

**CPTS 355–COMPUTABILITY NOTIONS
UPDATED SEPT 7, 2005**

Basics: Functions, total functions and partial functions. A function, $f : A \rightarrow B$ is a set of ordered pairs, $\langle a, b \rangle \subseteq A \times B$, such that

- (1) If $\langle x, y \rangle \in f$ and $\langle x, z \rangle \in f$ then $y = z$.
- (2) For every $x \in A$ there exists $y \in B$ with $\langle x, y \rangle \in f$

Rules 1 and 2 define a class of functions called the *total functions from A to B*. We also say that the *type* of f is $A \rightarrow B$. If the second rule is dropped, we have the definition for the class of *partial functions from A to B*. Examples of partial functions: division, tangent, ...? Notice that the class of partial functions includes the class of total functions.

Computer programs are often (but not always) thought of as computing partial functions. When would it be inappropriate to consider a computer program as computing a partial function?

Partiality can arise in a program in two ways in computing a function: by the occurrence of an error on a particular input or by the failure to halt (divergence) on an input.

Example of an error:

```
fun g (x:real, y:real) : real = x/y
```

What is the type of this function? ($(real \times real) \rightarrow real$). On what values is it defined? (pairs of real numbers such that the second is not zero)

Example of divergence:

```
fun f (x:int) : int = if x=0 then 0 else x+f(x-2)
```

What is the type of this function? ($int \rightarrow int$). On what values is it defined? (non-negative even integers). What happens if we try to evaluate f on an odd or negative integer? (divergence)

(Problem solving: give a closed-form definition of the function)

Computability. Is it possible to write down a program to compute *any* partial function having, say, type $\text{nat} \rightarrow \text{nat}$? Problems closely related to this puzzled mathematicians of the late 19th and early 20th centuries. In the 1930s Kurt Godel solved (in the negative) the related problem of whether a sufficiently powerful system of logic could be both consistent and complete. Shortly after that, Alan Turing proved that for a class of simple computers now called Turing machines there could be no program to compute certain functions. Furthermore, the class of Turing machines is as powerful as any other way of writing down rules for computation that has been conceived thus far.

A Turing machine consists of a fixed alphabet, an infinite bi-directional tape, a read-write head and a *finite control*. Draw diagram). Execution consists of repeating the sequence:

- considering the current state (of the finite control) and the symbol on the tape under the read-write head
- write a new symbol on the tape at the current location, move the head left or right one position or leave it where it is and move to a new state in the finite control.

That's all. The behavior of a Turing machine is entirely specified by specifying the fixed alphabet and the finite control.

Church's hypothesis, due to mathematician Alonzo Church, conjectures that *any* procedure that could reasonably be called a program can be realized on a Turing machine. Equivalence of the Turing machine class to other proposed expressions of programs has been proved repeatedly over the intervening years and Church's hypothesis is now generally accepted as true. The class of partial functions that can be computed by any of these mechanisms is called the class of *computable functions*. Computable functions \subset Partial functions.

Why should we care? First, as noted in the book, programming languages are usually designed to be *Turing complete* in the sense that in principle they can be used to write a program to compute any computable function. (Only *in principle* because programs are executed with finite time and space resources whereas computation of some functions will require arbitrarily amounts of time and space.) One way to test the Turing completeness of a language is to ask whether you can write a Turing machine simulator (interpreter) in the language—essentially the same thing as our little exercise in writing an interpreter for PS.

Second, it turns out that some of the uncomputable functions are precisely ones that we would find useful in programming! In particular the so-called

halting problem that Turing used in his proof of uncomputability would be a very handy thing to have a solution for (if it could be implemented, which it can't!).

The halting problem: Is there a program that when given another program $P : \text{string} \rightarrow \text{string}$ and $s : \text{string}$ as input can *always* (i.e., for any P and any s) determine whether P halts when given s as *its* input.

Said another way: “is there a program that when given a pair consisting of a description of a program, P , and a string, s , will return “true” if $P(s)$ halts and “false” if $P(s)$ does not halt. Notice that the type of this hypothetical program, if it existed, would be $\text{Program} \times \text{string} \rightarrow \text{boolean}$ and that it would be a total function. (Why?) Turing proved that no such program could exist.

Proof of the undecidability of the halting problem. The proof proceeds by contradiction. Suppose such a program exists. Call it Q :

$$Q(P,x) = \begin{cases} \text{true} & P(x) \text{ halts} \\ \text{false} & P(x) \text{ doesn't halt} \end{cases}$$

Now let's define another program using Q and our program f from before as subroutines:

```
fun D (P: program) : int = if Q(P,P) then f(-1) else 0
```

Notice that if P halts with its own description as its input then D doesn't halt and *vice versa*. Now consider $D(D)$. Let's expand it according to the definition:

```
D(D) = if Q(D,D) then f(-1) else 0
      = if D(D) halts then f(-1) /* doesn't halt */ else 0 /* halts */
```

but this is clearly an absurdity: we've just shown that if $D(D)$ halts then it doesn't halt and if $D(D)$ doesn't halt then it does. We've made two assumptions: that programs can be represented as strings so that a program that processes strings can be given a program as input, and that program Q exists. Turing machines can certainly be represented as strings, so our mistake must be in assuming the existence of Q .

(Note that on p. 15 of the book the presentation of this argument is confused (though not made incorrect) by the type used for Q at the top of the page. In that definition “halts” and “does not halt” are *strings*, not statements about the behavior of Q !)

Consequences.

- (1) Variants of this proof can be applied to programs that take integers as input as well as strings. Indeed, (see problem 2.2) the proof can be formulated for programs that take no input at all. You can also think about a piece of a program as being itself a program and ask about whether execution of a program reaches a particular point in its text (also undecidable).
- (2) A compiler can't give you an error message iff your program doesn't halt. It may be able to give you an error for some cases of not halting. And it may be able to give you a warning asserting that *maybe* the program doesn't halt.
- (3) Memory management: a compiler cannot in every case insert static calls to automatically free memory when it is no longer reachable.
- (4) Notice that the implications of the undecidability result occur when we ask for a *universal* solution (for all programs). In any particular case we may be able to marshal resources to prove halting or non-halting – and indeed we do so all the time when we program!