nevertheless, the subprograms could be worked into the mother program and the resulting program would have a more complicated structure of jumps and more variables but it would not need any subprograms.
All what can be programmed in this paradigm (data type natural numbers) can be programmed with register machines where the basic operations are to compare variables and to assign to variables values obtained by adding or subtracting values of variables and constants. Furthermore, one needs the ability to jump.

Arrays

An array is a sequence $a_0, a_1, \ldots$ of numbers where almost all of them are 0. It is storred in a number a as Cantor(a_0,Cantor(a_1,Cantor(a_2,...))) for which should be noted that Cantor(0,0) = 0.
One can scroll an array from the beginning to the point where one wants to read or write the array. For the scrolling, one has to decompose the array head into the current element and the part right of it and at the same time to save the scrolled part in a variable b.
After accessing a_n, the scrolling has to go the other way.
Arrays permit to simulate programs which have many line numbers and are stored in a variables.
Arrays also permit to simulate the registers of such a program as there might be many.
So defining arrays is a first step on defining register machines which can simulate other register machines.

Reading in Arrays

```
1. Function Read(a,n)
2. Let m = 0; Let b = 0; Let x = 0;
3. Let b = Cantor(b,x);
4. Let x = Cantorx(a);
5. Let a = Cantory(a);
6. If m < n then goto 8;
7. Let y = x; Goto 9;
8. Let m = m+1; Goto 3;
9. Let m = 0;
10. Let a = Cantor(x,a);
11. Let x = Cantory(b);
12. Let b = Cantorx(b);
13. If m < n then goto 15;
14. Return(y);
15. Let m = m+1;
16. Goto 10;
```

Writing in Arrays

1. Function Write(a,n,y)
2. Let m = 0; Let b = 0; Let x = 0;
3. Let b = Cantor(b,x);
4. Let x = Cantorx(a);
5. Let a = Cantory(a);
6. If m < n then goto 8;
7. Let x = y; Goto 9;
8. Let m = m+1; Goto 3;
9. Let m = 0;
10. Let a = Cantor(x,a);
11. Let x = Cantory(b);
12. Let b = Cantorx(b);
13. If m < n then goto 15;
14. Return(a);
15. Let m = m+1;
16. Goto 10;

Universal Function

There is a universal function which computes on inputs e and x the output of the e-th register machine on input x.
This function uses an array to store the register program.
The position of the statement in the array coincides with the line number.
The statement is a four-tuple consisting of the type of statement and three arguments; the statement type tells whether these arguments refer to variables, constants or line numbers.
For example, a statement of type 1 could just be a goto statement which jumps to the line in the first parameter.
The statement of type 2 could be an update of the form Let a[d]=a[f]+a[g] where d,f,g are the three parameters which refer to entries in the array a of the variables.
The variables are stored in an array as well.

Universal Function

There are only finitely many types of statements.
The interpreter runs in a cycle.
In each cycle, it takes the current line number l and reads out the statement from the array p representing the program. It furthermore reads out the values of the parameters and that of the variables in question.
Depending on the statement type, it does the following:
(a) Computing the new value of the corresponding variable and updating the variable array and going to statement l+1;
(b) Directly jumping to the new statement number according to the first parameter;
(c) Comparing two values and jumping to the new statement number if the comparison evaluates to true and going to the next statement l+1 if the comparison evaluates to false.
This cycle is repeat until a return-statement with the return value is reached. This event is denoted as “halting”.

General Idea for Universal Machine

The variables e and a are interpreted as arrays holding the program and the data, respectively, and l is the line number. In each cycle, the universal machine reads command i from entry l of the array e which holds the program and decomposes i into c,d,f,g where c is a code for the operation and d,f,g are parameters:

Code c = 0 stands for halting with output a[d];
Code c = 1 stands for going to line d;
Code c = 2 stands for a[d] = f;
Code c = 3 stands for a[d] = a[f]+a[g];
Code c = 4 stands for a[d] = a[f]-a[g];
Code c = 5 stands for a[d] = a[f]+g;
Code c = 6 stands for a[d] = a[f]-g;
Code c = 7 stands for if a[f] < a[g] then goto d.
Note that the syntax is a bit more restrictive than in the sample programs.

Example of a Universal Machine

```
1. Function Universal(e,x);
2. Let l = 1; Let a = Cantor(x,0);
3. Let i = Read(e,l);
4. Let c = Cantorx(Cantorx(i)); Let d = Cantory(Cantorx(i));
5. Let f = Cantorx(Cantory(i)); Let g = Cantory(Cantory(i));
6. If c > 0 Then goto 9;
7. Let h = Read(a,d);
8. Return(h);
9. If c > 1 then goto 11;
10. Let l = d; Goto 3;
11. If c > 2 then goto 13;
12. Let a = Write(a,d,f); Let l = l+1; Goto 3;
13. If c > 3 then goto 16;
14. Let v = Read(a,f); Let w = Read(a,g); Let u = v+w;
15. Let a = Write(a,d,u); Let l = l+1; Goto 3;
```

Continued next slide

Example Implementation Continued

16. If c > 4 then goto 19;
17. Let v = Read(a,f); Let w = Read(a,g); Let u = v-w;
18. Let a = Write(a,d,u); Let l = l+1; Goto 3;
19. If c > 5 then goto 22;
20. Let v = Read(a,f); Let u=v+g; a = Write(a,d,u);
21. Let l = l+1; Goto 3;
22. If c > 6 then goto 25;
23. Let v = Read(a,f); Let u=v-g; a = Write(a,d,u);
24. Let l = l+1; Goto 3;
25. Let v = Read(a,f); Let w = Read(a,g);
26. If v < w then goto 28;
27. Let l = l+1; Goto 3;
28. Let l = d; Goto 3;

