15–212: Fundamental Structures of Computer Science II

Basic Computability Theory

Robert Harper, Spring 1997
edited by Frank Pfenning, Fall 1997

Draft of November 18, 1997

1 Introduction

Computers can be programmed to perform an impressive variety of tasks, ranging from numerical computations to natural language processing. The computer is surely one of the most versatile tools ever devised, 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. *Complexity theory* (which we shall not study here) is concerned with calibrating the resources (both time and space) required to solve a problem.

Our treatment of computability theory is based on problems pertaining to ML programs. We begin by considering questions about ML *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 ML 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?

We then go on to consider questions about ML *programs* (presented as values of a `datatype` representing the abstract syntax of ML). The difference lies in the fact that in ML functions are "black boxes" — we can apply them to arguments, but we can't look inside the box. "Of course," you might think, "one can't test convergence of a *function* on a given input, but what if it were possible to look at the *code* of the function? What then? Maybe then one can decide convergence on a given input." Unfortunately (or fortunately, depending on your point of view), the problem remains undecidable.

2 Properties of Functions

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 ML function that, when applied to an instance of the problem, evaluates to either `true` or `false` according to whether or not the answer to the instance is "yes" or "no". The primality problem is decidable: there is an ML 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 ML 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 ML program. For example, in the case of the primality problem we are representing a natural number as an ML 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!

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 uncaught exception, then we do not regard it as halting.) The halting problem for suspensions is *undecidable*:

Theorem 1 *There is no ML 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 ML function H. Consider the following function of type `int->int`:

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

Here `loop` is the function defined by the declaration

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

(Obviously `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 our first assumption, 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 ML function H.

1. Suppose that H `(diag, 0)` evaluates to `true`. Then by the second assumption we know that `diag, 0` halts. But 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 the second assumption we know that `diag, 0` does not halt. But 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 the 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` 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* assumptions about H. Dropping the second assumption renders the theorem pointless: of course there are ML functions that always yield either `true` or `false`! But suppose we drop the first assumption instead. Is there an ML program H satisfying *only* the second assumption? Of course! It is defined as follows:

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

Clearly 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 suspension halts when forced, but may give no answer otherwise.

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 ML function to decide the halting problem can be defined if we are allowed to use an ML 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 ML function to solve the halting problem does not exist, it follows that there cannot exists an ML function to solve P. Here are some examples.

Is there an ML 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 ML function Z satisfying the conditions given above, *then* there would exist an ML 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. But this is easy: `fun` H `(f,x) =` Z `(fn 0 => (f x))`. Notice that 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, contradiction.

By a similar pattern of reasoning we may show that the halting problem for suspensions is undecidable. More precisely, there is no ML function S of type `(unit->int)->bool` such that for every t of type `unit->int`, the application $S\,t$ evaluates to `true` iff t `()` 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.)

Consider the following problem: given an ML 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 ML function of type (int->int)->bool solving this problem, then we could define an ML 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.)

A type in ML for which there is an equality test function is said to *admit equality*. For example, the types int, real, and string all 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? First let us be clear what we mean by equality of such functions. If f and g are functions of type int->int then f is equal to g iff for every input x of type int, either $f\,x$ and $g\,x$ both diverge, or both evaluate to the same value of type int. Thus fn x:int=>2*x and fn x:int=>x+x are equal functions of type int->int, as are fn x:int=>loop() and the function f defined by fun f(x:int)=f(x).

The *equality problem* for functions of type int->int is to decide whether or not two functions f and g of this type are equal in the sense just described. The equality problem is undecidable: there is no ML function E of type (int->int)*(int->int)->bool such that E(f,g) evaluates to true iff f is equal 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 equal iff f x halts. Thus H solves the halting problem, which is a contradiction. Thus we see that the limitation on equality in ML is feature, rather than a bug!

3 Properties of Programs

Is it possible to decide halting for suspensions if we are given the actual program, rather than just a "black box"? It may seem plausible, at first sight, since, after all, we as programmers make such judgements 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's far from clear that people can decide halting for *arbitrary* suspensions, 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.

Recall that we defined an interpreter for a small fragment of ML, called Mini-ML, written in full ML.[1] The implementation consisted of two main parts. First, we defined a datatype called exp for the abstract syntax of Mini-ML and a datatype called value for the values of Mini-ML. Then

