Recursion Theorem

Ziv Scully

18.504

P-p-p-plot twist!

$$\lambda$$

The master plan

1. A very λ-calculus appetizer
2. Main theorem
3. Applications
4. Fixed points and diagonalization

The story so far

What is computation?

Depending on who you ask, computation is...

- ... following clear procedures step by step (Turing machines).

What is computation?

Depending on who you ask, computation is...

- ... following clear procedures step by step (Turing machines).
- ... a composition of primitive functions on $\mathbb{N}$ (μ-recursive functions).

What is computation?

Depending on who you ask, computation is...

- ... following clear procedures step by step (Turing machines).
- ... a composition of primitive functions on $\mathbb{N}$ (μ-recursive functions).
- ... complexity that emerges from simple rules (cellular automata).

What is computation?

Depending on who you ask, computation is...

- ... following clear procedures step by step (Turing machines).
- ... a composition of primitive functions on $\mathbb{N}$ (μ-recursive functions).
- ... complexity that emerges from simple rules (cellular automata).
- ... applying functions to functions to get more functions (λ-calculus).

Definition of λ-calculus

A λ-calculus term is one of three things.

- A variable, such as x, y, or z.

Definition of λ-calculus

A λ-calculus term is one of three things.

- A variable, such as x, y, or z.
- A function application, one term applied to another, such as

$$f\,x, \qquad f\,(g\,x), \qquad \text{or } (f\,x)y = f\,x\,y.$$

Definition of λ-calculus

A λ-calculus term is one of three things.

- A variable, such as x, y, or z.
- A function application, one term applied to another, such as

$$f\,x, \qquad f\,(g\,x), \qquad \text{or } (f\,x)y = f\,x\,y.$$

- A function abstraction, making a term a one-argument function, such as

$$\lambda x.x, \qquad \lambda x.\,(\lambda y.x) = \lambda x.\,\lambda y.x, \qquad \text{or } \lambda f.\,\lambda g.\,\lambda x.f\,(g\,x).$$

Definition of λ-calculus

A λ-calculus term is one of three things.

- A variable, such as x, y, or z.
- A function application, one term applied to another, such as

$$f\,x, \qquad f\,(g\,x), \qquad \text{or } (f\,x)y = f\,x\,y.$$

- A function abstraction, making a term a one-argument function, such as

$$\lambda x.x, \qquad \lambda x.\,(\lambda y.x) = \lambda x.\,\lambda y.x, \qquad \text{or } \lambda f.\,\lambda g.\,\lambda x.f\,(g\,x).$$

Additionally, we require that, at the top level, no variable appear outside the scope of a function abstraction that "declares" it.

Evaluating λ-calculus terms

There are two rules for evaluating terms.

- α-renaming: change variable names, such as

$$\lambda x.\,\lambda y.\,x \rightsquigarrow \lambda f.\,\lambda g.\,f.$$

Evaluating λ-calculus terms

There are two rules for evaluating terms.

- α-renaming: change variable names, such as

$$\lambda x.\,\lambda y.\,x \rightsquigarrow \lambda f.\,\lambda g.\,f.$$

- β-reduction: apply a function to an argument by substitution, such as

$$(\lambda x.\,\lambda y.\,x)(\lambda z.\,z) \rightsquigarrow \lambda y.\,\lambda z.\,z.$$

Evaluating λ-calculus terms

There are two rules for evaluating terms.

- α-renaming: change variable names, such as

$$\lambda x.\,\lambda y.\,x \rightsquigarrow \lambda f.\,\lambda g.\,f.$$

- β-reduction: apply a function to an argument by substitution, such as

$$(\lambda x.\,\lambda y.\,x)(\lambda z.\,z) \rightsquigarrow \lambda y.\,\lambda z.\,z.$$

These rules are powerful enough to simulate a Turing machine.

Hello, factorial!

Let's define the factorial function.

$$F = \lambda x. \begin{cases} 1 & x = 0 \\ x \times (F(x-1)) & \text{otherwise.} \end{cases}$$

Hello, factorial!

