**Where Are We?**

We have an arguably correct notion of computation, based on any of a number of equivalent models: register machines, Turing machines, Herbrand-Gödel equations, $\mu$-recursion, $\lambda$-definability.

We can use computability to explain formally what it means to solve a problem. And we have a few examples of interesting problems that are semidecidable but not decidable (beyond just Halting).

The next step is to take a closer look at basic properties of the clone of computable functions.

---

**Enumeration Theorem**

Here is yet another way of expressing the fact that there are universal machines.

**Theorem**

There exists a partial recursive function $\Phi : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ such that for every partial recursive function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ there exists an index $\hat{f}$ of $f$ such that $f(x) \simeq \Phi(\hat{f}, x)$ for all $x$.

**Proof.**

One needs to construct a universal device in the corresponding model of computation. For example, for Turing machines the construction was carried out in detail by Turing.

---

**Kleene’s Notation**

We write $\{e\}$ for the $e$th function, $e \geq 0$, defined by the enumeration theorem. Here the index $e$ is a sequence number, but it is helpful to think of it as a program (in some suitable language).

Since these functions are partial in general we have to be a bit careful and write $\{e\}(x) \simeq y$ to indicate that $\{e\}$ with input $x$ returns output $y$ after finitely many steps.

---

**More Convergence**

To express convergence we also write

- $\{e\}(x) \downarrow$
- $\{e\}(x) \uparrow$

if $\{e\}$ on input $x$ terminates and produces some output, and when the computation fails to terminate.

For example, Kleene equality $\{e\}(x) \simeq \{e'\}(x)$ should be interpreted as:

- either $\{e\}(x) \downarrow$ and $\{e'\}(x) \downarrow$ and the output is the same; or
- $\{e\}(x) \uparrow$ and $\{e'\}(x) \uparrow$. 
Recall that a model of computation $M$ consists of a space of configurations with a one-step relation, plus input and output maps.

The one-step relation is always primitive recursive and different I/O conventions make no difference with the computable functions defined by the result.

One really should keep track of arities, so one should write something like

$$\{e\}^n(x) \simeq y$$

or even $\{e\}_{\sigma}^n(x) \simeq y$. We won’t bother with this level of detail.

We can even deal with the situation where $U$ is universal in one model of computation and $U'$ is universal in another model. Say, $U$ is a universal Turing machine and $U'$ is a universal system of Herbrand-Gödel equations.

Then there still is a primitive recursive function $f$ such that

$$\{e\}(x) \simeq \{f(e)\}'(x)$$

Exercise

Try to get an idea what $f$ would look like for the translation from Turing to Herbrand-Gödel.

It is not hard to see that there are infinitely many universal Turing machines, though only a handful interesting ones are known. Each one of these machines provides a different enumeration, so if wanted to be absolutely precise we would need to mention the corresponding machine. But there is no need to be quite so careful.

For suppose we have two different universal machines $U$ and $U'$.

**Lemma**

There is a primitive recursive function $f$ such that

$$\{e\}(x) \simeq \{f(e)\}'(x)$$

**Proof.**

$U'$ can simulate $U$ (and vice versa).  

### Cross-Model Simulation

Recall the basic idea behind any model of computation: we have a space $C$ of configurations and a one-step relation $C \overset{\sigma}{\rightarrow} C'$.

But that means that every computation naturally unfolds in stages

$$C_0, C_1, C_2, \ldots, C_n, C_t$$

where $C_0$ is an initial configuration and $C_t$ is a terminal configuration (at least in the finite case).

Abstractly we can write

$$\{e\}_{\sigma}(x) \simeq y$$

to indicate that $\{e\}$ with input $x$ returns output $y$ after at most $\sigma$ steps.

Recall that $\{e\}$ is a primitive recursive function $f$ such that

$$\{e\}(x) \simeq \{f(e)\}'(x)$$

**Proof.**

$U'$ can simulate $U$ (and vice versa).  

