Decidability

1 Decidable properties and computable functions

Recall the definition we saw earlier. A decision problem is said to be *decidable* when there is some program that, when given an argument, says whether the answer is Yes or No. But what is a "program"? A program in what language?

Let's say we have a decision problem concerning words in our language Σ^*. This can be expressed in two ways:

- As a function Σ^* to $\{\mathrm{Yes}, \mathrm{No}\}$, sending good words to Yes, and bad words to No.
- As a subset of Σ^*, viz. the set of good words.

Let's first think about solving our problem on a Turing machine. The tape alphabet is $\Sigma \cup \{\textvisiblespace\}$ and the set of return values is $\{\mathrm{Yes}, \mathrm{No}\}$. We start execution with a word w on an otherwise blank tape; the head is on the cell to the left of the word. At the end of execution, the head is where it began, the tape is blank, and the machine returns Yes if w is a good word and No if it's a bad word.

Is there such a Turing machine? If there is, we say that the problem is *decidable*. But surely Turing machine gives a rather restrictive notion of an algorithm? Maybe we should be more liberal. Maybe we should allow two-tape Turing machines, or two-dimensional Turing machines. Maybe we should allow programs written in a language with infinitely many integer variables and string variables, and `for` loops and `while` loops and recursive procedure calls and pointers and storable labels and ...

Many different notions of "algorithm" have been proposed that, at first sight, appear to be more expressive than a Turing machine. But every time, it has turned out that—as far as *decision problems on words* are concerned—the fancy notion is no more expressive than a Turing machine. If, in my fancy new programming language, I can solve such a decision problem, I could already solve it on a Turing machine.

This is remarkable. It tells us that the notion of decidability is *robust*. All these very different definitions turn out to be equivalent.

Perhaps somebody in the future will invent a new notion of "algorithm" that clearly counts as algorithmic and yet can solve decision problems on words that a Turing machine cannot? *Church's thesis* (named after Alonzo Church) says that this will not happen.

Church's thesis states: "any decision problem on words that can be solved by an algorithm can be solved by a Turing machine".

Another version is for functions $f\colon \Sigma^* \to \Sigma^*$. In this case the Turing machine must start with a word w on an otherwise blank; the head is to the left of the word. At the end of execution, the machine must stop with the word $f(w)$ on an otherwise blank; the head is to the left of the word. When there is such a machine, we say that f is *computable*. (Older literature uses the word

"recursive" instead.) Church's thesis for functions states: "any functions on words that can be computed by an algorithm can be computed by a Turing machine".

Let me again emphasize that natural numbers, integers, lists of words etc. can all be encoded as words, just as they can all be encoded as natural numbers. Also, we may talk about undecidable *problems*, undecidable *properties*, undecidable *subsets* and undecidable *languages*. But the idea is the same in all cases, despite the varying terminology.

2 Primitive and Basic Java

Here is a different viewpoint. Let's give the name *Primitive Java* to a Java-like language with only the type `nat`, for (unlimited) natural numbers. It has the the following facilities.

- Create a variable `nat i = 0`.
- Increment a variable `i++`.
- Decrement a variable `i--`. This does nothing if `i==0`.
- Conditionals `if i == 0` $\{M\}$ `else` $\{N\}$.
- Repetition a fixed number of times `repeat i times` $\{M\}$.

Just to get started, here are some useful encodings. We can encode `nat j = i` as follows:

```
nat j = 0;
repeat i times {j++;}
```

We can encode `j = 0` as follows:

```
repeat j times {j--;}
```

We can encode `j = i` as follows:

```
j = 0;
repeat i times {j++;}
```

We can encode `if i <= j` $\{M\}$ `else` $\{N\}$ as follows:

```
nat k = i;
repeat j times {k--;}
if k == 0 {M} else {N}
```

We can encode `if i < j` $\{M\}$ `else` $\{N\}$ as `if j <= i` $\{N\}$ `else` $\{M\}$. We can encode `if i == j` $\{M\}$ `else` $\{N\}$ as follows:

```
if i < j {N} else {if j < i {N} else {M}}
```

A program has several (immutable) parameters `input0`, `input1`, `input2`, etc., and a variable `output` that's initialized to 0. For example, here is a program that computes the sum of `input0` and `input1`.

```
nat output = 0;
repeat input0 times {
   output++;
}
repeat input1 times {
   output++;
}
```

If `input0` $== 3$ and `input1` $== 5$, then execution terminates with `output` $== 8$.