Let's define the factorial function.

$$F = \lambda x. \begin{cases} 1 & x = 0 \\ x \times (F(x-1)) & \text{otherwise.} \end{cases}$$

Problem: we can't have F refer to itself in λ-calculus.

Sneakier self-reference

What if we can't use self-reference?

$$F = G\,G, \text{ where}$$
$$G = \lambda g.\, \lambda x.\, \begin{cases} 1 & x = 0 \\ x \times (g\,g\,(x-1)) & \text{otherwise.} \end{cases}$$

Sneakier self-reference

What if we can't use self-reference?

$$F = G\,G, \text{ where}$$
$$G = \lambda g.\, \lambda x.\, \begin{cases} 1 & x = 0 \\ x \times (g\,g\,(x-1)) & \text{otherwise.} \end{cases}$$

We use an "open" definition: G refers to whatever function we want it to use next. Passing G to itself gives G a reference to itself.

Sneakier self-reference

What if we can't use self-reference?

$$F = G\,G, \text{ where}$$
$$G = \lambda g.\,\lambda x.\,\begin{cases} 1 & x = 0 \\ x \times (g\,g\,(x-1)) & \text{otherwise.} \end{cases}$$

We use an "open" definition: G refers to whatever function we want it to use next. Passing G to itself gives G a reference to itself.

$$\begin{aligned} G\,G &= \left(\lambda g.\,\lambda x.\,\begin{cases} 1 & x = 0 \\ x \times (g\,g\,(x-1)) & \text{otherwise} \end{cases}\right) G \\ &= \lambda x.\,\begin{cases} 1 & x = 0 \\ x \times (G\,G\,(x-1)) & \text{otherwise.} \end{cases} \end{aligned}$$

Laziness is a virtue

Using a self-application $g\,g$ for every recursive call is cumbersome. What if we're lazy and want to write f instead of $g\,g$?

$$F = \lambda f.\,\lambda x.\,\begin{cases} 1 & x = 0 \\ x \times (f\,(x-1)) & \text{otherwise.} \end{cases}$$

Laziness is a virtue

Using a self-application $g\,g$ for every recursive call is cumbersome. What if we're lazy and want to write f instead of $g\,g$?

$$F = \lambda f.\,\lambda x.\begin{cases} 1 & x = 0 \\ x \times (f\,(x-1)) & \text{otherwise.} \end{cases}$$

F is a "shell" that needs to be filled in by the true factorial function, f. We can think of F as having "type"

$$(\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}).$$

Laziness is a virtue

Using a self-application $g\,g$ for every recursive call is cumbersome. What if we're lazy and want to write f instead of $g\,g$?

$$F = \lambda f.\, \lambda x.\, \begin{cases} 1 & x = 0 \\ x \times (f\,(x-1)) & \text{otherwise.} \end{cases}$$

F is a "shell" that needs to be filled in by the true factorial function, f. We can think of F as having "type"

$$(\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}).$$

If f computes factorial, then so does $F\,f$, in which case

$$f = F\,f.$$

That is, we necessarily want a fixed point of F.

Wishful thinking

Let's wish for a function Y that finds a fixed point of its input. Namely, Y should satisfy

$$Y\,F = F\,(Y\,F).$$

Wishful thinking

Let's wish for a function Y that finds a fixed point of its input. Namely, Y should satisfy

$$Y\,F = F\,(Y\,F).$$

When F is our factorial "shell", we get

$$Y\,F = F\,(Y\,F) = \lambda x.\begin{cases} 1 & x = 0 \\ x * (Y\,F\,(x-1)) & \text{otherwise.} \end{cases}$$

Wishful thinking

Let's wish for a function Y that finds a fixed point of its input. Namely, Y should satisfy

$$Y\,F = F\,(Y\,F).$$

When F is our factorial "shell", we get

$$Y\,F = F\,(Y\,F) = \lambda x.\begin{cases} 1 & x = 0 \\ x * (Y\,F\,(x-1)) & \text{otherwise.} \end{cases}$$

That is, to do recursion, it suffices to find a fixed point of F.