Undecidability of the Halting Problem

Let φ_e be the function computed by the e-th register machine with one input x; that is, $\varphi_e(x)$ is the outcome of the computation Universal(e,x). The function φ_e is called the e-th partial-recursive function.
Assume that a program Halt could test whether $\varphi_e(x)$ halts.
Consider the following program.
1. Function Diag(e);
2. Let a = Halt(e,e); Let b = 0;
3. If a < 1 then goto 6;
4. b = Universal(e,e);
5. b = b+1;
6. Return(b);

This function computes a function different from all partial-recursive functions.

Diagonalisation

The function Diag(e) returns on input e the value 0 if $\varphi_e(e)$ is undefined and the value $\varphi_e(e)+1$ if that is defined. It uses the function Universal(e,x) simulating $\varphi_e(x)$ and the function Halt(e,x) which returns 1 whenever $\varphi_e(x)$ halts and 0 if it does not halt.

If now φ_e computes the function Diag(e) then $\varphi_e(e)$ is defined. Hence Halt(e,e) is 1 and Universal(e,e) is $\varphi_e(e)$. The function Diag takes then the value $\varphi_e(e)+1$, a contradiction.
Hence Halt does not exist and the halting problem is undecidable.

Problems like the halting problem are called undecidable. That means, there is no register machine which computes the characteristic function of the set $\{(e,x) : \varphi_e(x)$ halts$\}$.

Halting Behaviour of Functions

1. Function Collatz(n)
2. Let m = n;
3. Let k = 0;
4. If k+k = m then goto 7;
5. If k+k+1 = m then goto 8;
6. Let k = k+1; Goto 4;
7. Let m = k; Goto 3;
8. If m = 1 then goto 10;
9. Let m = m+m+m+1; Goto 3;
10. Return(1);

Does the function Collatz(n) halt for all natural numbers n>0?
Lothar Collatz conjectured this in 1937 and it is until today an open problem.

Searches which might not halt

Does a program which for input n finds the smallest $p \geq n$ such that p and $p+2$ are both primes halt for all n? It is easy to write such a program, but no one was up to now able to prove that there are indeed infinitely many twin primes.

Another search which might never terminate is the search for an even number $n > 2$ such that n is not the sum of two prime numbers. Again it is an open question in mathematics whether such an n exists; on June 1742 Christian Goldbach conjectured in a letter to Leonhard Euler that every integer greater than 5 can be written as the sum of up to three primes. A further conjecture (now known as the Goldbach conjecture) says that every even number greater than 2 is the sum of two primes: 4 = 2+2, 6 = 3+3, 8 = 3+5, 10=3+7, 12=5+7, 14=11+3, 16=13+3, 18=13+5, 20=17+3 and so on.

Partial-recursive functions

Besides register machines, there are also other approaches to define the notion of functions computed by machines.

One is with recursion: Starting from few base cases, one can define more involved functions by recursive schema like $f(0) = 1$ and $f(n+1) = f(n) + f(n)$ for the powers of 2. Furthermore, one can define new functions from old ones by search, for example $\log(n)$ is the first m with $f(m) \geq n$ (rounded value as only natural numbers are considered). The combination of these two principles — recursion and search — gives rise to the notion of partial-recursive functions. They are called "partial-recursive" as the function can be undefined when for some arguments the corresponding search does not terminate.

Church's Thesis

Church stated the thesis that all reasonable notions of computation give rise to the same concept: that of partial-recursive functions.

This has been verified for many notions of computation which had been formalised and so there is much evidence for this thesis.

Therefore, when proving that a function is partial-recursive, often the algorithm is given in an informal way without specifying register programs as done on the first slides.

Modern Programming Languages

Modern programming languages do not write programs as numbered lists of statements but rather with structured commands like “if - then - else”, “while - do” and “for - to - do”. Here two sample programs.

```
Function Isprime(x)
Begin If x > 1 Then r = 1 Else r = 0;
For y = 2 To x Do Begin
For z = 2 To x Do Begin
If Mult(y,z) == x Then r = 0 End End;
Return(r) End;
```

```
Function Findnexttwinprime(x)
Begin y = x+1;
While Isprime(y)==0 Or Isprime(y+2)==0
Do Begin y=y+1 End;
Return(y) End;
```

Recursively enumerable sets

A set is recursively enumerable iff there is a recursive function enumerating all its values. The recursively enumerable sets coincide with domains of partial-recursive functions and also with ranges of partial-recursive functions. This notion turned out to be very fruitful in the theory of computing.

A set is recursive iff there is a recursive function deciding its values.

The halting problem is by definition a recursively enumerable set which is not recursive.

Characterising recursive sets

THEOREM.
A set A is recursive if both, A and $\mathbb{N} - A$ are recursively enumerable.

It is clear that if A is recursive then both A and its complement are also r.e. sets.

So assume now that A is the range of a recursive function f and its complement the range of a recursive function g. Given input x, one can search for the first y such that either $f(y) = x$ or $g(y) = x$. If $f(y) = x$ then $x \in A$ and if $g(y) = x$ then $x \notin A$. The search terminates for every x as the union of the ranges of f and g is $\mathbb{N}$. Thus the informally given algorithm is a decision procedure for A and A is recursive.

