Computation Theory

Supervision 2 – Solutions

5. The Halting Problem and undecidability

1. Show that decidable sets are closed under union, intersection, and complementation. Do all of these closure properties hold for undecidable languages?

> Let S, T be decidable sets. Intuitively, this means that we can ask the question $n \in^? S$ and get an answer. Closure under union, intersection and complementation simply amounts to asking the appropriate questions and combining the answers: $n \in^? S \cup T$ if $n \in^? S$ or $n \in^? T$, $n \in^? S \cap T$ if $n \in^? S$ and $n \in^? T$, $n \in^? S^\complement$ if not $n \in^? S$. The logical operations are of course computable: for instance, for deciding $S \cup T$, we run the machine deciding S, halt if the result register is 1, run the machine deciding T otherwise and return its result.
>
> These constructions rely on the fact that the membership test returns an answer, so the reasoning can't be adapted for undecidable languages. The only somewhat obvious result is that the complement of an undecidable language is also undecidable: if it wasn't, we could just negate the answer to decide the original set. However, undecidable sets are *not* closed under union and intersection (and some other set operations): there are undecidable sets which combine to become decidable. There's nothing magical about this: if S is undecidable, then so is $S^\complement$, but $S \cup S^\complement = \mathbb{N}$ and $S \cap S^\complement = \emptyset$, which are of course decidable.

2. Suppose S_1 and S_2 are subsets of $\mathbb{N}$. Suppose $r \in \mathbb{N} \to \mathbb{N}$ is a register machine computable function satisfying: for all n in $\mathbb{N}$, n is an element of S_1 if and only if $r(n)$ is an element of S_2. Show that S_1 is register machine decidable if S_2 is. Is the converse, inverse, or contrapositive of this statement true?

> Such a mapping $r : \mathbb{N} \to \mathbb{N}$ between subsets is called a *reduction* $r : S_1 \leq S_2$: decidability of S_1 can be *reduced* to decidability of S_2. Intuitively, if S_2 is decidable, then we can turn the problem of decidability of S_1 into the decidability of S_2 via the reduction, and due to the assumption on r, the answer of $r(x) \in^? S_2$ gives us the answer for $x \in^? S_1$.
>
> We can express this formally by considering the characteristic functions of the subsets. The reduction property of r, $\forall n \in \mathbb{N}.\ n \in S_1 \iff r(n) \in S_2$, can be expressed as
>
> $$\forall n \in \mathbb{N}.\ \chi_{S_1}(n) \iff \chi_{S_2}(r(n))$$
>
> That is, the functions χ_{S_1} and $\chi_{S_2} \circ r$ are equal. If S_2 is decidable, χ_{S_2} is a computable function; composing it with another computable function r implies that χ_{S_1} is also computable, i.e. S_1 is a decidable subset of $\mathbb{N}$.
>
> This theorem gives us a useful proof technique: to show that a set S is decidable, we need to find a computable reduction to another set T which we know to be decidable. Such proofs

> will be common in Complexity Theory next term. In this course, we are more interested in the *contrapositive* of the statement: if S_1 is undecidable and there is a reduction r from S_1 to S_2, then S_2 is also undecidable. Reductions "propagate" undecidability: this is because, as the proof above shows, decidability of S_2 would imply decidability of S_1. Of course, the "de facto" undecidable problem is the Halting Problem; expressed as a set, it's defined as
>
> $$H \triangleq \{ \langle e, x \rangle \mid \varphi_e(x)\downarrow \}$$
>
> To show that another set S is undecidable, it is enough to construct a reduction $r : H \leq S$, i.e. a function $r : \mathbb{N} \to \mathbb{N}$ which maps the code of a program e and an argument x to an element $s \in \mathbb{N}$ which is furthermore an element of $S \subseteq \mathbb{N}$ if and only if $\varphi_e(x)$ halts.
>
> Note that despite the reduction condition being a bi-implication, the reduction itself is still a one-sided function: in particular, neither the converse (if S_1 is decidable then S_2 is decidable) or inverse (if S_2 is undecidable then S_1 is undecidable) of the above statement hold. The theorem might seem a bit backwards, but as you can see from the proof, that is the only way it could work: the reduction translates the *input* to the problem (i.e. the natural $n \in \mathbb{N}$), not the *answer* (i.e. the Boolean $\chi_S(x) \in \mathbb{B}$).

