15–150: Principles of Functional Programming

Some Notes on Computability Theory

Michael Erdmann*

Spring 2025

1 Introduction

Computers can be programmed to perform an impressive variety of tasks, ranging from numerical computations to natural language processing. The computer is a very versatile tool, playing a critical role in solving all manner of "real world" problems. Some would argue that computers can solve any problem that a human can solve; some would argue the opposite; and some regard the question as irrelevant. Whatever view one adopts, it is still interesting to consider whether there are any limits to what can be solved by a computer.

Any given computer has only a finite memory capacity, so certainly there are problems that are too large for it to solve. We abstract away from these limitations, and consider the question of what can be solved by an *ideal* computing device, one which is not limited in its memory capacity (but is still required to produce an answer in a finite amount of time). *Computability theory* is concerned with exploring the limitations of such idealized computing devices. This is different from *complexity theory*, which we considered in our work and span analyses, that is concerned with calibrating the resources required to solve a problem.

Our treatment of computability theory is based on problems pertaining to SML programs. We begin by considering questions about SML *functions* such as "does the function f yield a value when applied to an input x?" or "are functions f and g equal for all inputs?". It would be handy to build a debugging package that included SML programs to answer these (and related) questions for us. Can such a package be built? You may well suspect that it cannot, but how does one prove that this suspicion is well-founded?

2 Properties of Functions

2.1 Decision Problems and Decision Procedures

A *decision problem* is a well-defined question about well-specified data (called *instances* of the problem) that has a "yes" or "no" answer. For example, the *primality problem* is a decision problem, namely to decide whether or not a given natural number n is prime. A decision problem is *decidable* (or *solvable* or *computable*) iff there is an SML function that, when applied to an instance of the problem, evaluates to either `true` or `false` according to whether or not the answer

*Adapted from a document by Robert Harper.

to the instance is "yes" or "no". Such a function, when it exists, is called a *decision procedure*. The primality problem is decidable: there is an SML function `is_prime` of type `int->bool` such that for every natural number n, `is_prime` n evaluates to `true` iff n is prime, and evaluate to `false` otherwise. We will show that there are *undecidable* problems — ones for which no SML program can decide every instance.

It is important to stress that the whole question of decidability can only be considered for well-posed problems. In particular, it must be absolutely clear what are the problem instances and how they are to be represented as input to an SML program. For example, in the case of the primality problem we are representing a natural number as an SML value of type `int`. (We could also represent it as a string, or as a list of Booleans corresponding to its binary representation.) Questions such as "is sentence s grammatical according to the rules of English grammar?" are not well-posed in this sense because it is not clear what is the grammar of English, and so it is never clear whether a given sentence is grammatical or not. Many fallacious arguments hinge on the fact that the "problem" under consideration is so ill-defined as to render it meaningless to ask whether or not a computer may be used to solve it.

2.2 The Halting Problem

The fundamental result of computability theory is the *unsolvability of the halting problem*: given a function f of type `int->int` and an input x of type `int`, does f evaluate to a value on input x? That is, does f *halt* on input x? (If f on input x raises an unhandled exception, then we do not regard it as halting. In both that case and the case of infinite looping, we say that $f(x)$ *diverges*.) The halting problem for functions is *undecidable*:

Theorem 1 *There is no SML function H of type `(int->int)*int->bool` such that for every f of type `int->int` and every x of type `int`,*

(1) H`(`f`,` x`)` evaluates to either `true` or `false`.

(2) H`(`f`,` x`)` evaluates to `true` iff $f(x)$ evaluates to a value.

Proof: Suppose, for a contradiction, that there were such an SML function H. Consider the following function of type `int->int`:

```
fun diag(x:int):int = if H(diag, x) then loop () else 0.
```

Here `loop` is the function defined by the declaration

```
fun loop () = loop ().
```

(So, `loop ()` runs forever.)

Now consider the behavior of H`(diag, 0)`. (There is nothing special about `0`; we could as well choose any number.) By condition (1), either H`(diag, 0)` evaluates to `true` or H`(diag, 0)` evaluates to `false`. We show that in either case we arrive at a contradiction. It follows that there is no such SML function H.

1. Suppose that H`(diag, 0)` evaluates to `true`. Then, by condition (2), we know that `diag 0` halts. However, by inspecting the definition of `diag`, we see that `diag 0` halts only if H`(diag, 0)` evaluates to `false`! Since `true` is not equal to `false`, we have a contradiction.