Recursive subsets of r.e. sets

THEOREM.
Every infinite r.e. set has an infinite recursive subset.

Assume that A is an infinite r.e. set. Then A is the range of a recursive function f. Now let

$$B = \{y : \exists v\, [v = f(v) \wedge \forall w < v\, [f(w) < y]]\}.$$

It is easy to see that the set B is recursively enumerable, as one can construct from f inductively a function g such that $g(0) = f(0)$ and $g(n+1) = f(\min\{m : f(m) > g(n)\})$. This function g is recursive and it is strictly monotonically increasing. Now the formula

$$y \in B \Leftrightarrow y \in \{g(0), g(1), \ldots, g(y)\}$$

provides a decision procedure for B using the function g.

Existential and Bounded Quantifiers

A formula $\exists b \forall u < b[x \neq u \cdot u \wedge x < b]$ is called an existential bounded formula. Such type of formulas are quite flexible and can be used to describe many sets. A Σ_1^0-set is a set which is defined by existentially bounded formulas, that is, $x \in A$ iff a formula of above type is true.

It can easily be seen that every Σ_1^0-set is recursively enumerable. Given the existentially bounded formula, one can, for any possible bound, check by exhaustive search whether the condition specified by the bounded quantifiers can be made true. One can easily make a partial-recursive function which for input x returns the first bound which makes the formula true (if it exists); then the set of all x where the function is defined coincides with A and hence A is recursively enumerable.

Examples

The formula $Prime(u)$ denotes whether u is a prime:
$Prime(u) \Leftrightarrow \forall x < u \forall y < u[1 < u \wedge (x+2)\cdot(y+2) \neq u]$;
$Divides(v,w) \Leftrightarrow \exists u \leq w\,[w = u \cdot v]$;
$Primepower(x) \Leftrightarrow \exists y \leq x \forall z \leq x[Divides(y,x) \wedge Prime(y) \wedge (Divides(z,x) \wedge Prime(z) \rightarrow z = y)]$.
All these formulas can be expressed using bounded quantifiers only and are therefore Σ_1^0-formulas; their negations (due to the absence of unboundedly quantified variables) are also Σ_1^0-formulas.

Machine configuration

For a given register machine program, a configuration of the machine is a tuple consisting of the current line number l and the current values of the variables before the step of line l is executed. One can code such configurations in one number; for example if the variables are x,y,z then the configuration is Cantor(l,Cantor(x,Cantor(y,z))) for the function Cantor defined previously.

A computation is a sequence of configurations such that each two subsequent configurations (l,x,y,z) and (l’,x’,y’,z’) it holds that (l’,x’,y’,z’) is the configuration after doing the statement in line l and adjusting the variables accordingly.

The next slides make a formula to define the factorial as defined on Slide 8 by using configurations for that program.

Elements of the formula

The formula is the following: w is the factorial of v iff there is a prime p and a computation c such that for all powers q of p with $q < c$ the following conditions hold:

1. If $q * p < c$ then c is of the form $r + q * Cantor(l, Cantor(x, Cantor(y, z))) + q * p * Cantor(l', Cantor(x', Cantor(y', z'))) + s * q * p * p$ where $r < q$, $Cantor(l, Cantor(x, Cantor(y, z))) < p$, $Cantor(l', Cantor(x', Cantor(y', z'))) < p$ and (l', x', y', z') is the next step after (l, x, y, z);
2. If $q = 1$ then $l = 1$ and $x = v$;
3. If $q < c < q * p$ then $c = r + q * Cantor(l, Cantor(x, Cantor(y, z)))$ with $r < q$, $l = 5$ and $y = w$.

Conditions on Transition

In 1., the conditions on (l, x, y, z) and (l', x', y', z') are the following:

$l = 1 \Rightarrow l' = 2$ and $x' = x$ and $y' = y$ and $z' = z$;
$l = 2 \Rightarrow l' = 3$ and $x' = x$ and $z' = x$ and $y' = y$;
$l = 3 \Rightarrow l' = 4$ and $x' = x$ and $y' = 1$ and $z' = z$;
$l = 4 \Rightarrow (z > 0 \rightarrow l' = 6 \wedge z = 0 \rightarrow l' = 5)$ and $x' = x$ and $y' = y$ and $z' = z$;
$l = 5 \Rightarrow l' = l$ and $x' = x$ and $y' = y$ and $z' = z$ (a halted program has no activity);
$l = 6 \Rightarrow l' = 7$ and $x' = x$ and $y' = y * z$ and $z' = z$;
$l = 7 \Rightarrow l' = 8$ and $x' = x$ and $y' = y$ and $z' + 1 = z$;
$l = 8 \Rightarrow l' = 4$ and $x' = x$ and $y' = y$ and $z' = z$.

Summary of Formula

The previous slide made a Σ^0_1 formula $\phi(v,w)$ of the form
$\exists c \exists p < c \forall q < l, l', x, y, z, x', y', z', r, s < c\, [p$ is a prime and
(if q is a power of p and c is of the form
$r + q * C(l,x,y,z) + q * p * C(l',x',y',z') + q * p^2 * s$ and
$C(l,x,y,z), C(l',x',y',z') < p$ then the conditions from last slide hold) and
(if $c = C(l,x,y,z) + p * s$ then $l = 1$ and $x = v$) and
(if q is a power of p and $c = r + q * C(l,x,y,z)$ then $l = 5$ and
$y = w)]$.
So the formula $\phi(v,w)$ is true iff the computation of the program from slide 8 terminates and produces the output w from v.
Note that here the computation is stored as a number
$c = c_0 + pc_1 + p^2 c_2 + \ldots + p^m c_m$ where c_0 is the initial and c_m the halting configuration; any prime p which is larger than all configurations in the computation can be used for the coding.

