UNIMPLEMENTABLE SPECIFICATIONS

This material is covered in Chapter 12 of the textbook.

There exist specifications that cannot be implemented because the specified function is *uncomputable.* This is possible even for well-defined and consistent specifications!

The Halting Problem

We want to implement a function HALTS that receives file parameters `func` and `arg` and

1. HALTS returns `true` if the file `func` contains a function definition with one file parameter, and the function terminates if applied to file `arg`.
2. HALTS returns `false` otherwise.

Theorem. HALTS cannot be implemented in C
(that is, the function HALTS is uncomputable).

The function HALTS cannot be implemented in any (currently used) programming language. More generally, it is not *effectively computable* (or algorithmically computable).

In a general setting, a computational problem consists of

1. a potentially infinite set of inputs, and
2. a function that associates an output to each input.[1]

Since computers are finite, we usually require that each individual input has a finite representation, that is, can be encoded as a finite string. Similarly, each individual output is

[1]In some computatitional problems, for a given input we may allow several acceptable otuputs. To formalize this kind of computational problems we need to use a relation instead of a function.

required to have a finite representation. This is not a restriction in practice since all natural computational problems can be encoded in this way.

A *solution* to a computational problem is then an algorithm that for each input finds (computes) the correct output.

The fact that the function HALTS is uncomputable is not caused by any "defect" in the programming language C. Exactly the same argument works with any other programming language!

- It has been shown that all the standard programming languages are computationally equivalent in the sense that the same class of functions can be implemented in each of them.
- Furthermore, the functions that can be implemented in some programming language (C, Java, ...) are exactly the functions that can be computed by so called Turing machines. Turing machines are a formal model of algorithms originally developed by Alan Turing in the 1930's.

Church-Turing thesis: The functions computable by Turing machines are exactly all the functions that can be effectively computed by some informal algorithm.

- It is generally believed that the Church-Turing thesis holds. However, it is impossible to prove that it is correct since the notion of "informal algorithm" is not precisely defined.

Note: The fact that Turing machines compute the same functions that can be implemented in C (or Java, Lisp, etc.) can be proved. This also establishes that the different programming languages can implement exactly the same class of functions.

The technique we used to show that HALTS is uncomputable[2] is called diagonalization.

The technique of diagonalization allows us to establish the uncomputability of specific algorithmic problems. Comparing the "numbers"[3] of all possible programs and of all algorithmic problems, we can establish that, in fact, "most" algorithmic problems are unsolvable. All possible programs can be listed as an infinite unending list P_1, P_2, ... On the other hand, the set of algorithmic problems is "as large" as the set of real numbers.

Reducibility

Since the diagonalization argument is fairly complicated, it would not be convenient to use it directly for each new computational problem we suspect to be unsolvable. Once we have some problem (function) that is known to be uncomputable, we can show other problems to be uncomputable using a technique called reduction.

An algorithmic problem A is said to be reducible to a problem B, if a solution (algorithm) for B can be used to solve A.

When problem a problem A is reducible to problem B, we can also say that

- A is easier than or as hard as B, or,
- A is not harder than B.

Now the following holds:

1. If A is uncomputable, and
2. A is reducible to B

[2]or unsolvable

[3]In more precise terminology, we compare the cardinality of the set of all programs and the cardinality of the set of algorithmic problems.

then also B is uncomputable.

Why?

Thus, if problem A is reducible to problem B, the following three situations are possible:

- A and B are are both computable,
- A is computable and B is uncomputable,
- both A and B are uncomputable.

Note that if A is reducible to B, it is not possible that B is computable and A is uncomputable.

The above general definition of reducibility can be used to establish algorithmic uncomputability. For more precise comparisons of the hardness (or complexity) of algorithmic problems we use *mapping reductions* that transform instances of one problem to instances of another problem.

Rice's theorem

For computing problems related to program testing, we can distinguish between *syntactic* and *semantic* problems. Syntactic problems are, roughly speaking, related to the correct representation of a program in the formalism of a given programming language, and these problems are usually algorithmically solvable. Semantic problems are related to the meaning of programs. Examples of semantic problems include:

- Does a program accept a given input? (that is, produce the answer “yes” for a given input)
- Does a program satisfy a given specification?
- Does a program halt on a given input, or does a program halt on all inputs?

By a *decision problem* we mean a problem where the answer is simply “yes” or “no”.

The so called *Rice’s theorem* says that all non-trivial semantic decision problems for programs are unsolvable.

A (semantic) decision problem is said to be non-trivial if there is an input program for which the answer is “yes” and another input program for which the answer is “no”. Note that the requirement is very weak, and any decision problems considered in practice are non-trivial in the above sense.

For trivial decision problems the answer is the same for all inputs. What would be an example of a trivial semantic problem?