3. Show that the set E of codes $\langle e, e' \rangle$ of pairs of numbers satisfying $\varphi_e = \varphi_{e'}$ is undecidable.

> This corollary demonstrates that equality of programs is undecidable. There are several ways of establishing this: all we need to show is that some other undecidable set would be decidable if we could decide equality of programs. Consider, for instance, the set $S_0 = \{ e \mid \varphi_e(0)\downarrow \}$ from Slide 57: the set of program codes that halt with argument 0. To decide whether a function φ_e halts at 0, we can consider the partial function $c_e(x) = \varphi_e(0)$ which is the constant function with value $\varphi_e(0)$ when $\varphi_e(0)$ is defined, and the constant undefined function otherwise. To choose between the two, we can ask whether c_e is equal to the totally undefined function $\bot$ (where $\bot(x)\uparrow$ for all x) using our machine deciding E. If c_e is equal to the totally undefined function, $\varphi_e(0)$ is undefined so it shouldn't be an element of S_0. If E says that the two are not equal, then $\varphi_e(0)$ must be defined, so $e \in S_0$. Thus, if we could decide E, we could decide S_0, but that is a contradiction.
>
> We can present this reasoning more formally as a reduction proof from S_0 to $E^\complement = \{ \langle e, e' \rangle \mid \varphi_e \neq \varphi_{e'} \}$, then using the fact that undecidable languages are closed under complement. The reduction $r : S_0 \leq E$ maps the code e to the code of the pair consisting of the (code of the) constant function $x \mapsto \varphi_e(0)$, and the totally undefined function $\bot$. By the reasoning above, e will be in S_0 if and only if $x \mapsto \varphi_e(0)$ is not equal to the totally undefined function, i.e. both are in $E^\complement$. This implies that $E^\complement$ is undecidable, and so is E.

4. Show that there is a register machine computable partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ such that both sets $\{ n \in \mathbb{N} \mid f(n)\downarrow \}$ and $\{ y \in \mathbb{N} \mid \exists n \in \mathbb{N}.\, f(n) = y \}$ are register machine undecidable.

> We are asked to define a partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ such that the sets $S_1 \triangleq \{ n \in \mathbb{N} \mid f(n)\downarrow \}$ and $S_2 \triangleq \{ y \in \mathbb{N} \mid \exists n \in \mathbb{N}.\, f(n) = y \}$ are undecidable. Undecidability means that we

have two reductions $r_1 : H \leq S_1$ and $r_2 : H \leq S_2$, mapping pairs $\langle e, x \rangle$ to natural numbers such that

$$\varphi_e(x)\downarrow \iff f(r_1(\langle e, x \rangle))\downarrow \ \wedge \ \exists n \in \mathbb{N}.\, f(n) = r_2(\langle e, x \rangle)$$

The first condition suggests that f should preserve halting, while the second one suggests that its return value should have something to do with the pair $\langle e, x \rangle$. Therefore a good first guess is a function whose domain of definition (the set of naturals where it is defined) is the set of pairs $\langle e, x \rangle$ where $\varphi_e(x)\downarrow$, which is precisely H. Given a natural n, f should decode it as a pair $\langle e, x \rangle$, and return a value only if $\varphi_e(x)$ halts. What should be the return value? From the second condition we see that $f(\langle e, x \rangle) = \langle e, x \rangle$, which certainly holds for the identity function. Hence our guess is the "partial identity function"

$$f(n) \triangleq \begin{cases} n & \text{if } n = \langle e, x \rangle \text{ and } \varphi_e(x)\downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

Using the RM components for decoding n as the pair, and the universal RM to run the computation, we can show that f is computable. However, the set S_1 is equal to $\{ n \in \mathbb{N} \mid n = \langle e, x \rangle \ \wedge \ \varphi_e(x)\downarrow \}$, and S_2 is $\{ y \in \mathbb{N} \mid \exists n \in \mathbb{N}.\, n = y \ \wedge \ n = \langle e, x \rangle \ \wedge \ \varphi_e(x) \}$, both of which are precisely the set $\{ \langle e, x \rangle \mid \varphi_e(x)\downarrow \} = H$. The sets S_1 and S_2 are equal to the set associated with the Halting Problem, and therefore are undecidable.