Universal Formula

One can make a formula $\exists p, c[\phi(e, c, p, x, y)]$ with additional bounded quantification over the variables which is true iff p, c code a halting computation with output y for $\varphi_e(x)$. Similarly $Halt(e, x)$ is true iff $\exists p, c, y[\phi(e, c, p, x, y)]$. Note that the body of the formula only contains addition and multiplication and other operations from the natural numbers and Boolean combinations of the resulting terms. Hence the theory of $(\mathbb{N}, +, *, 0, 1)$ is undecidable.

A byproduct of this proof is that every recursively enumerable set A of natural numbers can be represented by a Σ_1^0 formula as one can choose an e for which $\varphi_e(x)$ exactly on the members x of A halts; then $x \in A \Leftrightarrow \exists p, c, y[\phi(e, c, p, x, y)]$.

Diophantine Sets

A special case of Σ_1^0 sets are Diophantine sets. Here a set A is Diophantine iff there is a polynomial p with integer coefficients (can be negative) in several variables such that $x \in A$ iff there are natural numbers $y_1, y_2, \ldots, y_n$ with $p(x, y_1, y_2, \ldots, y_n) = 0$.

Example: $x \in A \Leftrightarrow \exists y[x - y \cdot y = 0]$ defines the set of all square numbers;
$x \in B \Leftrightarrow \exists y, v, w[(y^2 + 1 + v - x)^2 + ((y + 1)^2 - x - w - 1)^2 = 0]$ defines the set of all numbers which are not squares;
$x \in C \Leftrightarrow \exists y, z[x - (y + 2) * (z + 2) = 0]$ defines the set of all composite numbers.

Hilbert's Tenth Problem

In the year 1900 at the International Congress of Mathematician, David Hilbert formulated 23 problems which mathematicians should solve during the century. The tenth problem was to find an algorithm to solve Diophantine equations and to check whether a Diophantine set is non-empty.

In the year 1944, Emil Leon Post conjectures that Hilbert's tenth problem does not have an algorithm, but that instead one needs an undecidability proof.

1949 Martin Davis conjectures that the Diophantine sets coincide with the recursively enumerable sets and that therefore there is a Diophantine set which is undecidable.

1970 Matiyasevich proves the conjecture of Martin Davis and establishes that every r.e. set of natural numbers is Diophantine.

Part 2

Part 1 formalised the notion of computation and showed that one can express in formulas using arithmetics over the natural numbers whether a computation halts. Furthermore, Part 1 showed that there are problems related to computation which are undecidable, most notable the halting problem of programs.

Part 2 will formalise the notion of proof. It is shown that one can formulate algorithms which check whether a certain proof is correct. Furthermore, Part 2 deals with the notion of axiomatisable theories and their treatment from the viewpoint of computation. The central notion here the notion of recursively enumerable and decidable sets of formulas.

Formalising Proofs and Formulas

Goal is to formalise the notion of proof.
The first part is to convert all formulas into an abstract representation.
Enumerate all variables and symbols, so $x_0, x_1, x_2, \ldots$, $c_0, c_1, c_2, \ldots$, $f_0, f_1, f_2, \ldots$ and $r_0, r_1, r_2, \ldots$ where f_n has n inputs (may be defined to be irrelevant if not needed by $\forall x_0, x_1, x_2\, [f_2(x_0, x_1) = f_2(x_0, x_2)]$) and similarly for relations.
Furthermore, one can construct terms and prime formulas by some fixed conventions, see next slide.

The exact form of the coding is not so important; it is only important that all formulas can be found explicitly and that one can test explicitly with a computer program whether certain terms or formulas have a collision, whether formulas are prime, which variables in formulas are free and so on.

Sample Definition of Formulas

Here a sample coding for terms and formulas (Rautenberg has some other, own method) where C is a tuple version of Cantor's pairing function:

$t_{C(0,n)}$ represents c_n,

$t_{C(1,n)}$ represents x_n,

$t_{C(2,n,a_1,a_2,\ldots,a_n)}$ represents $f_n(t_{a_1}, t_{a_2}, \ldots, t_{a_n})$,

$\phi_{C(3,i,j)}$ represents the equation $t_i = t_j$,

$\phi_{C(4,n,a_1,a_2,\ldots,a_n)}$ represents $r_n(t_{a_1}, t_{a_2}, \ldots, t_{a_n})$,

$\phi_{C(5,i,j)}$ represents $\phi_i \wedge \phi_j$,

$\phi_{C(6,i,j)}$ represents $\phi_i \vee \phi_j$,

$\phi_{C(7,i)}$ represents $\neg\phi_i$,

$\phi_{C(8,i,j)}$ represents $\phi_i \rightarrow \phi_j$,

$\phi_{C(9,i,j)}$ represents $\forall x_i[\phi_j]$,

$\phi_{C(10,i,j)}$ represents $\exists x_i[\phi_j]$.

Formulas and Proofs

The set F of all codes of legal formulas and the set S of all codes of legal sentences (= closed formulas) are recursive.