Note what's lacking: while-loops and recursion. Let's give the name *Basic Java* to Primitive Java extended with while-loops: `while i > 0` $\{M\}$. For example, we can encode `hang` as follows:

```
nat i = 0;
i++;
while i > 0 { }
```

To encode `while` c $\{M\}$ for a condition c we write

```
nat k = 0;
if c { k++; } else { }   // Sets k to be 1 if c is true, else 0.
while k > 0 {
   M
   k = 0;
   if c { k++; } else {}
}
```

where `k` is a *fresh* variable, i.e. one that doesn't appear in M.

3 Computing functions

Consider a program that is k-*ary*, i.e. has k inputs. If the program is in Primitive Java, every list of inputs $\mathbf{x} \in \mathbb{N}^k$, it will terminate with an output. Thus the program computes a (total) function from $\mathbb{N}^k$ to $\mathbb{N}$. If the program uses while-loops, there's a possibility of nontermination. So the program computes a *partial* function. For example, the following unary program

```
nat output=0;
nat j=0;
j++;
j++;
while (j > 0) {
  if (input0 <= j){
    j--;
    output++;
  }
}
```

will return an output of 2 if the input is 0 or 1, but otherwise will hang. So it computes the partial function that sends 0 and 1 to 2 and is undefined on all numbers $>= 2$.

We can translate between Basic Java and Turing machines, in such a way that the translation preserves the meaning: i.e., the translated program computes the same partial function as the original one. Thus the partial functions that are expressible in Java are precisely the computable ones.

Any function that can be computed by a Primitive Java program is said to be *primitive recursive*. The important case is where $k = 1$, because a k-tuple of natural numbers can be encoded as a single one. Instead of (or as well as) natural numbers, we could use integers or strings or lists of natural numbers, provided we modify Primitive Java to include suitable operations.

The primitive recursive functions include all the familiar functions such as comparison, multiplication, exponentiation, factorial and so on. They also include all functions that can be computed in a Turing machine in polynomial time, exponential time and much more. People used to think that they were the only functions on $\mathbb{N}$ that could be computed algorithmically.

However in 1928, Wilhelm Ackermann made a shock discovery: a function that can be computed algorithmically, but is not primitive recursive. The Ackermann function (actually a simplified version due to Rózsa Péter) takes two arguments, and is given as follows:

$$\begin{aligned} A(0,n) &= n+1 \\ A(m+1,0) &= A(m,1) \\ A(m+1,n+1) &= A(m,A(m+1,n)) \end{aligned}$$

This is a program using recursion. Here is a proof that it terminates.

For $m, n \in \mathbb{N}$, let $Q(m,n)$ be the statement that the evaluation of $A(m,n)$ terminates. We prove $\forall m \in \mathbb{N}.\, \forall n \in \mathbb{N}.\, Q(m,n)$ by induction.

- The base case says $\forall n \in \mathbb{N}.\, Q(0,n)$. It holds since $A(0,n)$ returns $n+1$.
- For the inductive step, we suppose $\forall n \in \mathbb{N}.\, Q(m,n)$ and want to prove $\forall n \in \mathbb{N}.\, Q(m+1,n)$. We do this by induction.
  - The base case says $Q(m+1,0)$. This holds because $A(m,1)$ returns a value p, by the outer inductive hypothesis *i.e.* $Q(m,n)$, so $A(m+1,0)$ also returns p.
  - For the inductive step, we suppose $Q(m+1,n)$ and we want to prove $Q(m+1,n+1)$. Then $A(m+1,n)$ returns an answer p, by the inner inductive hypothesis *i.e.* $Q(m+1,n)$, and $A(m,p)$ returns a value q, by the outer inductive hypothesis *i.e.* $Q(m,n)$, and so $A(m+1,n+1)$ also returns q.