### Stages of a Computation

### Dire Warning

But note that the bound $\sigma$ cannot be computed: there is no total recursive function $f$ such that

$$\{e\}(x) \simeq y \iff \exists \sigma < f(e,x) (\{e\}_{\sigma}(x) \simeq y)$$

Otherwise we could solve the Halting Problem.

However, for practical algorithms (as in 15-451) this is not a problem, we can always predict how long a computation will run. In fact, some elementary bound will work.
We may safely assume that \( \{e\}_\sigma(x) \) always returns a value. To express that the computation has not terminated yet we use the same trick as for bounded search in primitive recursive functions and set
\[
\{e\}_\sigma(x) \simeq \sigma
\]
By the same token, if the computation already converges in \( \sigma \) steps, all parameters are less than \( \sigma \):
\[
\{e\}_\sigma(x) \simeq y \text{ implies } e, x, y < \sigma
\]
Note that the function \( \{e\}_\sigma \) is primitive recursive.

Once we have a fixed enumeration, one can compute with indices (think manipulating programs).

Here is a particularly useful, though admittedly quite unspectacular instance of such an index computation: we can fix some of the arguments of a computable function to obtain another computable function.

**Theorem**

*For every \( m, n \geq 1 \) there is a primitive recursive function \( S^m_n \) such that*
\[
\{S^m_n(e,p)\}^*(x) \simeq \{e\}^{m+n}(p, x).
\]

*Proof.*

Klar (pace Landau).

\( \square \)

We claim that there is a primitive recursive function \( f \) such that
\[
\{f(e, e')\} \simeq \{e\} \circ \{e'\}.
\]

Given two devices, we can compute a new device that represents the composition of the given ones.

Or we could do
\[
\{g(e, e')\} \simeq \{e\} + \{e'\}.
\]

And so forth and so on.

We will use the same notation for any computable function \( f \). So
\[
f(x) \simeq \lim_{\sigma \to \infty} f_\sigma(x)
\]
where the limit is taken in the discrete topology, and may well fail to exist.

This also works the other way around: suppose \( f : \mathbb{N}^2 \to \mathbb{N} \) is primitive recursive such that
- \( f(x, \sigma) \leq \sigma \), and
- \( f(x, \sigma) = y < \sigma \) implies \( \forall t \geq \sigma \ f(x, t) = y \)

Then \( f(x) \simeq \lim f(x, \sigma) \) is computable (partial recursive).

Note that we can also obtain functions like
\[
f = \lambda x, y. F(5, x, 42, y)
\]
since we are always working in a clone.

This may seem a bit nit-picky, but in other frameworks we may have to insist on the existence of transposition operations \( T_{i,j} \) that swap \( x_i \) and \( x_j \) (and thus provide arbitrary permutations of arguments).

The next result is utterly amazing: it shows that we solve functional equations of computable functions in mind-numbing generality.

**Theorem (Kleene, Second Recursion Theorem, 1938)**

Let \( F : \mathbb{N}^{n+1} \to \mathbb{N} \) be a partial recursive function. Then there exists an index \( e^* \) such that for all \( x \in \mathbb{N} :\)
\[
\{e^*\}(x) \simeq F(e^*, x).
\]

Moreover, \( e^* \) can be computed effectively from an index for \( F \).

Think of \( e^* \) as some program, \( x \) as input and \( F \) as an interpreter running \( e^* \) on \( x \); then the claim is obvious. However, the theorem holds for arbitrary computable \( F \).
At first glance, this sounds utterly wrong.
What if $F(u,x) = u + 1$?
Then $\{e^*\}(x) \simeq \{e^*\}(x) + 1$.

Solution: any totally undefined function $\{e^*\}$.

In general it requires extra effort to show that a function obtained from
the theorem is, say, total. A priori, all we get is a computable function, no
more. And, this computable function may be undefined in many places.

The first result, $e^* \simeq \{e^*\}(x)$, is the theoretical foundation for quines,
programs that print themselves.

The real challenge here is that one needs to deal with the idiosyncrasies
(more often: idiocies) of a particular programming language.