Having these sets, one can now say that a theory T (given by its codes) is axiomatisable iff $T \subseteq S$ and there exists an r.e. set $R \subseteq F$ such that every ϕ_e with $e \in T$ can be proven from the universal closures of the sets ϕ_d with $d \in R$.

Here a proof of a formula $\phi_e \in T$ as a finite tuple of the form $(n, (e_0, p_0, i_0, j_0), (e_1, p_1, i_1, j_1), \ldots, (e_n, p_n, i_n, j_n))$ where for $e_k = 0$ the formula ϕ_{p_k} is the i_k-th member of a recursive enumeration of R and for $e_k = 1$ the formula ϕ_{p_k} is the i_k-th member of a recursive enumeration of the tautologies $\Lambda_1, \ldots, \Lambda_{10}$ on page 122 and for $e_k = 2$ it holds that $i_k, j_k < k$ and $\phi_{p_{j_k}}$ is of the form $\phi_{p_{i_k}} \to \phi_{p_k}$ so that ϕ_{p_k} is obtained by using the modus ponens on $\phi_{p_{i_k}}$ and $\phi_{p_{j_k}}$.

Hilbert's Axioms

The set of Hilbert's tautologies consists of the following formulas (with parameter formulas α, β, γ and parameter variables x, y and parameter terms t):

Λ_1: $(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$;
Λ_2: $\alpha \to \beta \to \alpha \wedge \beta$;
Λ_3: $\alpha \wedge \beta \to \alpha$ and $\alpha \wedge \beta \to \beta$;
Λ_4: $(\alpha \to \neg\beta) \to \beta \to \neg\alpha$;
Λ_5: $\forall x[\alpha] \to \alpha\frac{t}{x}$ provided that $\alpha, \frac{t}{x}$ are collision-free;
Λ_6: $\alpha \to \forall x[\alpha]$ provided that $x \notin free(\alpha)$;
Λ_7: $\forall x[\alpha \to \beta] \to \forall x[\alpha] \to \forall x[\beta]$;
Λ_8: $\forall y[\alpha\frac{y}{x}] \to \forall x[\alpha]$ provided that $y \notin var(\alpha)$;
Λ_9: $t = t$;
Λ_{10}: $x = y \to \alpha \to \alpha\frac{y}{x}$ provided that α is a prime formula.

Λ contains these formulas and all formulas of the form $\forall x_1 \forall x_2 \ldots \forall x_n[\phi]$ derived from ϕ as above.

Completeness of Hilbert Calculi

THEOREM.
Given a set X of formulas and a formula α. Then $X \models \alpha$ iff α can be proven from X using the Hilbert calculus.

THEOREM.
The following is equivalent for a formula α:
(a) α can be derived using the rules of the Hilbert calculus and the modus ponens;
(b) α can be derived using the rules of the Hilbert calculus and the modus ponens and the operation which replaces any formula β by $\forall x[\beta]$;
(c) α is a tautology, that is, $X \models \alpha$ for all sets X of formulas.

Axiomatisable Theories

THEOREM.
If a theory is axiomatisable then its theorems are recursively enumerable.

Given an axiomatisable theory T and the r.e. set R consisting of the axioms, one enumerates all sentences ϕ_e with $e \in S$ for which there is a proof. So the naive method to enumerate the theory is just to do an exhaustive search over all possible proofs and to enumerate those sentences for which a valid proof has been found.

COROLLARY.
The set of tautologies (sentences true in every model) is recursively enumerable as it coincides with those sentences which have a proof.

Example

THEOREM.
Let $(F, +, *, 0, 1)$ be a fixed finite field. Then the theory of this finite field is axiomatisable.

The reason is that one can evaluate all terms and quantifiers efficiently. Given a term t and knowing all the values of the variables involved, one can compute the value of the term t. Similarly, one can compare the values of terms. As the only predicate is the equality of two terms, one can also evaluate all prime formulas (provided that the variable values are known). For quantified formulas, they variables range only over the finitely many values of F and so one can evaluate $\exists x[\phi]$ by checking out whether one of the finitely many values in F for x makes the formula ϕ true and correspondingly with $\forall x[\phi]$. This permits to determine the truth-value of every sentence.

Outlook

Gödel proved that there is a r.e. axiomatisable theory T using the language of arithmetics such that every further theory $T' \supseteq T$ is either incomplete or inconsistent or not axiomatisable. He proved two theorems known as the First and Second Incompleteness Theorem.

Hilbert asked 1928 whether there is any algorithm which can decide whether a sentence is true in arithmetics. The language use are the natural numbers with addition and multiplication.

Church 1936 and Turing 1937 developed a theory of algorithms and then showed that an algorithm as asked for by Hilbert cannot exist.

Primitive Recursive Functions

Primitive recursive functions are the smallest class of functions satisfying 1-3 below.

1. Every constant function and every function selecting a variable out of many is primitive recursive, for example $f(x, y, z) = y$ would be a primitive recursive function. Furthermore, the function $Succ(x) = x + 1$ is primitive recursive.

2. The concatenation of primitive recursive functions is primitive recursive.

3. If an n-ary function g and an $n+2$-ary function h is primitive recursive then there is a unique $n=1$-ary function f satisfying $f(x_1, x_2, \ldots, x_n, 0) = g(x_1, x_2, \ldots, x_n)$ and $f(x_1, x_2, \ldots, x_n, y + 1) = h(x_1, x_2, \ldots, x_n, y, f(x_1, x_2, \ldots, x_n, y))$ and this function f is primitive recursive as well.