Wish really, really sneakily

We know how to do self-reference: make a function accept any reference as an argument, then feed the function to itself. With that inspiration, we wish for some G_F such that

$$Y\,F = F\,(Y\,F) = G_F\,G_F.$$

Y combinator? Because functions!

$$Y\,F = F\,(Y\,F) = G_F\,G_F$$

Y combinator? Because functions!

$$Y\,F = F\,(Y\,F) = G_F\,G_F$$

$$\Uparrow$$

$$G_F\,G_F = F\,(G_F\,G_F)$$

Y combinator? Because functions!

$$Y\,F = F\,(Y\,F) = G_F\,G_F$$

$$\Uparrow$$

$$\begin{aligned} G_F\,G_F &= F\,(G_F\,G_F) \\ &= (\lambda g.\,F\,(g\,g))\,G_F \end{aligned}$$

Y combinator? Because functions!

$$Y\,F = F\,(Y\,F) = G_F\,G_F$$

$$\Uparrow$$

$$\begin{aligned} G_F\,G_F &= F\,(G_F\,G_F) \\ &= (\lambda g.\,F\,(g\,g))\,G_F \end{aligned}$$

$$\Uparrow$$

$$G_F = \lambda g.\,F\,(g\,g).$$

Y combinator? Because functions!

$$Y\,F = F\,(Y\,F) = G_F\,G_F$$

$$\Uparrow$$

$$\begin{aligned} G_F\,G_F &= F\,(G_F\,G_F) \\ &= (\lambda g.\,F\,(g\,g))\,G_F \end{aligned}$$

$$\Uparrow$$

$$G_F = \lambda g.\,F\,(g\,g).$$

Theorem (Existence of the Y combinator)

The combinator

$$Y = \lambda f.\,G_f\,G_f = \lambda f.\,(\lambda g.f\,(g\,g))\,(\lambda g.f\,(g\,g))$$

satisfies $Y\,F = F\,(Y\,F)$ for all F.

The story so far

1. A very λ-calculus appetizer
2. Main theorem
3. Applications
4. Fixed points and diagonalization

Turing machines are computers, too!

Notation: $[n]$ is the Turing machine n encodes.

Theorem (Recursion theorem)

There exists a total (always halting) computable function Y such that for all F, if $[F]$ is total, then

$$[Y(F)] = [[F](Y(F))].$$

Turing machines are computers, too!

Notation: $[n]$ is the Turing machine n encodes.

Theorem (Recursion theorem)

There exists a total (always halting) computable function Y such that for all F, if $[F]$ is total, then

$$[Y(F)] = [[F](Y(F))].$$

Put another way: if we consider numbers equivalent if they encode equivalent Turing machines, then $[F]$ has a fixed point for all F, and that fixed point is computable from F.

Proof of recursion theorem

Notation: $[n]$ is the Turing machine n encodes, $x \mapsto E(x)$ is the procedure that maps x to expression $E(x)$, and $\langle P \rangle$ is the encoding of procedure P.

$$Y(F) = [F](Y(F)) = [G_F](G_F)$$

Proof of recursion theorem

Notation: $[n]$ is the Turing machine n encodes, $x \mapsto E(x)$ is the procedure that maps x to expression $E(x)$, and $\langle P \rangle$ is the encoding of procedure P.

$$Y(F) = [F](Y(F)) = [G_F](G_F)$$

$$\Uparrow$$

$$[G_F](G_F) = [F]([G_F](G_F))$$

Proof of recursion theorem

Notation: $[n]$ is the Turing machine n encodes, $x \mapsto E(x)$ is the procedure that maps x to expression $E(x)$, and $\langle P \rangle$ is the encoding of procedure P.

$$Y(F) = [F](Y(F)) = [G_F](G_F)$$

$$\Uparrow$$

$$\begin{aligned} [G_F](G_F) &= [F]([G_F](G_F)) \\ &= (g \mapsto [F]([g](g)))(G_F) \end{aligned}$$