Exercise
Write a quine in your favorite programming language.

This is one of the most infuriating proofs known to mankind. Every step
is trivial equational reasoning, but the whole argument makes no sense.

J. C. Owings called it “barbarically short” and “nearly incapable of
rational analysis.”

Owings also suggested a way to make sense out of it: think of it as a
diagonal argument that fails.

In a diagonal argument we have an infinite matrix $S$ over some set $A$:

$$S : \mathbb{N} \times \mathbb{N} \rightarrow A$$

We can think of the rows as infinite sequences over $A$, so $S$ is a table of
infinite sequences.

We have some operation $\alpha$ on $A$ such that the sequence obtained by
applying $\alpha$ to the diagonal $(\alpha(S(i,i)))_{i \geq 0}$ is not in $S$. 
Informal RT

Here is a good intuitive way to think about RT. The typical definition of a computable function $Q$ looks like so:

$Q:\begin{array}{l}
\text{input } x \\
\text{some computation} \\
\text{return } \ldots
\end{array}$

But with the RT we can use an index $q$ for $Q$ inside the definition:

$Q:\begin{array}{l}
\text{input } x \\
\text{some computation using } q \\
\text{return } \ldots
\end{array}$

Application: Halting

Assume for the sake of a contradiction that the halting set $K = \{ e \mid \{ e \}(e) \downarrow \}$ is decidable.

Define $Q$ by

$\begin{array}{l}
\text{input } x \\
\text{if } \text{char}_K(q) \simeq 1 \quad \text{// check } q \in K \\
\quad \text{then } \uparrow \\
\text{else return } 0
\end{array}$

Then $Q(q) \downarrow$ implies $Q(q) \uparrow$, and $Q(q) \uparrow$ implies $Q(q) \downarrow$.

\begin{align*}
\uparrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow
\end{align*}

Application: Ackermann

Define $Q$ by

$\begin{array}{l}
\text{input } x, y \\
\quad \text{if } x = 0 \\
\quad \quad \quad \text{then return } y + 1 \\
\quad \text{else if } y = 0 \\
\quad \quad \quad \text{then return } \{ q \}(x - 1, 1) \\
\quad \text{else return } \{ q \}(x - 1, \{ q \}(x, y - 1))
\end{array}$

Then $Q = \{ q \}$ is none other than the Ackermann function and we have another proof of its computability. Of course, one still has to work to demonstrate totality.

Application: Rice’s Theorem

Let’s say that $P \subseteq \mathbb{N}$ is a non-trivial property of semidecidable sets if

- $W_e = W_{e'}$ implies $e \in P \iff e' \in P$.
- $e_0 \in P$ and $e_1 \notin P$ for some $e_0$ and $e_1$.

Examples are “$W_e$ is empty,” “$W_e$ is finite,” or “$W_e$ is decidable.”

Theorem (Rice 1953)

Every non-trivial property of semidecidable sets is undecidable.
Proof

For the sake of a contradiction assume \( P \) is decidable.

Define \( Q \) by

\[
\text{input } x
\begin{align*}
\text{if } \text{char}_P(q) \equiv 1 & \quad \text{// check } q \in P \\
\text{then return } \{e_1\}(x) \\
\text{else return } \{e_0\}(x)
\end{align*}
\]

But then \( q \in P \) implies \( Q \simeq \{e_1\} \) and thus \( q \notin P \).

On the other hand, \( q \notin P \) implies \( Q \simeq \{e_0\} \) and thus \( q \in P \).

Application: Minimal Machines

All models of computation can be associated with a natural size function. For example, we could define the size of a Turing machine \( M \) to be its index \( \bar{M} \) or the number of bits needed to specify its transition function.

Call \( M \) minimal if no smaller machine is equivalent to \( M \). Here equivalent means that \( \forall z \ (M(z) \simeq M'(z)) \).

Claim

The set of minimal Turing machines is not semidecidable.

So this is a little stronger than just saying minimality is undecidable.