6. Turing machines

1. Compare and contrast register machines with Turing machines: how do they keep track of state, how are programs represented, what form do machine configurations and computations take?

   - A register machine stores data as natural numbers in its registers, while a Turing machine writes symbols on a tape. Since there can only be a finite number of symbols, natural numbers (and other data) have to be encoded explicitly on the tape, using e.g. unary encoding. The state of program execution (program counter) corresponds to the label of the currently executed register machine instruction; in TMs, it is a function of the symbol under the current tape head, and the internal state of the machine.
   - RM programs are a finite list of RM instructions: increment, conditional decrement, and halt. In Turing machines, the program is the transition function $\delta : (Q \times \Sigma) \to (Q \cup \{ \texttt{acc}, \texttt{rej} \}) \times \Sigma \times \{ L, R, S \}$ which assigns a state, symbol, and movement direction to every pairing of the current TM state and symbol under the tape head.
   - The configuration $(\ell, r_0, r_1, \ldots, r_n)$ of a RM is the current instruction label ℓ and the contents of the registers. A computation $c_0, c_1, \ldots$ is a sequence of configurations starting with the initial configuration c_0 (containing the initial register contents and instruction label 0), with each c_{n+1} determined from $c_n = (\ell, r_0, r_1, \ldots, r_n)$ by executing the instruction at label ℓ on the registers $r_0, \ldots, r_n$. The computation halts if the sequence of configurations is finite (and ends at a HALT instruction), and doesn't halt if the sequence is infinite.

> A TM configuration (q, w, u) consists of the current machine state q, string of symbols w up to and including the tape head, and (finite) string of symbols to the right of the tape head. A computation starts from the initial configuration $(s, \triangleright, u)$ and transitions to new configuration based on the transition function δ. The computation halts if the sequence of configurations is finite (and ends in an `acc` or `rej` state), and doesn't halt if the sequence is infinite.

2. Familiarise yourself with the Chomsky hierarchy and explain the connection between regular expressions and Turing machines.

> The Chomsky hierarchy is a classification of formal languages of which regular languages and (semi)decidable languages are the two extremes. Each type of language/grammar in the hierarchy is associated with a machine that can recognise membership for a language: for example, regular languages are accepted by finite state automata (recall the last part of the *Discrete Mathematics* course last year), while semidecidable languages are recognised by Turing machines. The precise distinction between recognising and deciding a language (and semidecidable vs. decidable languages) will be discussed in *Complexity Theory*; for now it's worth noting that decidable languages require a machine computing the (total) characteristic function (i.e. they must reject the string explicitly, if it's not a member of a set), while semidecidable languages only need firm acceptance for elements of the set, but can reject or diverge if the element is not in the set.
>
> The full Chomsky hierarchy for demonstration purposes (more detail in the *Formal Models of Language* course next term):
>
> | Grammar | Languages | Machine |
> |---|---|---|
> | Type-0 | Semidecidable | Turing machine |
> | Type-1 | Context-sensitive | Linear-bounded nondeterministic TM |
> | Type-2 | Context-free | Nondeterministic pushdown automaton |
> | Type-3 | Regular | Finite state automaton |

7. Notions of computability

1. Before the formal development of the field of computation theory, mathematicians often used the term *effectively computable* to describe functions that can – in principle – be computed using mechanical, pen-and-paper methods.

   a) How did Church and Turing propose to formalise the notion of effective computability and generalise it to other models of computation?