Proof of recursion theorem

Notation: $[n]$ is the Turing machine n encodes, $x \mapsto E(x)$ is the procedure that maps x to expression $E(x)$, and $\langle P \rangle$ is the encoding of procedure P.

$$Y(F) = [F](Y(F)) = [G_F](G_F)$$

$$\Uparrow$$

$$\begin{aligned} [G_F](G_F) &= [F]([G_F](G_F)) \\ &= (g \mapsto [F]([g](g)))(G_F) \end{aligned}$$

$$\Uparrow$$

$$G_F = \langle g \mapsto [F]([g](g)) \rangle.$$

Proof of recursion theorem

Notation: $[n]$ is the Turing machine n encodes, $x \mapsto E(x)$ is the procedure that maps x to expression $E(x)$, and $\langle P \rangle$ is the encoding of procedure P.

$$Y(F) = [F](Y(F)) = [G_F](G_F)$$

$$\Uparrow$$

$$\begin{aligned}[G_F](G_F) &= [F]([G_F](G_F)) \\ &= (g \mapsto [F]([g](g)))(G_F)\end{aligned}$$

$$\Uparrow$$

$$G_F = \langle g \mapsto [F]([g](g)) \rangle.$$

We can compute G_F, so we win!

Proof of recursion theorem

Notation: $[n]$ is the Turing machine n encodes, $x \mapsto E(x)$ is the procedure that maps x to expression $E(x)$, and $\langle P \rangle$ is the encoding of procedure P.

$$Y(F) = [F](Y(F)) = [G_F](G_F)$$

$$\Uparrow$$

$$\begin{aligned}[G_F](G_F) &= [F]([G_F](G_F)) \\ &= (g \mapsto [F]([g](g)))(G_F)\end{aligned}$$

$$\Uparrow$$

$$G_F = \langle g \mapsto [F]([g](g)) \rangle.$$

We can compute G_F, so we win!
What happens if $[F](x) = x + 1$?

Proof of recursion theorem

Notation: $[n]$ is the Turing machine n encodes, $x \mapsto E(x)$ is the procedure that maps x to expression $E(x)$, and $\langle P \rangle$ is the encoding of procedure P.

$$Y(F) = [F](Y(F)) = [G_F](G_F)$$

$$\Uparrow$$

$$\begin{aligned} [G_F](G_F) &= [F]([G_F](G_F)) \\ &= (g \mapsto [F]([g](g)))(G_F) \end{aligned}$$

$$\Uparrow$$

$$G_F = \langle g \mapsto [F]([g](g)) \rangle.$$

We can compute G_F, so we win!

What happens if $[F](x) = x + 1$?

$Y(F) = Y(F) + 1$, so the universe explodes.

Proof of recursion theorem

Notation: $[n]$ is the Turing machine n encodes, $x \mapsto E(x)$ is the procedure that maps x to expression $E(x)$, and $\langle P \rangle$ is the encoding of procedure P.

$$Y(F) = [F](Y(F)) = [G_F](G_F)$$

$$\Uparrow$$

$$\begin{aligned}[G_F](G_F) &= [F]([G_F](G_F)) \\ &= (g \mapsto [F]([g](g)))(G_F)\end{aligned}$$

$$\Uparrow$$

$$G_F = \langle g \mapsto [F]([g](g)) \rangle.$$

We can compute G_F, so we win!
What happens if $[F](x) = x + 1$?
$Y(F) = Y(F) + 1$, so computing $Y(F) = [G_F](G_F)$ must not halt.

Proof of recursion theorem

Notation: $[n]$ is the Turing machine n encodes, $x \mapsto E(x)$ is the procedure that maps x to expression $E(x)$, and $\langle P \rangle$ is the encoding of procedure P.

$$Y(F) = [F](Y(F)) = [G_F](G_F)$$
$$\Uparrow$$
$$\begin{aligned}[G_F](G_F) &= [F]([G_F](G_F)) \\ &= (g \mapsto [F]([g](g)))(G_F)\end{aligned}$$
$$\Uparrow$$
$$G_F = \langle g \mapsto [F]([g](g)) \rangle.$$

We can compute G_F, so we win!
What happens if $[F](x) = x + 1$?
$Y(F) = Y(F) + 1$, so computing $Y(F) = [G_F](G_F)$ must not halt.
We tried to find a fixed point, but we only need a fixed point modulo equivalence of encoded Turing machines.

