Computability with higher-order functions

Daniel Gratzer

Tuesday 29^{th} April, 2025

Turings Venner

Hi!

- I'm an assistant professor in the logsem group/PL^{2}S^2 section
- I'm interested in programming languages.

Hi!

- I'm an assistant professor in the logsem group/PL^2S^2 section
- I'm interested in programming languages.
- "how do (weird) programming language interact with (weirder) geometry."
- Key words: type theory, higher category theory, homotopy theory.

Hi!

- I'm an assistant professor in the logsem group/PL^{2}S^2 section
- I'm interested in programming languages.
- "how do (weird) programming language interact with (weirder) geometry."
- Key words: type theory, higher category theory, homotopy theory.

Real type theory, done by real type theorists:

$$
\begin{aligned}
&\hom_{\widehat{C}}(X, f^* d_* Z) \\
&\simeq \textstyle\prod_{c:\langle \mathrm{op} | C \rangle} X(c) \to \hom(f^\dagger c, d) \to Z \\
&\simeq \left(\textstyle\sum_{c:\langle \mathrm{op} | C \rangle} X(c) \times \hom(f^\dagger c, d)\right) \to Z
\end{aligned}
$$

Good news: I'm not talking about type theory today



Today's talk:

- not my work!
- not super proof-y/rigorous
- mostly from a blogpost I read 10+ years ago.
- we get to where we get to; just interrupt with questions ☺

Real goal: show something I found surprising and exciting

A bit of history

What sort of problems admit *computable* solutions?

A bit of history

What sort of problems admit *computable* solutions?

- considered informally in math for a *long* time
- lots of attention in the early 20^{th} century
- NB: predates computers as we know them

Turing Machines

Definition

A partial function $\mathbb{N} \rightharpoonup \mathbb{N}$ is computable if there is a *Turing machine* tracking it.

Theorem (Church and Turing)

A bunch of reasonable-looking problems are not computable.

Turing Machines

Definition

A partial function $\mathbb{N} \rightharpoonup \mathbb{N}$ is computable if there is a *Turing machine* tracking it.

Theorem (Church and Turing)

A bunch of reasonable-looking problems are not computable.

Small quiz, do we know who those people are?

Turing Machines



Definition

A partial function $\mathbb{N} \rightharpoonup \mathbb{N}$ is computable if there is a *Turing machine* tracking it.

Theorem (Church and Turing)

A bunch of reasonable-looking problems are not computable.

Why is this a good definition though?

The Chuch–Turing Hypothesis

Turing machines unintuitive, I propose the following instead:

Definition (Daniel computability)

$f : \mathbb{N} \rightharpoonup \mathbb{N}$ is computable if Daniel can guess $f(n)$ for each n in ≤ 3 seconds.

More seriously: why not some other definition?

The Chuch–Turing Hypothesis

We hypothesize all reasonable ones coincide:

Thesis

All effective models of computation encode the same computable functions $\mathbb{N} \rightharpoonup \mathbb{N}$.

Primary evidence:

Theorem (Church, Turing, Rosser, ...)

Turing machines, the λ-calculus, Post's machines, ... all recover the same functions

So, why is this a good definition? It's incredibly robust

A bit of history: II

And thus the problem of what computation is was solved forever...

A bit of history: II

We actually have two problems to consider:

1. How do we describe *data* (how do we describe/encode input and output)
2. How do we describe *computation*

Thus far, we've basically assumed that our data was various natural numbers.

In defense of $\mathbb{N}$

- We can encode a lot of stuff using just $\mathbb{N}$
- As CS people, we encode stuff as a bunch of bits all the time!

So... why care about anything besides $\mathbb{N}$?

- We can encode a lot of stuff using just $\mathbb{N}$
- As CS people, we encode stuff as a bunch of bits all the time!
- Mathematician term: Gödel encoding

So... why care about anything besides $\mathbb{N}$?

Leaky encodings...

Two basic ways we could fail to adequately encode something:

- Need to make sure every widget w is represented by some n ($n \Vdash w$).
- If $n \Vdash w$, we need to compute just the expected operations for widgets on n.

Leaky encodings...

Two basic ways we could fail to adequately encode something:

- Need to make sure every widget w is represented by some n ($n \Vdash w$).
- If $n \Vdash w$, we need to compute just the expected operations for widgets on n.

Leaky encodings...