Examples

The addition is primitive recursive. The function mapping x, y to $x + y$ satisfies $x + 0 = x$ and $x + (y + 1) = (x + y) + 1$ where the second equation is formally be given derived from three inputs by $h(x, y, x + y) = (x + y) + 1$.

The multiplication is primitive recursive. The function mapping x, y to $x \cdot y$ has the base case $x \cdot 0 = 0$ and the inductive condition is $h(x, y, x \cdot y) = (x \cdot y) + x$ using addition and projection of functions.

The exponentiation is primitive recursive (working with $0^0 = 1$) as given by the base-case $x^0 = 1$ and the inductive case $x^{y+1} = x^y \cdot x$.

The factorial is primitive recursive by $0! = 1$ and $(x + 1)! = x! \cdot (x + 1)$.

Prim. Rec. and Register Machines

One can show by induction that every primitive recursive function is computed by a register machine. This is clearly due for the functions coming from cases 1 and 2 in the definition. Here case 3 with sample arity 2:

1. Function f(x,y) made from g and h
2. Let z = g(x,0); Let u = 0;
3. If u < y then goto 5;
4. Return(z);
5. Let z = h(x,u,y);
6. Let u = u+1;
7. Goto 3;

This program computes f using programs for g and h

Ackermann function

Wilhelm Ackermann provided the following example of an easy function which has a recursive definition in two parameters but which is not primitive recursive:

$$f(x,y) = \begin{cases} y+1 & \text{if } x = 0; \\ f(x-1,1) & \text{if } x > 0 \wedge y = 0; \\ f(x-1, f(x,y-1)) & \text{if } x > 0 \wedge y > 0. \end{cases}$$

This function is not primitive recursive; indeed, the diagonal $x \to f(x,x)$ grows faster than every primitive recursive function with one input.

Numbering

It is possible to make a numbering of all primitive-recursive functions. That is, a function $e, x \mapsto \vartheta_e(x)$ such that every primitive recursive function with one input equals to some ϑ_e and that the full function can be computed by a register machine.

This function, in whatever way it is chosen, is an example of a total function which is computed by a register machine but which is not primitive recursive. The reason is that the function $sum(x) = \vartheta_0(x) + \vartheta_1(x) + \ldots + \vartheta_x(x)$ grows faster than all ϑ_e; this function sum would be primitive recursive whenever the function $e, x \mapsto \vartheta_e(x)$ is primitive recursive.

Primitive Recursive Sets

A set is primitive recursive iff its characteristic function is primitive recursive.

Given a coding (as on slides 38 and 39), one can show that the following sets are primitive recursive:

The set of all terms;
The set of all formulas;
The set of all sentences;
The set of all logical axioms of the Hilbert calculus;
The set of all correct proofs.

Not every decidable set is primitive recursive: one can make a recursive listing $A_0, A_1, \ldots$ of all primitive recursive sets and then the universal set $\{(e, x) : x \in A_e\}$ and the diagonal $\{e : e \in A_e\}$ are both not primitive recursive.

Recursive Functions

Partial-recursive functions can be obtained from primitive recursive functions by search. Given two primitive recursive functions g and h in $n+1$ inputs, one can define $f(x_1, x_2, \ldots, x_n)$ to be $g(x_1, x_2, \ldots, x_n, t)$ for the first t found such that $h(x_1, x_2, \ldots, x_n, t) = 1$. The resulting f can be partial as the value t might never be found; in the case that f is nevertheless total, f is just called a recursive function.

All the functions which are definable using register machines are partial-recursive. The proof is done using a run-time parameter t for the universal function so that it runs the main loop only t times.

Realising Universal Machine by Search

A counter s is introduced for the universal function to measure how often the simulator runs through line 3. In each round of the simulation, the function goes through this line and makes the parameter s by 1 larger. If $s = t$ then f/g halt outputting the corresponding configuration. If s is large enough so that the functions reach termination then $g(e, x, t)$ outputs the function value and $h(e, x, t)$ outputs 1 (W.l.o.g. 1 differs from all configurations).

The program on the next slides is for both functions f and g. The update from one configuration to the next is primitive recursive, hence also the functions f and g defined there are primitive recursive.

Now $\varphi_e(x)$ is $g(e, x, t)$ for the first t where $h(e, x, t) = 1$.

Universal Machine with Counter

1. Function Universal(e,x,t);
2. Let l = 1; Let a = Cantor(x,0); Let $s = 0$;
3. Let i = Read(e,l); Let $s = s + 1$; If $s = t$ then goto 29;
4. Let c = Cantorx(Cantorx(i)); Let d = Cantory(Cantorx(i));
5. Let f = Cantorx(Cantory(i)); Let g = Cantory(Cantory(i));
6. If c > 0 Then goto 9;
7. Let h = Read(a,d);
8. Return(h); (for function g) // Return(1); (for function h)
9. If c > 1 then goto 11;
10. Let l = d; Goto 3;
11. If c > 2 then goto 13;
12. a = Write(a,d,f); Let l = l+1; Goto 3;
13. If c > 3 then goto 16;
14. Let v = Read(a,f); Let w = Read(a,g); Let u = v+w;
15. a = Write(a,d,u); Let l = l+1; Goto 3;

Continued next slide

Example Implementation Continued

```
16. If c > 4 then goto 19;
17. Let v = Read(a,f); Let w = Read(a,g); Let u = v-w;
18. a = Write(a,d,u); Let l = l+1; Goto 3;
19. If c > 5 then goto 22;
20. Let v = Read(a,f); Let u=v+g; a = Write(a,d,u);
21. Let l = l+1; Goto 3;
22. If c > 6 then goto 25;
23. Let v = Read(a,f); Let u=v-g; a = Write(a,d,u);
24. Let l = l+1; Goto 3;
25. Let v = Read(a,f); Let w = Read(a,g);
26. If v < w then goto 28;
27. Let l = l+1; Goto 3;
28. Let l = d; Goto 3;
29. Return Cantor(l,a)+2;
```

Equivalence

One can show along the above ideas that a partial function is partial-recursive iff it is computed by a register machine.