The Ackermann function cannot be written in Primitive Java (though I won't prove this). However, it *can* be written in Basic Java: the rough idea is to represent the stack of recursive calls as an extra parameter. This shows the importance of while-loops even for the purpose of computing total functions on natural numbers.

Church's thesis says that we cannot extend Basic Java to a language that can express *more* partial functions from $\mathbb{N}^k$ to $\mathbb{N}$ and yet is still a *programming language* i.e. the programs written in it can be computed algorithmically.

A language like Basic Java that can express all computable total functions is said to be *Turing-complete*. We have seen that Primitive Java is not Turing-complete, as it cannot express the Ackermann function. On the other hand it has the advantage of being a *total* language, i.e. all programs terminate. Would you rather use a language that is Turing-complete, or one that is total? Unfortunately, it turns out that no language can be both! Most languages used in practice (if they allow unlimited natural numbers) are Turing-complete, like Basic Java. But a few, such as Agda, are total and Turing-incomplete, like Primitive Java.

4 Examples of decidable and undecidable properties

Here are some examples of problems and their decidability status.

- The problem of saying whether two regular expressions are equivalent. This is decidable, as we learnt earlier in the term.
- The problem of saying whether a given context-free grammar accepts a given word. This is decidable.
- The problem of saying whether a given context-free grammar accepts any word at all. This is decidable.
- The problem of saying whether a given context-free grammar accepts every word (for the alphabet Σ). This is undecidable.
- The problem of saying whether a given context-free grammar is ambiguous. This is undecidable.
- The problem of saying whether a given integer-coefficient polynomial in one variable has a rational solution.[1] This is decidable.
- The problem of saying whether a given integer-coefficient polynomial in several variables has a rational solution. It is currently unknown whether this is decidable or not.
- The problem of saying whether a given integer-coefficient polynomial in several variables has an integer solution. This is undecidable, as proved in 1970 by Matiyasevic, Robinson, Davis and Putnam. The question was originally asked by Hilbert in a famous lecture in 1900, although nobody at that time had a precise definition of "decidable".
- The problem of saying whether a given propositional formula is true for all interpretations of the propostional atoms. (Such a formula is called a *tautology.*) This is decidable—just evaluate the formula for each assignment.
- The problem of saying whether a given first-order formula is true for all interpretations of the predicate symbols. This is undecidable.

There are many other examples of decidable and undecidable problems. We'll see some of them in the coming weeks.

[1] A *solution* of a polynomial is an assignment of values to the variables that makes the polynomial equal to zero.

5 Semidecidable properties

A property P of natural numbers is said to be *semidecidable* when there's a program M that, when executed on a number n, returns True if n satisfies P, and runs forever otherwise. (For decidability, the program would have to return False otherwise.)

Here's an example: the problem of saying whether a given integer-coefficient polynomial p in several variables has an integer solution. Let's see that this is semidecidable.

First of all, recall that we can treat lists of integers as natural numbers. So consider the following program. Given polynomial p in k variables, apply p to the first list of k integers, then to the second list of k integers, then to the third list, and so on. If any of these calculations returns 0, stop and return True. Since every list of k integers appears somewhere in this enumeration, the program will stop and return True if p has an integer solution . But if p has no integer solution, then the program will run forever.

What is the relationship between semidecidability and decidability?

- Any property P that is decidable must be semidecidable. For if M is a program the decides it, then here is a program that semidecides it. For any natural number n, run M on it, and if this returns True, then return True, but if it returns False, then run forever.

- If both P and the negation of P are semidecidable, then P is decidable. For if M is a program that semidecides P, and N is a program that semidecides the negation of P, then here is a program that decides P. For any natural number n, run M for one step, then N for one step, then M for another step, then N for another step, and so forth. If one of the steps of M returns True, then return True, but if one of the steps of N returns True, then return False.

We conclude that a property P is decidable iff both P and the negation of P are semidecidable. (To see the forwards direction, note that if P is decidable, then the negation of P must also be.)