Two basic ways we could fail to adequately encode something:

- Need to make sure every widget w is represented by some n ($n \Vdash w$).
- If $n \Vdash w$, we need to compute just the expected operations for widgets on n.

The halting problem is decidable... if we encode a TM with a halting bit in front!

First-class functions

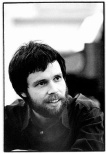
Where this comes to a head: computing with functions.

- The halting problem etc., are about computing with source-code
- Can we study what it means to just compute with functions on their own?

Serious problems encoding these as natural numbers if we want *all* functions...

If this was a math-y talk

I can't resist just defining computability structures:



Definition (via Longley–Normann)

A computability structure C basically describes

- A collection of types T
- A bunch of sets C_τ describing the values of type τ.
- A predicate comp : $(C_\tau \rightharpoonup C_\sigma) \to \{\top, \bot\}$ telling us what's C-computable.
- We insist that C-computable functions contain (1) the identity and (2) compose.

Just a way to talk about computing with different data.

This is not a math-y talk

Really, what we want to think about is a programming language!

- We have a bunch of types and values/constants
- We know how to run a program on an input and inspect the result.

Our goal: study what programs we can write at types other than `nat`.

Encodings matter

The bitter truth:

Theorem

The Church–Turing thesis does ***not*** *extend to computability at higher type.*[1]

This is actually true already for the λ-calculus and Turing machines.

(Be careful though! This is sensitive to how we encode functions)

[1] One way to mathematize this: $\mathbf{RT}(\mathcal{K}_1)$ is not equivalent to $\mathbf{RT}(\mathcal{K}_2)$.

An example

Our goal for the rest of the day: give an example such a divergence.

- To describe our example, we need a programming language for it.
- I am a PL person; become agitated if my talk doesn't introducing a new language.

Let's define a baby functional programming language to work in.

Questions? 5 minute break?

A minimal, statically-typed, functional language

I want a language with the following types:

1. Natural numbers `nat`
2. Booleans `bool`
3. First-class functions $\tau \to \sigma$

The basic values of these types are as follows:

$$\texttt{true} : \texttt{bool} \quad \texttt{false} : \texttt{bool}$$

$$\bar{n} : \texttt{nat}$$

$$\texttt{fn}\, x \to e : \tau \to \sigma$$

Primitive expressions

Besides values, we have a few key program constructs we'll use:

- Recursive functions
- $\texttt{if}\ b\ \texttt{then}\ e_t\ \texttt{else}\ e_f$
- $\texttt{inc}, \texttt{dec} : \texttt{nat} \to \texttt{nat}$
- $\texttt{isZero} : \texttt{nat} \to \texttt{bool}$

Using these we can define things like addition, equality operators for `nat` and `bool`.

Primitive expressions

Besides values, we have a few key program constructs we'll use:

- Recursive functions
- $\texttt{if}\ b\ \texttt{then}\ e_t\ \texttt{else}\ e_f$
- $\texttt{inc}, \texttt{dec} : \texttt{nat} \to \texttt{nat}$
- $\texttt{isZero} : \texttt{nat} \to \texttt{bool}$

Using these we can define things like addition, equality operators for $\texttt{nat}$ and $\texttt{bool}$.

Sometimes called PCF

An example: factorial

$$
\begin{aligned}
&\texttt{let } n + m = \\
&\quad \texttt{if isZero}(n) \texttt{ then } m \texttt{ else inc}(\texttt{dec}(n) + m) \\
&\texttt{let } n * m = \\
&\quad \texttt{if isZero}(n) \texttt{ then } 0 \texttt{ else } m + (\texttt{dec}(n) * m) \\
&\texttt{let } \mathsf{factorial}\ n = \\
&\quad \texttt{if isZero}(n) \texttt{ then } 1 \texttt{ else } n * \mathsf{factorial}(\texttt{dec}(n))
\end{aligned}
$$

An example: factorial

$$\texttt{let}\ n + m =$$
$$\quad \texttt{if isZero}(n)\ \texttt{then}\ m\ \texttt{else inc}(\texttt{dec}(n) + m)$$
$$\texttt{let}\ n * m =$$
$$\quad \texttt{if isZero}(n)\ \texttt{then}\ 0\ \texttt{else}\ m + (\texttt{dec}(n) * m)$$
$$\texttt{let}\ \text{factorial}\ n =$$
$$\quad \texttt{if isZero}(n)\ \texttt{then}\ 1\ \texttt{else}\ n * \text{factorial}(\texttt{dec}(n))$$