> The Church–Turing thesis states that formal models of computation exactly characterise the nature of effective computation: a function on the natural numbers is effectively computable if and only if it is Turing-computable. This is just a hypothesis, and since there is no formal definition of effective computability, it cannot be formally proved; however, the thesis is universally accepted as identifying the classes of form-

> ally and effectively computable functions. In addition, the thesis also states that all (sufficiently strong) models of computation are in fact equivalent: Church and Turing proved this for the independently developed models of Turing machines, partial recursive functions and the λ-calculus, and it has since been reinforced through many new models of computation that are all equivalent to Turing machines. In summary, the Church–Turing thesis gives a formal definition of *computable functions* (functions that are computed by a Turing machine, or any other model of computation), and equates this definition with the informal notion of *effective computability*. A useful consequence of this is that it gives us a shortcut to establishing the computability of functions: instead of giving a full, formal specification of an operation as a Turing/register machine program or lambda-expression, we can write an informal, English description of the algorithm, and as long as there are no dubious steps (such as *if the computation halts, do …, else do …*), we have strong reasons to believe that the description corresponds to a computable function.

b) Suppose we invented a new model of computation. How can we establish that it is as "powerful" as mechanical methods? Make sure to formally explain what "power" means in this case.

> The power of a model of computation simply refers to the class of functions that it can compute: if that class is as big as the class of Turing-computable function, the model of computation is as powerful as a Turing machine. To establish Turing-completeness it is sufficient to encode a Turing machine (or any other Turing-complete model of computation) in the system; to compute a computable function, we can simply simulate the associated Turing machine computation.

c) Can our new model be even more powerful?

> The most likely answer is no: so far, every new model of computation was proved to be computationally equivalent to a Turing machine, meaning that both can simulate each other. The Church–Turing thesis implies that any model of computation that can simulate a Turing machine is computationally equivalent to a Turing machine (as a TM can simulate any other model of computation), and all known models of computation support this. However, we do not have a definitive proof of this (because the Church–Turing thesis cannot be formally proved), so in principle there may be a model of computation that is more powerful than a Turing machine and cannot be simulated by one. This is the realm of *hypercomputation* or super-Turing computation, and while there are some theoretical models of hypercomputation (using random oracles or infinite time), there is little hope in discovering a "practical" model that would invalidate the Church–Turing thesis.

2. Briefly describe of three Turing-complete models of computation not covered in the course.

> There are many examples: abstract rewriting systems, combinatory logic, Kahn process networks, some cellular automata, etc. Sometimes Turing-completeness arises unintentionally in a system not necessarily developed as a model of computation: many games and software have been shown to be Turing-complete by enthusiastic users. Some unsurprising examples are *Minecraft, LittleBigPlanet, Excel, Factorio, Opus Magnum*; some slightly more surprising ones are C++ templates, Java generics, SQL, *Magic: The Gathering*, PowerPoint, and of course the sewage-based 4-bit adder in *Cities: Skylines*. It's a fun internet hole to get lost in.

8. Partial recursive functions

1. Show that the following functions are all primitive recursive. Make sure to give the final form of the function as a composition of primitive functions and projection.

   a) Truncated subtraction function, $minus\colon \mathbb{N}^2 \to \mathbb{N}$, where

   $$minus(x,y) \triangleq \begin{cases} x-y & \text{if } y \leq x \\ 0 & \text{if } y > x \end{cases}$$

   > The recursive definition of the function is as follows:
   >
   > $$\begin{cases} minus(x,0) = x \\ minus(x,y+1) = pred(minus(x,y)) \end{cases}$$
   >
   > We know that $pred$ is primitive recursive, and thus so is $minus$. Explicitly, we have
   >
   > $$minus \triangleq \rho^1\left(\mathtt{proj}_1^1,\ pred \circ \mathtt{proj}_3^3\right) = \rho^1\left(\mathtt{proj}_1^1,\ \rho^0\left(\mathtt{zero}^0,\ \mathtt{proj}_1^2\right) \circ \mathtt{proj}_3^3\right)$$

   b) Exponentiation, $exp\colon \mathbb{N}^2 \to \mathbb{N}$, where $exp(x,y) = x^y$.