2. Suppose that H`(diag, 0)` evaluates to `false`. Then, by condition (2), we know that `diag 0` does not halt. However, by inspecting the definition of `diag`, we see that this happens only if H`(diag, 0)` evaluates to `true`, again a contradiction.

□

It is worthwhile to contemplate this theorem and its proof very carefully. The function `diag` used in the proof is said to be defined by *diagonalization*, a technique introduced by Georg Cantor in his proof of the uncountability of the real numbers. The idea is that `diag` calls H on itself, and then "does the opposite" — if H`(diag, x)` evaluates to true, `diag` goes into an infinite loop, and otherwise terminates immediately. Thus `diag` is a demonic adversary that tries (and succeeds!) to refute the existence of a function H satisfying the conditions of the theorem.

Note that the proof relies on *both* conditions about H. Dropping the second condition renders the theorem pointless: of course there are SML functions that always yield either `true` or `false`. But suppose we drop the first condition instead. Is there an SML program H satisfying *only* the second condition? Of course! It is defined as follows:

```
fun H(f, x) = (f x; true)
```

H`(`f`, `x`)` evaluates to `true` iff $f\,x$ halts, and that is all that is required. The function H so defined is sometimes called a *semi-decision procedure* for the halting problem because it yields `true` iff the given function halts when called with argument x, but may give no answer otherwise.

2.3 Proof by Reduction

The unsolvability of the halting problem can be used to establish the unsolvability of a number of related problems about functions. The idea is to show that a problem P is unsolvable by showing that if P were solvable, then the halting problem would also be solvable. This is achieved by showing that an SML function to decide the halting problem can be defined if we are allowed to use an SML function to decide P as a "subroutine". In this way we *reduce* the halting problem to the problem P by showing how instances of the halting problem can be "coded up" as instances of problem P. Since an SML function to solve the halting problem does not exist, it follows that there cannot exist an SML function to solve P. Here are some examples:

I. Is there an SML function Z of type `(int->int)->bool` such that $Z(f)$ evaluates to `true` iff $f(0)$ halts, and evaluates to `false` otherwise? That is, is the "halts on zero" problem decidable? No. Here is a tempting, but incorrect, attempt at a proof:

> Clearly, $Z(f)$ evaluates to true iff H`(`f`, 0)`, so Z must be undecidable. So we can define Z by `fun` Z`(`f`)=`H`(`f`,0)`. Hence no such Z can exist.

But this is backwards! A correct proof proceeds by showing that *if* there were an SML function Z satisfying the conditions given above, *then* there would exist an SML function H solving the halting problem. Stated contrapositively, if no function H solving the halting problem exists, then no function Z solving the "halts on zero" problem exists. Since we've already shown there is no such H, then there is no such Z. To complete the proof, we must show how to define H from Z. This is done as follows: `fun` H`(f,x) =` Z`(fn 0 => (f x))`. The function `(fn 0 => (f x))` halts on input `0` iff `f` halts on input `x`, so the proposed definition of H in terms of Z solves the halting problem, and we have our desired contradiction.

II. By a similar pattern of reasoning we may show that the halting problem for suspensions is undecidable. More precisely, there is no SML function S of type `(unit->int)->bool` such that

for every s of type `unit->int`, the application $S(s)$ evaluates to `true` iff s`()` halts, and evaluates to `false` otherwise. Suppose there were such an S. Then we may define a procedure H to solve the halting problem as follows: `fun` H`(f,x) =` S`(fn () => (f x))`. (Convince yourself that this definition refutes the existence of S as described.)

III. Consider the following problem: given an SML function f of type `int->int`, is there *any* argument x of type *int* such that $f(x)$ halts? This problem is also undecidable. For if F were an SML function of type `(int->int)->bool` solving this problem, then we could define an SML function to solve the halting problem as follows:

```
fun H(f, x) =
    let
       fun g y = (f(x); y)
    in
       F g
    end.
```

Note that the function `g` has the property that it halts on some input only if `f` halts on `x`.

It is worth pointing out that there is nothing special about the type `int` in the above arguments. The proofs would go through for any type τ, provided that there is a value v of that type. (In the above cases we took $\tau =$ `int` and $v =$ `0`.)