Proof of recursion theorem 2: electric boogaloo

We try again with a slightly weaker goal.

$$Y(F) = [G_F](G_F)$$
$$[Y(F)] = [[F](Y(F))]$$

Proof of recursion theorem 2: electric boogaloo

We try again with a slightly weaker goal.

$$Y(F) = [G_F](G_F)$$

$$[Y(F)] = [[F](Y(F))]$$

$$\Uparrow$$

$$[G_F](G_F) = \langle x \mapsto [[F]([G_F](G_F))](x) \rangle$$

Proof of recursion theorem 2: electric boogaloo

We try again with a slightly weaker goal.

$$
\begin{aligned}
Y(F) &= [G_F](G_F) \\
[Y(F)] &= [[F](Y(F))]
\end{aligned}
$$

$$\Uparrow$$

$$
\begin{aligned}
[G_F](G_F) &= \langle x \mapsto [[F]([G_F](G_F))](x)\rangle \\
&= (g \mapsto \langle x \mapsto [[F]([g](g))](x)\rangle)(G_F)
\end{aligned}
$$

Proof of recursion theorem 2: electric boogaloo

We try again with a slightly weaker goal.

$$
\begin{gathered}
Y(F) = [G_F](G_F) \\
[Y(F)] = [[F](Y(F))] \\
\Uparrow \\
\begin{aligned}
[G_F](G_F) &= \langle x \mapsto [[F]([G_F](G_F))](x) \rangle \\
&= (g \mapsto \langle x \mapsto [[F]([g](g))](x) \rangle)(G_F)
\end{aligned} \\
\Uparrow \\
G_F = \langle g \mapsto \langle x \mapsto [[F]([g](g))](x) \rangle \rangle .
\end{gathered}
$$

Proof of recursion theorem 2: electric boogaloo

We try again with a slightly weaker goal.

$$\begin{gathered}
Y(F) = [G_F](G_F) \\
[Y(F)] = [[F](Y(F))] \\
\Uparrow \\
\begin{aligned}
[G_F](G_F) &= \langle x \mapsto [[F]([G_F](G_F))](x)\rangle \\
&= (g \mapsto \langle x \mapsto [[F]([g](g))](x)\rangle)(G_F)
\end{aligned} \\
\Uparrow \\
G_F = \langle g \mapsto \langle x \mapsto [[F]([g](g))](x)\rangle\rangle.
\end{gathered}$$

This time, not only can we compute G_F, but $[G_F]$ is a total function, so computing $Y(F) = [G_F](G_F)$ always halts!

For the skeptical

Just to make sure we have this right, if $Y(F) = [G_F](G_F)$, then

$$\begin{aligned}
[Y(F)](X) &= [[G_F](G_F)](X) \\
&= [[\langle g \mapsto \langle x \mapsto [[F]([g](g))](x)\rangle\rangle](G_F)](X) \\
&= [\langle x \mapsto [[F]([G_F](G_F))](x)\rangle](X) \\
&= [[F]([G_F](G_F))](X) \\
&= [[F](Y(F))](X).
\end{aligned}$$

As desired, $Y(F)$ and $[F](Y(F))$ encode equivalent Turing machines.

The story so far

Recursion (duh)

Letting $[F](f) = \langle \dots [f] \dots \rangle$ gives

$$[Y(F)] = [[F](Y(F))] = \dots [Y(F)] \dots,$$

so $[Y(F)]$ can make recursive calls.