One can show that a set is decidable iff its characteristic function is recursive iff its characteristic function is computed by a register machine.

Gödel used in his paper primitive recursive and recursive functions to get his proof; the equivalent notions using more explicit computing devices like register machines and Turing machines followed later.

Recursive functions have the advantage that one sees easily that they are inductively definable from the base cases and can therefore be used in logic.

Robinson Arithmetic

1. $\forall x\,[Succ(x) \neq 0]$;
2. $\forall x \forall y\,[Succ(x) = Succ(y) \rightarrow x = y]$;
3. $\forall x \neq 0 \exists y\,[x = Succ(y)]$;
4. $\forall x[x + 0 = x]$;
5. $\forall x, y\,[x + Succ(y) = Succ(x + y)]$;
6. $\forall x\,[x \cdot 0 = 0]$;
7. $\forall x, y\,[x \cdot Succ(y) = x \cdot y + x]$;

This axioms encompass the primitive recursive definitions of the basic functions but they fall short for being able to prove $\forall x\,[x \neq Succ(x)]$.

To see this, consider the domain $\mathbb{N} \cup \{\infty\}$ with $\infty = Succ(\infty) = \infty + \infty = \infty \cdot \infty = \infty + n = n + \infty = m \cdot \infty = \infty \cdot m$ where n is any member of $\mathbb{N}$ and m is any nonnull member of $\mathbb{N}$.

The System PA$^-$

0. $x + 0 = x$;
1. $x + y = y + x$;
2. $(x + y) + z = x + (y + z)$;
3. $x \cdot 1 = x$;
4. $x \cdot y = y \cdot x$;
5. $(x \cdot y) \cdot z = (x \cdot y) \cdot z$;
6. $x \cdot (y + z) = x \cdot y + x \cdot z$;
7. $Succ(x) = x + 1$;
8. $x + y = x + z \to y = z$;
9. $x \leq y \vee y \leq x$;
10. $x \leq 0 \to x = 0$;
11. $x < y \Leftrightarrow Succ(x) \leq y$;

The universal closure of this axiom system generates the theory PA$^-$ and it implies Robison's arithmetic Q.

PA is PA$^-$ plus $\alpha\frac{0}{x} \wedge \forall x(\alpha \to \alpha\frac{Succ(x)}{x}) \to \forall x\,(\alpha)$ for every α.

Encoding Numerical Constants

In the following let $\underline{0}$ be the value denoted by 0, $\underline{1}$ denote $Succ(\underline{0})$, $\underline{2}$ denote $Succ(\underline{1})$ what is $Succ(Succ(\underline{0}))$ and $\underline{n+1}$ denote $Succ(\underline{n})$.

One can prove all the usual properties of natural numbers from PA^{-}.

0. $Succ(x) + \underline{n} = x + Succ(\underline{n})$;
1. $\underline{m} + \underline{n} = \underline{m+n}$ and $\underline{m} \cdot \underline{n} = \underline{m \cdot n}$;
2. $\underline{m} \neq \underline{n}$ whenever $m \neq n$;
3. $\underline{m} \leq \underline{n}$ whenever $m \leq n$;
4. $\underline{m} \not\leq \underline{n}$ whenever $m \not\leq n$;
5. $x \leq \underline{n} \leftrightarrow x = \underline{0} \vee x = \underline{1} \vee \ldots \vee x = \underline{n}$;
6. $x \leq \underline{n} \vee \underline{n} \leq x$.

Tutorial Question 5.1 (a)

The set X formalises that f is a function from A to A such that for all $x, y, v, w \in A$ with $P(x), P(y), P(v), P(w)$, if $(x, y) \neq (v, w)$ then $f(x, y) \neq f(v, w)$. Furthermore, if $x, y \in A$ with $P(x), P(y)$ then $\neg P(f(x, y))$ and every $z \in A$ with $\neg P(z)$ equals to $f(x, y)$ for some $x, y \in A$ with $P(x), P(y)$. Thus each pair of values in A satisfying P corresponds to one element of A not satisfying P and so $m = n^2$.

Tutorial Question 5.1 (b)

The new set Y has the modification that one has to identify unordered pairs of elements satisfying P with elements satisfying $\neg P$ via f.

- $\exists x\,[Px]$;
- $\forall x, y, v, w\,[(Px \wedge Py \wedge Pv \wedge Pw \wedge ((x \neq v \wedge x \neq w) \vee (y \neq v \wedge y \neq w) \vee (v \neq x \wedge v \neq y) \vee (w \neq x \wedge w \neq y)) \rightarrow f(x, y) \neq f(v, w)]$;
- $\forall v, w\,[Pv \wedge Pw \rightarrow \neg P(f(v, w))]$;
- $\forall u \exists v, w\,[\neg Pu \rightarrow Pv \wedge Pw \wedge f(v, w) = u \wedge f(w, v) = u]$.