IV. A type in SML for which there is an equality test function is said to *admit equality*. For example, the types `int` and `string` admit equality. But not every type admits equality. For example, there is no equality test for values of type `int->int`. Is this just an oversight? No.

The *equality problem* for functions of type `int->int` is to decide whether or not two (pure) functions `f` and `g` of this type are extensionally equivalent, as previously defined in the course.

The equality problem is undecidable:
There is no SML function E of type `(int->int)*(int->int)->bool` such that E`(f,g)` evaluates to `true` iff `f` is extensionally equivalent to `g`, and evaluates to false otherwise. Suppose there were such an E. Then we may define a function H to solve the halting problem as follows:

```
fun H(f, x) = E (fn y:int => (f(x); y),  fn (y:int) => y)
```

Notice that the two functions in the call to E are extensionally equivalent iff `f(x)` halts. Thus H solves the halting problem, which is a contradiction. Thus we see that the limitation on equality in SML is a feature, rather than a bug!

3 Church's Thesis

To complete our discussion of computability, we consider the generality of our results. So far we have demonstrated that several properties regarding SML functions are undecidable. Perhaps this is an artifact of the representation. Might it be possible to decide halting for functions if we are given the actual program, rather than just a "black box" function? It may seem plausible, at first sight, since, after all, we as programmers make such judgments based on the program itself, so why might not a computer be able to do the same thing? And if computers have the same capabilities as people (as some would say), then perhaps computers can do this too. One problem with this argument is that it is far from clear that people can decide halting for *arbitrary* functions, even

given the code: the program might be so complicated as to overwhelm even the most clever among us. Be that as it may, it is possible to prove that halting remains undecidable, even if the program is given as input to the halting tester. The proofs are very similar to the ones we have already given.

The possibility remains, however, that all of this is an artifact of specifically SML. Might it perhaps be possible to decide halting of representations of C functions as C data structures? Or of Java applets represented as Java data structures?

The answer is "no" because each of these languages is capable of simulating the other. That is, we can write an SML interpreter for C code and a C interpreter for SML code (but it wouldn't be much fun). Therefore, their halting problems are equally unsolvable: a halting checker for C could be used to build a halting checker for SML by asking about the behavior of the SML interpreter written in C.[1] The point is that all of these languages are *Turing equivalent* which means that they compute the exact same functions over the natural numbers, namely the *partial recursive functions*.

What about the languages of the future? So far no one has invented a programming language that can be executed by a computer that is more powerful (in the sense of representing number-theoretic functions) than the languages we all know. Might there be one someday? Who knows? *Church's Thesis* is the claim that this will never happen — according to Alonzo Church, the great logician and mathematician, the very idea of *computable* function on the numbers coincides with the *partial recursive* functions on the natural numbers. That is, all programming languages are of the same expressive power when it comes to functions on the natural numbers.

What about functions on other types? It is easy to see that any types whose values are themselves finite (*e.g.*, lists of integers, or lists of lists of pairs of integers, *etc.*) may be coded up as integers by some sort of Gödelization scheme. So we cannot hope to make progress here. But what about functions on the real numbers? Is there a sensible notion of computability for inherently infinite objects such as the reals? And are all such notions equivalent? These and related questions are a fascinating part of the extension of computation theory to higher types, all of which lie far beyond the scope of these notes.

4 Conclusion

We see that there *are* limitations to what can be computed. What are the practical implications of these limitations? An immediate consequence is that we should not expect too much from compilers. For example, it is undecidable whether or not a given conditional branch (say, to the `else` clause) will ever be taken in a given program. (It is easy to devise a program for which a given `if` expression takes the `else` branch iff a given program halts on a given input.) Consequently, the compiler must allow for the possibility that either branch may be taken, and must refrain from making optimizations that rely on knowing the outcome. Nor is it possible to determine whether a given variable will ever take on values greater than, say, 255, limiting the possibilities for the compiler to represent it as a byte rather than a full word. In fact *any* non-trivial property of the execution behavior of programs is undecidable (this can be proved!). On the one hand, this is disappointing — only so much can be automated. On the other hand, it is a "full employment" guarantee for compiler writers — since the ultimate solution is not programmable, there is always room to improve the partial solutions that *are* programmable.

[1] We are glossing over a few details here, but this is the general idea.