Recursion (duh)

Letting $[F](f) = \langle \ldots [f] \ldots \rangle$ gives

$$[Y(F)] = [[F](Y(F))] = \ldots [Y(F)] \ldots,$$

so $[Y(F)]$ can make recursive calls.
In general, $[F](f) = \langle \ldots f \ldots \rangle$ gives

$$[Y(F)] = [[F](Y(F))] = \ldots Y(F) \ldots,$$

which gives us something stronger: $[Y(F)]$ can access its own source code.
Practical application is that, when defining a procedure P, we can use $\langle P \rangle$ in the definition.

Quines

A Quine is a program that prints its own source code. This sounds easy at first, but after some trial and error it starts to seem impossible.

Quines

A Quine is a program that prints its own source code. This sounds easy at first, but after some trial and error it starts to seem impossible.
But we just said that defining $P(x) = \langle P \rangle$ is okay!

Quines

A Quine is a program that prints its own source code. This sounds easy at first, but after some trial and error it starts to seem impossible.
But we just said that defining $P(x) = \langle P \rangle$ is okay!
Letting $[F](f) = \langle x \mapsto f \rangle$ gives

$$[Y(F)](x) = [[F](Y(F))](x) = Y(F).$$

That is, $[Y(F)]$ always returns its encoding.

Quines

A Quine is a program that prints its own source code. This sounds easy at first, but after some trial and error it starts to seem impossible.
But we just said that defining $P(x) = \langle P \rangle$ is okay!
Letting $[F](f) = \langle x \mapsto f \rangle$ gives

$$[Y(F)](x) = [[F](Y(F))](x) = Y(F).$$

That is, $[Y(F)]$ always returns its encoding.
This example is simple enough that we can examine G_F directly.

$$G_F = \langle g \mapsto \langle x \mapsto [[F]([g](g))](x) \rangle \rangle = \langle g \mapsto \langle x \mapsto [g](g) \rangle \rangle.$$

Quines

A Quine is a program that prints its own source code. This sounds easy at first, but after some trial and error it starts to seem impossible.
But we just said that defining $P(x) = \langle P \rangle$ is okay!
Letting $[F](f) = \langle x \mapsto f \rangle$ gives

$$[Y(F)](x) = [[F](Y(F))](x) = Y(F).$$

That is, $[Y(F)]$ always returns its encoding.
This example is simple enough that we can examine G_F directly.

$$G_F = \langle g \mapsto \langle x \mapsto [[F]([g](g))](x) \rangle \rangle = \langle g \mapsto \langle x \mapsto [g](g) \rangle \rangle.$$

G_F says "apply my first argument, decoded, to itself, still encoded".
$Y(F) = [G_F](G_F)$ decodes G_F and applies it to a still encoded G_F.

Quines

A Quine is a program that prints its own source code. This sounds easy at first, but after some trial and error it starts to seem impossible.
But we just said that defining $P(x) = \langle P \rangle$ is okay!
Letting $[F](f) = \langle x \mapsto f \rangle$ gives

$$[Y(F)](x) = [[F](Y(F))](x) = Y(F).$$

That is, $[Y(F)]$ always returns its encoding.
This example is simple enough that we can examine G_F directly.

$$G_F = \langle g \mapsto \langle x \mapsto [[F]([g](g))](x) \rangle\rangle = \langle g \mapsto \langle x \mapsto [g](g) \rangle\rangle.$$

G_F says "apply my first argument, decoded, to itself, still encoded".
$Y(F) = [G_F](G_F)$ decodes G_F and applies it to a still encoded G_F.
This echoes the famous natural-language Quine: "quoted and followed by itself is a Quine" quoted and followed by itself is a Quine.

Impossibility results

Theorem (Rice's theorem)

No nontrivial predicate of Turing machines is decidable.

Impossibility results

Theorem (Rice's theorem)

No nontrivial predicate of Turing machines is decidable.

Proof.

Fix some nontrivial predicate decidable by P, and let $[A]$ and $[B]$ satisfy and not satisfy the predicate, respectively.

Impossibility results

Theorem (Rice's theorem)

No nontrivial predicate of Turing machines is decidable.