[1]see the code for lecture 23

Expressions

$$
\begin{array}{rcl}
\ulcorner x \urcorner & = & \texttt{Var "}x\texttt{"} \\
\ulcorner n \urcorner & = & \texttt{Integer } n \\
\ulcorner e_1 \texttt{+} e_2 \urcorner & = & \texttt{Plus (}\ulcorner e_1 \urcorner\texttt{, }\ulcorner e_2 \urcorner\texttt{)} \\
\ulcorner \texttt{true} \urcorner & = & \texttt{True} \\
\ulcorner \texttt{false} \urcorner & = & \texttt{False} \\
\ulcorner \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \urcorner & = & \texttt{IfThenElse (}\ulcorner e_1 \urcorner\texttt{,}\ulcorner e_2 \urcorner\texttt{,}\ulcorner e_3 \urcorner\texttt{)} \\
\ulcorner \texttt{fn } x \texttt{ => e} \urcorner & = & \texttt{Fn ("}x\texttt{", }\ulcorner e \urcorner\texttt{)} \\
\ulcorner e_1\, e_2 \urcorner & = & \texttt{App (}\ulcorner e_1 \urcorner\texttt{, }\ulcorner e_2 \urcorner\texttt{)} \\
\ulcorner \texttt{let}\, d \,\texttt{in}\, e \,\texttt{end} \urcorner & = & \texttt{Let (}\ulcorner d \urcorner\texttt{, }\ulcorner e \urcorner\texttt{)}
\end{array}
$$

Declarations

$$
\begin{array}{rcl}
\ulcorner . \urcorner & = & \texttt{Null} \\
\ulcorner d \texttt{ val } x \texttt{ = } e \urcorner & = & \texttt{Dec (}\ulcorner d \urcorner\texttt{, ("}x\texttt{", }\ulcorner e \urcorner\texttt{))} \\
\ulcorner d \texttt{ val rec } x \texttt{ = } e \urcorner & = & \texttt{Rec (}\ulcorner d \urcorner\texttt{, ("}x\texttt{", }\ulcorner e \urcorner\texttt{))}
\end{array}
$$

Values

$$
\begin{array}{rcl}
\ulcorner r \urcorner & = & \texttt{Rational } r \\
\ulcorner b \urcorner & = & \texttt{Boolean } b \\
\ulcorner \{\eta; e\} \urcorner & = & \texttt{Closure (}\ulcorner \eta \urcorner\texttt{, }\ulcorner e \urcorner\texttt{)}
\end{array}
$$

Value Environments

$$
\begin{array}{rcl}
\ulcorner . \urcorner & = & \texttt{Null} \\
\ulcorner \eta, x = v \urcorner & = & \texttt{Dec (}\ulcorner \eta \urcorner\texttt{, ("}x\texttt{", }\ulcorner v \urcorner\texttt{))} \\
\ulcorner \eta, \mathit{rec}\, x = e \urcorner & = & \texttt{Rec (}\ulcorner \eta \urcorner\texttt{, ("}x\texttt{", Freeze }\ulcorner e \urcorner\texttt{))}
\end{array}
$$

Figure 1: Quotation for Mini-ML

we defined a function `eval` of type `value env * exp -> value` that, given the representation of a value environment and an ML expression as a value of type `value env * exp`, evaluates that expression and yields a representation of its value as a value of type `value`. From this we can easily define another function `eval` of type `exp -> value` which fixes the top-level value environment to be empty.

First off, let's be more precise about the representation of Mini-ML expressions as values of type `exp`. For example, the Mini-ML expression `2+3` is represented by the value `Plus(Integer 2, Integer 3)` of type `exp`, and the expression `fn x => x` is represented by the value `Fn ("x", Var "x")`. In general, if e is a Mini-ML expression, then $\ulcorner e \urcorner$ ("corners e" or "quote e") is its representation as a value of type `exp`. Thus $\ulcorner e \urcorner =$ `Plus(Integer(2), Integer(3))`, and so on. A formal definition of quotation is given in Figure 1. There is a corresponding representation function for *values*, which we write the same way as $\ulcorner v \urcorner$. Note that we have omitted some cases which are analogous to the given ones.

Now we can state precisely the behavior of `eval`: given a Mini-ML expression e, `eval` $\ulcorner e \urcorner$ evaluates to $\ulcorner v \urcorner$ iff e evaluates to v. For example, if e is the Mini-ML expression `2+3`, which evaluates to `5`, then `eval` $\ulcorner e \urcorner$ evaluates to `Rational(5//1)`, which is $\ulcorner 5/1 \urcorner$ as a value. In other words, given the representation $\ulcorner e \urcorner$ of a Mini-ML expression e, the function `eval` yields the representation $\ulcorner v \urcorner$ of its value v according to the rules of evaluation for Mini-ML.

For the remainder of these notes I ask you to accept the following claim without proof: there is a type `exp` of the abstract syntax of *all* of Standard ML for which we can write a function `eval` of type `exp -> value` such that `eval` $\ulcorner e \urcorner$ evaluates to $\ulcorner v \urcorner$ iff e evaluates to v. The extension to all of ML does not involve many new ideas beyond those appearing in the interpreter for Mini-ML, but, as you can readily imagine, there are a lot more cases to consider.