   > The recursive definition is:
   >
   > $$\begin{cases} exp(x,0) = 1 \\ exp(x,y+1) = mult(x, exp(x,y)) \end{cases}$$
   >
   > $$\begin{aligned} exp &\triangleq \rho^1\left(\mathtt{succ} \circ \mathtt{zero}^1,\ mult \circ \left[\mathtt{proj}_3^3, \mathtt{proj}_1^3\right]\right) \\ &= \rho^1\left(\mathtt{succ} \circ \mathtt{zero}^1,\ \rho^1\left(\mathtt{zero}^1,\ plus \circ \left[\mathtt{proj}_3^3, \mathtt{proj}_1^3\right]\right) \circ \left[\mathtt{proj}_3^3, \mathtt{proj}_1^3\right]\right) \\ &= \rho^1\left(\mathtt{succ} \circ \mathtt{zero}^1, \rho^1\left(\mathtt{zero}^1,\ \rho^1\left(\mathtt{proj}_1^1,\ \mathtt{succ} \circ \mathtt{proj}_3^3\right) \circ \left[\mathtt{proj}_3^3, \mathtt{proj}_1^3\right]\right)\right. \\ &\qquad \left. \circ \left[\mathtt{proj}_3^3, \mathtt{proj}_1^3\right]\right) \end{aligned}$$

   c) Conditional branch on zero, $ifzero\colon \mathbb{N}^3 \to \mathbb{N}$, where

   $$ifzero(x,y,z) \triangleq \begin{cases} y & \text{if } x = 0 \\ z & \text{if } x > 0 \end{cases}$$

> The definition is simple, but we need to swap the arguments of the function so the Boolean condition is the last argument (since we only pattern-match on the last argument). Define $C : \mathbb{N}^3 \to \mathbb{N}$ as:
>
> $$\begin{cases} C(x_1, x_2, 0) = x_1 \\ C(x_1, x_2, x+1) = x_2 \end{cases}$$
>
> Then $\mathit{ifzero} = \rho^2\big(\mathtt{proj}^2_1,\ \mathtt{proj}^4_2\big) \circ \big[\mathtt{proj}^3_2, \mathtt{proj}^3_3, \mathtt{proj}^3_1\big]$.

d) Bounded summation: if $f : \mathbb{N}^{n+1} \to \mathbb{N}$ is primitive recursive, then so is $g : \mathbb{N}^{n+1} \to \mathbb{N}$ where where

$$g(\vec{x}, x) \triangleq \begin{cases} 0 & \text{if } x = 0 \\ f(\vec{x}, 0) & \text{if } x = 1 \\ f(\vec{x}, 0) + \cdots + f(\vec{x}, x-1) & \text{if } x > 1 \end{cases}$$

> The function g can be defined recursively as:
>
> $$\begin{cases} g(\vec{x}, 0) = 0 \\ g(\vec{x}, x+1) = \mathit{add}(g(\vec{x}, x), f(\vec{x}, x)) \end{cases}$$
>
> The explicit definition depends on the number of arguments, but it's easy to see that the function is primitive recursive because both f and add are.

2. Explain the motivation and intuition behind *minimisation*. How does it extend the set of functions computable using primitive recursion? Give three examples of computable partial functions that are not definable using primitive recursion, justifying your answer in each case.

> Given a partial function $f : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$, the minimisation of f, denoted $\mu^n f(\vec{x})$ is the least argument $x \in \mathbb{N}$ such that $f(\vec{x}, x) = 0$ while $f(\vec{x}, i)$ for all $i = 0, \ldots, x-1$ is defined and strictly greater than 0. This looks like a very specific operation of seemingly limited applicability, however, it is the operator that expands the class of primitive recursive functions to the class of partial recursive, or computable functions. The way to think about minimisation is as *unbounded search*: we're looking for a least value x satisfying some decidable property P. The key is that the functions we try to minimise are often not some traditional, "naturally arising" functions, but custom-made ones defined specifically to encapsulate the property P that we want satisfied after minimisation. Quite often, the function f will simply be a step function that switches from 1 to 0 as soon as the property holds:
>
> $$f(\vec{x}, x) \triangleq \begin{cases} 1 & \text{if } \neg P(\vec{x}, x) \\ 0 & \text{if } P(\vec{x}, x) \end{cases}$$
>
> Minimising such a function will give us the least x that satisfies the required property. As an example, we can compute the integer division function $\mathit{div} : \mathbb{N}^2 \rightharpoonup \mathbb{N}$ using minimisation

by noticing that the output of $div(x_1, x_2)$ has the unique property that it is the least x such that $x_1 < x_2(x+1)$. Extracting this into a helper function:

$$f(x_1, x_2, x) \triangleq \begin{cases} 1 & \text{if } x_1 \geq x_2(x+1) \\ 0 & \text{if } x_1 < x_2(x+1) \end{cases}$$

This function will switch to 0 exactly when $x = \lfloor x_1/x_2 \rfloor$, so $div \triangleq \mu^2 f$.