Tutorial Question 5.2

The set X contains for each m, n a formula saying the following statement: For all $x_1, \ldots, x_m$ and all $y_1, \ldots, y_n$ there is a z such that if all the x_i are different from all the y_j then z is connected to all x_i and to none of the y_j.

By using the shorthand $\wedge_{i=1,\ldots,m}$ for the conjunction of m terms and correspondingly for disjunctions, the formula can be formalised as follows.

$\alpha_{m,n}$ is the formula $\forall x_1 \ldots \forall x_m \, \forall y_1 \ldots \forall y_n \, \exists z$
$[(\wedge_{i=1,\ldots,m} \wedge_{j=1,\ldots,n} \, (x_i \neq y_j)) \rightarrow$
$\wedge_{i=1,\ldots,m} \wedge_{j=1,\ldots,n} \, (E(x_i, z) \wedge \neg E(y_j, z))])$.

X consists of all formulas $\alpha_{m,n}$ with $m, n \geq 1$. In order to get an undirected graph, the formula $\forall v, w \, [E(v, w) \leftrightarrow E(w, v)]$ has to be added.

Tutorial Question 5.3

The axioms are the following.

$\forall x \in R_1\,[x \in R_2]$; $0 \in R_1$; $1 \in R_1$; $0 \neq 1$;

All axioms from the lecture for non-commutative rings for R_2 without the multiplicative inverse (associativeness of $+$ and $\cdot$, distributive laws from both sides, neutral elements for $+$ and $\cdot$ in R_2, commutativeness of $+$, existence of inverse for $+$, $e_0 + e_3 = 1$);

The axiom which says that R_1 is closed under $+$ and $\cdot$;

The axiom $\forall x \in R_1 \forall y \in R_2\,[x \cdot y = y \cdot x]$;

The axioms
$\forall x \in R_2 \exists y_0, y_1, y_2, y_3 \in R_1\,[x = e_0 \cdot y_0 + e_1 \cdot y_1 + e_2 \cdot y_2 + e_3 \cdot y_3]$
and $\forall x \in R_1\,[x = (x \cdot e_0) + (x \cdot e_3)]$;

Continuation of Axioms

The axiom
$\forall y_0, y_1, y_2, y_3, x_0, x_1, x_2, x_3 \in R_1\, [e_0 \cdot y_0 + e_1 \cdot y_1 + e_2 \cdot y_2 + e_3 \cdot y_3 = e_0 \cdot x_0 + e_1 \cdot x_1 + e_2 \cdot x_2 + e_3 \cdot x_3 \rightarrow x_0 = y_0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge x_3 = y_3]$;

The axiom $\forall y_0, y_1, y_2, y_3 \in R_1$ $[det(e_0 \cdot y_0 + e_1 \cdot y_1 + e_2 \cdot y_2 + e_3 \cdot y_3) = y_0 \cdot y_3 - y_1 \cdot y_2]$;

The 16 axioms defining the multiplication rules for the four generators: $e_0 \cdot e_0 = e_0$, $e_0 \cdot e_1 = e_1$, $e_0 \cdot e_2 = 0$, $e_0 \cdot e_3 = 0$, $e_1 \cdot e_0 = 0$, $e_1 \cdot e_1 = 0$, $e_1 \cdot e_2 = e_1$, $e_1 \cdot e_3 = e_1$, $e_2 \cdot e_0 = e_2$, $e_2 \cdot e_1 = e_3$, $e_2 \cdot e_2 = 0$, $e_2 \cdot e_3 = 0$, $e_3 \cdot e_0 = 0$, $e_3 \cdot e_1 = e_2$, $e_3 \cdot e_2 = 0$, $e_3 \cdot e_3 = e_3$.

Tutorial Question 5.4

The goal is to make an isomorphism between two countable structures $(\{a_0, a_1, \ldots\}, <, 0, 1)$ and $(\{b_0, b_1, \ldots\}, <, 0, 1)$. For this, assume that $a_0 = 0, a_1 = 1$ and $b_0 = 0, b_1 = 1$ in the respective structures. Now let $f(a_0) = b_0$, $f(a_1) = b_1$ and, for $n \geq 2$, let $f(a_n)$ be b_m for the least m such that for all $k < n$ it holds that $a_k < a_n$ implies $f(a_k) < b_m$ and $a_n < a_k$ implies $b_m < f(a_k)$.

It is easy to see that f preserves the relation $<$ and f is total: As the ordering $(\{b_0, b_1, \ldots\}, <, 0, 1)$ is dense, there is for every a_n a value b_m found this way and one can easily see by induction that for all i, j with $i < j$ it holds that $a_i < a_j$ implies $f(a_i) < f(a_j)$ and $a_j < a_i$ implies $f(a_j) < f(a_i)$.

Continuation

Furthermore, assume now that some b_m would be left out, without loss of generality, m is the least such index. It happens infinitely often that $f(a_k)$ needs to be defined for an a_k such that all $h < k$ satisfy $a_h < a_k \Rightarrow f(a_h) < b_m$ and $a_h > a_k \Rightarrow f(a_h) > b_m$; otherwise there would be a_i and a_j such that $f(a_i) < b_m$ and $f(a_j) > b_m$ and no a_k with $a_i < a_k < a_j$ would be in the list in contradiction to the denseness. Now, from some time onwards, m will be for those k as indicated above the least index such that b_m is not yet in the range of f and therefore one of these a_k will be mapped to b_m. Therefore the function f is an order isomorphism.