With this bit of machinery in hand we return to the question of decidability of halting for *programs*, rather than *functions*. We will prove that there is no ML function K of type `exp * exp -> bool` with the following properties:

1. K `(`v`,`w`)` evaluates to either `true` or `false` for any values v and w of type `exp`;

2. if e is an ML expression of type `exp -> exp` and v is a value of type `exp`, then K `(`$\ulcorner e \urcorner$`,`v`)` evaluates to `true` iff $e\,v$ halts.

Before giving the proof, let us note carefully what is being said. First of all, we require that K terminate for all pairs of inputs of type `exp * exp`. Second, we specify that K test halting of *the representation of* an ML function of type `exp->exp` on a given input of type `exp`. That is, if e is an ML function of type `exp->exp` for which we are interested in testing whether or not it halts on a given input v, then we call K with the argument `(`$\ulcorner e \urcorner$`,` v`)` and see whether or not the result is `true`. This is tricky, and you should pause to make sure you understand precisely what is happening here before reading any further.

Suppose that we are given such a function K. Let `diag` be the ML function of type `exp -> exp` defined by

```
fun diag (x) = if K (x,x) then loop () else Integer(0)
```

where `loop` is as defined before and `Integer(0)` is a convenient value of type `exp`.

Consider the evaluation of K `(`⌜`diag`⌝`,`⌜`diag`⌝`)`. By the first assumption on K, this expression evaluates to either `true` or `false`. We consider each case in turn, as before, in order to derive a contradiction:

1. Suppose that K `(`⌜`diag`⌝`,`⌜`diag`⌝`)` evaluates to `true`. Then by the second assumption on K it follows that `diag`⌜`diag`⌝ halts. But by the definition of `diag` we see that `diag`⌜`diag`⌝ halts only if K `(`⌜`diag`⌝`,`⌜`diag`⌝`)` evaluates to `false`, contradicting the assumption.

2. Suppose on the other hand that K `(`⌜`diag`⌝`,`⌜`diag`⌝`)` evaluates to `false`. Then by the second assumption on K it follows that `diag`⌜`diag`⌝ loops forever. But by the definition of `diag` this happens only if K `(`⌜`diag`⌝`,`⌜`diag`⌝`)` evaluates to `true`, contradicting the assumption.

Thus we arrive at a contradiction in either case, and conclude that there is no ML function K satisfying the two conditions stated above. That is, the halting problem is undecidable even if we are given the (representation of the) program, and not just the function as a "black box".

We've just proved that the halting problem for representations of functions of type `exp -> exp` as values of type `exp` and arguments of type `exp` is undecidable: there is no ML function K taking a representation of such a function and a purported argument and yields `true` or `false` according to whether or not that function halts on the given input. The restriction to functions of type `exp -> exp` may seem rather odd. After all, few programs that we've written this semester (with the exception of `eval`!) have this type. Perhaps halting is decidable for (representations of) functions of type, say, `int -> int` and purported arguments of type `int`?

The answer is "no" because we can reduce the halting problem to this problem. That is, we can encode instances of the halting problem as instances of the latter problem, demonstrating that no such decision procedure exists. The reduction is achieved by a technique called *Gödelization*, named after the great Austrian mathematician Kurt Gödel who invented it. The technique is based on a fundamental lemma: *there is a one-to-one and onto correspondence between values of type `exp` and values of type `int`.* That is, we can encode programs as integers and, conversely, decode integers into programs, without any loss of information.[2]

How is this done? A simple approach is to think of the ASCII representation of characters. Each character is assigned a code number between 0 and 127. We may think of each character as a "digit" in base 128. Then a string is nothing more than a base-128 number — in other words, a very large integer. Now this representation is somewhat awkward to work with, especially if we want to take apart programs and put them back together. But we can, if we'd like, write a parser and un-parser to translate to and from abstract syntax (the type `exp`!), where we can do the real work. The important point is that the parsing and un-parsing can be designed to be mutual inverses: if you parse a string, then un-parse it, you get back the same string, and vice-versa. In this way we can reduce the halting problem for functions of type `exp -> exp` to the halting problem for functions of type `int -> int`, demonstrating that the latter is also undecidable.

Exercise 2 *Give an informal argument that halting is undecidable for representations of functions of type `int list -> int list`.*

4 Church's Thesis

To complete our discussion of computability, we consider the generality of our results. So far we've demonstrated that several problems about ML functions and about representations of ML functions as ML data structures are all undecidable. The possibility remains, however, that all of this is an artifact of ML. 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 ML interpreter for C code and a C interpreter for ML 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 ML by asking about the behavior of the ML interpreter written in C.[3] 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 for 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

[2]For this to work with programs of any size, we must assume that our integers are of unbounded size. Mere 32-bit machine words will not suffice. But, as I said in the introduction, we are abstracting away from the physical capacity of a given machine.

[3]I'm glossing over a few details here, but this is the general idea.

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 computabiity 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.

5 Conclusion

Thus 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.