Still, many of the functions that are expressible as a minimisation problem can be defined using primitive recursion. The only definite non-example is Ackermann's function, which can be proved to grow faster than any definable primitive recursive function, but has a valid definition using minimisation. This does not mean that the only examples of computable, non-primitive-recursive functions are variations on Ackermann, however. Primitive recursive functions are all total by construction, so any *partial* function is also not primitive recursive – not because of some "limitation" in the power of primitive recursion, but because it can never diverge. Minimisation gives us partiality because there may not be a least element satisfying our property; for instance, in the integer division function above, f never becomes 0 if x_2 is 0, so $\mu^2 f(x_1, 0)$ is undefined, as expected. The Ackermann function is still special, however, as it is one of the simplest *total* computable functions that is not primitive recursive.

3. Use minimisation to show that the following functions are partial recursive:

   a) the binary maximum function $max \colon \mathbb{N}^2 \to \mathbb{N}$.

   The maximum of two numbers $x_1, x_2 \in \mathbb{N}$ is the least x greater than or equal to both x_1 and x_2 – clearly this will return one of the two numbers. We can encapsulate this with the helper function:

   $$f(x_1, x_2, x) \triangleq \begin{cases} 1 & \text{if } x < x_1 \text{ or } x < x_2 \\ 0 & \text{if } x \geq x_1 \text{ and } x \geq x_2 \end{cases}$$

   As before, this switches to 0 when we reached the maximum, so $max \triangleq \mu^2 f$.

   b) the integer square root function $sqrt \colon \mathbb{N} \rightharpoonup \mathbb{N}$ which is only defined if its argument is a perfect square.

   Unlike before, now we ask for the function to be undefined whenever its argument is not of the right form. The usual helper step-function still works here, with $f(x, y) = 0$ if $x = y^2$ and 1 otherwise. However, we can often define the helper function in a more natural way. Consider $f(x, y) = |x - y^2|$. This will be zero only when $x = y^2$ and positive otherwise; moreover, it will never reach 0 if x is not a perfect square. This is precisely what we were looking for, so $sqrt \triangleq \mu^1 f$.

Optional exercises

1. Write a Turing machine simulator in a programming language you prefer (a functional language such as ML or Haskell is recommended). Implement the machine described on Slide 64.

2. For the example Turing machine given on Slide 64, give the register machine program implementing $(S, T, D) := \delta(S, T)$, as described on Slide 70.

3. Recall the definition of Ackermann's function ack. Sketch how to build a register machine M that computes $ack(x_1, x_2)$ in R_0 when started with x_1 in R_1 and x_2 in R_2 and all other registers zero. (E9)

   Hint: Call a finite list $L = [(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots]$ of triples of numbers *suitable* if it satisfies

   a) if $(0, y, z) \in L$, then $z = y + 1$

   b) if $(x + 1, 0, z) \in L$, then $(x, 1, z) \in L$

   c) if $(x + 1, y + 1, z) \in L$, then there is some u with $(x + 1, y, u) \in L$ and $(x, u, z) \in L$.

   The idea is that if $(x, y, z) \in L$ and L is suitable then $z = ack(x, y)$ and L contains all the triples $(x', y', ack(x, y'))$ needed to calculate $ack(x, y)$. Show how to code lists of triples of numbers as numbers in such a way that we can (in principle, no need to do it explicitly!) build a register machine that recognises whether or not a number is the code for a suitable list of triples. Show how to use that machine to build a machine computing $ack(x, y)$ by searching for the code of a suitable list containing a triple with x and y in its first two components.