Proof

For the sake of a contradiction assume minimal machines are semidecidable.

Define \( Q \) by

\[
\text{input } x
\begin{align*}
\text{enumerate minimal machines } M_e \text{ until } e > q \\
\text{return } M_e(x)
\end{align*}
\]

But then \( Q = \{q\} \) and \( M_e \) are equivalent, yet \( q < e \), contradicting minimality.

And the First?

Where there is a second, there must be a first. Alas, the First Recursion Theorem is a bit harder to explain, since it uses higher order functionals rather than just functions.

Let us write \( \mathcal{P} \) for all partial arithmetic functions (of fixed arity, we will fudge a bit).

We can define a partial order \( \subseteq \) on \( \mathcal{P} \) by setting \( f \subseteq g \) if

\[\forall x \ (f(x) \downarrow \implies f(x) \simeq g(x))\]

Thus \( f \) and \( g \) agree on the domain of definition of \( f \) (\( g \) extends \( f \)).

Note that this partial order is complete: given an ascending chain \( (f_i) \) we can form \( \bigsqcup f_i \).

Example

Let \( \bot = \emptyset \) be the totally undefined function.

Given any partial function \( f \) define the functional \( F \) by

\[F(f) = f \cup \{(0,1)\} \cup \{(x,x \cdot y) \mid (x-1,y) \in f\}\]

Then \( f = \bigsqcup F^i(\bot) \) is the factorial function.

As it turns out, all computable functions can be constructed in this manner: by a (easily computable) chain of finite approximations that get closer and closer to the target.

Example

We can also model primitive recursion in this manner. Suppose \( f = \text{Prec}[h,g] \). Define the functional \( F \) by

\[F(f) = f \cup \{(0, y, g(y)) \mid y \} \cup \{(x,y, h(x-1, z,y)) \mid (x-1,y,z) \in f\}\]

Then \( f = \bigsqcup F^i(\bot) \).

Note that, as written, these approximations are not finite, \( F(\bot) \) already contains all of \( g \).

Make sure you understand how \( F \) could be adjusted so that all \( F^i(\bot) \) are finite. Try addition first.
Good Functionals

Definition
A functional \( F : \mathcal{P} \to \mathcal{P} \) is effectively continuous if

- monotonicity: \( f \sqsubseteq g \) implies \( F(f) \sqsubseteq F(g) \)
- continuity: if \( (f_i) \) is an ascending sequence in \( \mathcal{P} \), then \( F(\bigsqcup f_i) = \bigsqcup F(f_i) \)
- finite approximations: for some \( F_{\text{fin}} \) partial recursive
  \[ F(\Theta)(x) = F_{\text{fin}}(\hat{\Theta}, x) \]

where \( \Theta \) is a finite function with index \( \hat{\Theta} \).

Compactness

Lemma
Functional \( F \) is effectively continuous iff

\[ F(f)(x) \simeq y \iff \exists \Theta \sqsubseteq f \text{ finite} \quad (F(\Theta)(x) \simeq y) \]

In other words, we can already determine the value of \( F(f)(x) \) by using a suitable finite approximation \( \Theta \sqsubseteq f \): no infinite amount of information is needed (say, the values of \( f \) on all even numbers).

This comports nicely with any intuitive notion of what it means to effectively compute the functional \( F \).

First Recursion Theorem

Theorem (Kleene, First Recursion Theorem, 1938)
Every effectively continuous functional \( F \) has a least fixed point \( f \), a partial recursive function. Moreover, an index for \( f \) can be computed effectively from an index for \( F \).

The construction of the “solution” \( f \) uses an increasing chain of approximation: Let \( f_0 = \bot \) and \( f_{i+1} = F(f_i) \). Then

\[ f = \bigsqcup f_i = \bigsqcup F(\bot) \]

Note that this is essentially call-by-value rather than call-by-name as in the FRT.

Semidecidable Sets and Domains

The definition of a semidecidable set is based on a “semi-algorithm”. Alternatively we can use the semi-characteristic function, but note that any other computable function will do as well.