Proof.

Fix some nontrivial predicate decidable by P, and let $[A]$ and $[B]$ satisfy and not satisfy the predicate, respectively.
The procedure

$$Q(x) = \begin{cases} B & P(\langle Q \rangle) \\ A & \text{otherwise} \end{cases}$$

gives a contradiction. □

The story so far

1. A very λ-calculus appetizer
2. Main theorem
3. Applications
4. Fixed points and diagonalization

Yet another proof of Cantor's theorem

A surjection $\hat{a} : \mathbb{N} \to 2^{\mathbb{N}}$ gives a function $a : \mathbb{N} \times \mathbb{N} \to 2$ such that for any $b : \mathbb{N} \to 2$, there is some n such that $b = a(n, -)$, in which case we say that b is "representable" by a.

Yet another proof of Cantor's theorem

A surjection $\hat{a} : \mathbb{N} \to 2^{\mathbb{N}}$ gives a function $a : \mathbb{N} \times \mathbb{N} \to 2$ such that for any $b : \mathbb{N} \to 2$, there is some n such that $b = a(n, -)$, in which case we say that b is "representable" by a.
Let $\Delta(n) = (n, n)$ and $\sigma(p) = 1 - p$.

Yet another proof of Cantor's theorem

A surjection $\hat{a} : \mathbb{N} \to 2^{\mathbb{N}}$ gives a function $a : \mathbb{N} \times \mathbb{N} \to 2$ such that for any $b : \mathbb{N} \to 2$, there is some n such that $b = a(n, -)$, in which case we say that b is "representable" by a.
Let $\Delta(n) = (n, n)$ and $\sigma(p) = 1 - p$.

$$\begin{tikzcd}
\mathbb{N} \times \mathbb{N} \arrow[r, "a"] & 2 \arrow[d, "\sigma"] \\
\mathbb{N} \arrow[u, "\Delta"] \arrow[r, "b"'] & 2
\end{tikzcd}$$

Yet another proof of Cantor's theorem

A surjection $\hat{a} : \mathbb{N} \to 2^{\mathbb{N}}$ gives a function $a : \mathbb{N} \times \mathbb{N} \to 2$ such that for any $b : \mathbb{N} \to 2$, there is some n such that $b = a(n, -)$, in which case we say that b is "representable" by a.
Let $\Delta(n) = (n, n)$ and $\sigma(p) = 1 - p$.

$$\begin{tikzcd}
\mathbb{N} \times \mathbb{N} \arrow[r, "a"] & 2 \arrow[d, "\sigma"] \\
\mathbb{N} \arrow[u, "\Delta"] \arrow[r, "b"'] & 2
\end{tikzcd}$$

By construction, because σ has no fixed points, $b(n) \neq a(n, n)$ for all n, which means $b \neq a(n, -)$ for all n. Therefore, $\hat{a}$ is not a surjection.

Yet another proof the halting problem is undecidable

Suppose $a : \mathbb{N} \times \mathbb{N} \to 2$ decides the halting problem. That is, $a(F, X)$ is 1 if and only if $[F](X)$ halts.

Yet another proof the halting problem is undecidable

Suppose $a : \mathbb{N} \times \mathbb{N} \to 2$ decides the halting problem. That is, $a(F, X)$ is 1 if and only if $[F](X)$ halts.

Let $\sigma(0) = 1$ and $\sigma(1)$ never terminate.

Yet another proof the halting problem is undecidable

Suppose $a : \mathbb{N} \times \mathbb{N} \to 2$ decides the halting problem. That is, $a(F, X)$ is 1 if and only if $[F](X)$ halts.
Let $\sigma(0) = 1$ and $\sigma(1)$ never terminate.

$$
\begin{tikzcd}
\mathbb{N} \times \mathbb{N} \arrow[r, "a"] & 2 \arrow[d, "\sigma"] \\
\mathbb{N} \arrow[u, "\Delta"] \arrow[r, "b"'] & 2
\end{tikzcd}
$$