Proposition
A set is semidecidable if, and only if, it is the domain of a partial computable function.

By domain we mean domain of convergence, aka support.

Only convergence matters, the output is irrelevant (unlike with decision algorithms).

Recursively Enumerable Sets

Recursively Enumerable Sets

There is another way to look at semidecidable sets: one can generate them in a computable manner.

Definition
\( A \subseteq \mathbb{N} \) recursively enumerable (r.e.) if there is a computable function \( f : \mathbb{N} \to \mathbb{N} \) such that \( A \) is the range of \( f \).

Except for \( A = \emptyset \) we can choose \( f \) to be total. More useful is the following:

Lemma
We may assume without loss of generality that the domain of \( f \) is \( \mathbb{N} \) or \( \{0, 1, \ldots, n - 1\} \) for some \( n \), and that \( f \) is injective.
Given $f$, we construct a new function $g$ with the right properties.

We proceed in stages $\sigma \geq 0$. Set $z = 0$.

**Stage $\sigma$:**
Compute $f_0(0), \ldots, f_\sigma(\sigma - 1)$.
If a new value $y$ appears, set $g(z) \simeq y$ and let $z = z + 1$.

Then $g$ is computable, injective, has the same range as $f$ and its support is an initial segment of $\mathbb{N}$, as required.

---

**Approximating Semidecidable Sets**

We can approximate semidecidable sets much the way we can approximate computable functions (in fact, it’s a bit easier).

Let $f$ be the semi-characteristic function for some semidecidable set $A$. Define

$$W_\sigma = \{ x \mid f_\sigma(x) < \sigma \}$$

Note that $W_\sigma \subseteq W$ is a finite set (in fact, it has cardinality at most $\sigma$). Also, $W_\sigma$ is primitive recursive uniformly in $\sigma$.

Lastly, $W = \bigcup W_\sigma$.

---

**Closure**

**Lemma**

The collection of semidecidable sets is closed under union and intersection.

**Proof.**

For union let $A = W_e$ and $B = W_{e'}$. Define

$$f_\sigma(x) \simeq \begin{cases} 0 & \text{if } x \in W_{e,\sigma} \cup W_{e',\sigma}, \\ 1 & \text{otherwise.} \end{cases}$$

Then $f = \lim f_\sigma$ is computable and is none other than the semi-characteristic function of $A \cup B$. The argument of intersection is similar. $\square$
And again: this justifies definitions of the form

\[ f(x) \simeq \begin{cases} g(x) & \text{if } x \in A, \\ \uparrow & \text{otherwise}. \end{cases} \]

If \( g \) is computable and \( A \) is semidecidable, then \( f \) is also computable.

And again again: we cannot replace the \( \uparrow \) in the second case by \( h(x) \) unless \( A \) is decidable.

It follows immediately that the complement \( \overline{K} \) of the Halting set is not semidecidable. Let’s call such sets co-semidecidable.

So we have to cope with at least three types of problems:

- decidable
- semidecidable
- co-semidecidable

Lemma

A partial function \( f \) is computable if, and only if, its graph is recursively enumerable. For total \( f \) the graph is decidable.

Proof. Write \( F \subseteq \mathbb{N}^2 \) for the graph of \( f \).

Suppose \( f \) is computable. To semidecide \((x, y) \in F\), we try to compute \( f(x) \). If the computation converges, we check that the output is \( y \).

For the opposite direction, given \( x \), start enumerating \( F \). If a pair \((x, y)\) appears, output \( y \).

Definition

Suppose \( R \subseteq \mathbb{N}^{n+1} \). The projection \( S \subseteq \mathbb{N}^n \) of \( R \) is defined by

\[ S(x) :\iff \exists z R(z, x) \]

Note that \( S \) is semidecidable whenever \( R \) is decidable: we can search for a witness \( z \).

Lemma

Every semidecidable set is a projection of a decidable set.

This follows immediately from Kleene normal form.