Yet another proof the halting problem is undecidable

Suppose $a : \mathbb{N} \times \mathbb{N} \to 2$ decides the halting problem. That is, $a(F,X)$ is 1 if and only if $[F](X)$ halts.
Let $\sigma(0) = 1$ and $\sigma(1)$ never terminate.

$$\begin{tikzcd}
\mathbb{N} \times \mathbb{N} \arrow[r, "a"] & 2 \arrow[d, "\sigma"] \\
\mathbb{N} \arrow[u, "\Delta"] \arrow[r, "b"'] & 2
\end{tikzcd}$$

By construction, because σ has no fixed points, $b(\langle b \rangle) \neq a(\langle b \rangle, \langle b \rangle)$. This manifests itself as the usual contradiction:

$$a(\langle b \rangle, \langle b \rangle) = 0 \quad \Leftrightarrow \quad b(\langle b \rangle) \text{ doesn't halt} \quad \Leftrightarrow \quad a(\langle b \rangle, \langle b \rangle) = 1$$

Yet another cookie-cutter diagonalization proof

The general picture, due originally to Lawvere and nicely explained by Yanofsky (`http://arxiv.org/abs/math/0305282`):

$$\begin{tikzcd} X \times X \arrow[r, "a"] & Y \arrow[d, "\sigma"] \\ X \arrow[u, "\Delta"] \arrow[r, "b"'] & Y \end{tikzcd}$$

$X =$ domain	$\Delta =$ diagonal	$a =$ "application"
$Y =$ "truth values"	$\sigma =$ shuffle	$b =$ "bad",

where "bad" means not representable as $a(x, -)$ for some x.

Cookies contrapositively cut

$$\begin{tikzcd} X \times X \arrow[r, "a"] & Y \arrow[d, "\sigma"] \\ X \arrow[u, "\Delta"] \arrow[r, "b"'] & Y \end{tikzcd}$$

Contrapositively, if b is representable as $a(x, -)$ for some x, then σ must have a fixed point.

A sketchier, boxier recursion theorem

Fix total computable P. Let $a(f,x) = [f](x)$ and $\sigma([f]) = \sigma([P(f)])$. (We strategically choose not to worry about how σ is well-defined as part of the composition but isn't on its own.)

$$\begin{tikzcd}
\mathbb{N}\times\mathbb{N} \arrow[r, "a"] & \mathrm{TM} \arrow[d, "\sigma"] \\
\mathbb{N} \arrow[u, "\Delta"] \arrow[r, "b"'] & \mathrm{TM}
\end{tikzcd}$$

A sketchier, boxier recursion theorem

Fix total computable P. Let $a(f,x) = [f](x)$ and $\sigma([f]) = \sigma([P(f)])$. (We strategically choose not to worry about how σ is well-defined as part of the composition but isn't on its own.)

$$\begin{tikzcd} \mathbb{N} \times \mathbb{N} \arrow[r, "a"] & \mathrm{TM} \arrow[d, "\sigma"] \\ \mathbb{N} \arrow[u, "\Delta"] \arrow[r, "b"'] & \mathrm{TM} \end{tikzcd}$$

All of Δ, a, and σ do straightforward computations, so b is computable and therefore representable as $a(\langle b \rangle, -)$.

A sketchier, boxier recursion theorem

Fix total computable P. Let $a(f,x) = [f](x)$ and $\sigma([f]) = \sigma([P(f)])$. (We strategically choose not to worry about how σ is well-defined as part of the composition but isn't on its own.)

$$\begin{tikzcd} \mathbb{N} \times \mathbb{N} \arrow[r, "a"] & \mathrm{TM} \arrow[d, "\sigma"] \\ \mathbb{N} \arrow[u, "\Delta"] \arrow[r, "b"'] & \mathrm{TM} \end{tikzcd}$$

All of Δ, a, and σ do straightforward computations, so b is computable and therefore representable as $a(\langle b\rangle, -)$.
This means σ has a fixed point, or, equivalently, P has a fixed point modulo equivalence of encoded Turing machines.