An example: factorial

$$\texttt{let}\ n + m =$$
$$\quad \texttt{if isZero}(n)\ \texttt{then}\ m\ \texttt{else inc}(\texttt{dec}(n) + m)$$
$$\texttt{let}\ n * m =$$
$$\quad \texttt{if isZero}(n)\ \texttt{then}\ 0\ \texttt{else}\ m + (\texttt{dec}(n) * m)$$
$$\texttt{let}\ \text{factorial}\ n =$$
$$\quad \texttt{if isZero}(n)\ \texttt{then}\ 1\ \texttt{else}\ n * \text{factorial}(\texttt{dec}(n))$$

We're using Currying for multi-argument functions; could also just add pairs.

Now back to the good part

$$\texttt{type}\ \text{BitStream} = \texttt{nat} \to \texttt{bool}$$

$$\texttt{type}\ \text{BitStreamPred} = \text{BitStream} \to \texttt{bool}$$

Our main result:

Theorem

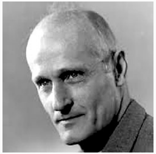
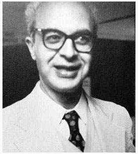
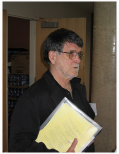
We can decide the equality of total BitStreamPreds*:*

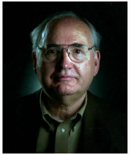
$$\text{eq} : \text{BitStreamPred} \to \text{BitStreamPred} \to \texttt{bool}$$

Who worked this out?

This result is a bit hard to attribute precisely; it has many related incarnations

In addition to Berger, Escardó, and Simpson, these people are certainly relevant:



A few words about totality

All of these operators are hereditarily total:

- For ground types (`bool`, `nat`), hereditary totality is just termination.
- For $\tau \to \sigma$, HT means mapping HT inputs to HT outputs.

Example

An HT BitStreamPred needs to terminate only when given an HT bitstream

Warning!

All of our operators may do whatever they want on non-HT inputs

Is this expected?

No, this is actually very weird:

- BitStream does not admit decidable equality.
- $(\texttt{nat} \to \texttt{nat}) \to \texttt{bool}$ does not admit decidable equality.
- This is *not true* for Turing machine model!

Consequence: $\texttt{nat} \to \texttt{bool}$ and $\texttt{nat} \to \texttt{nat}$ are not equivalent!

What's special about BitStreamPred?

Key Idea

The only way to use a function $f : \tau \to \sigma$ is to apply f.

- In particular: all we can do with f : BitStream is query bits
- If $\Phi(f)$: BitStreamPred terminates, then Φ queries only finitely many bits of f.
- In fact, upper bound for these queries across all fs

Modulus of uniform continuity

Theorem

If $\Phi : \mathrm{BitStreamPred}$ *is HT, then there is N such that:*

$$\forall f, g : \mathrm{BitStream}.\, f, g \text{ are } HT \to (\forall i \leq N.\, f(i) = g(i)) \to \Phi(f) = \Phi(g)$$

Informally: Φ *never looks past N bits of its input.*

N is the *modulus of uniform continuity* of Φ.

Modulus of uniform continuity

Theorem

If $\Phi : \mathrm{BitStreamPred}$ *is HT, then there is* N *such that:*

$$\forall f, g : \mathrm{BitStream}.\, f, g \textit{ are HT} \rightarrow (\forall i \leq N.\, f(i) = g(i)) \rightarrow \Phi(f) = \Phi(g)$$

Informally: Φ *never looks past* N *bits of its input.*

N is the *modulus of uniform continuity* of Φ.

"Proof".

To the board! $\square$

Are we then done?

Moral idea

To compute $\Phi = \Psi$, suffices to check output on finitely many cases ($2^{\max(N_\Phi, N_\Psi)}$).

This... isn't enough.

- Given Φ, if we could *compute* N, that'd be good.
- This is possible
- Subtle differences in definition of modulus of uniform continuity matter

We'll be more indirect, but it only works because of N's existence.

Escardó's and Simpson's approach

We'll break this into defining the following functions:

$$\text{search} : \text{BitStreamPred} \to \text{BitStream}$$

$$\text{forall} : \text{BitStreamPred} \to \text{bool}$$

$$\text{exists} : \text{BitStreamPred} \to \text{bool}$$

We'll define these three functions

Escardó's and Simpson's approach

We'll break this into defining the following functions:

$$\text{search} : \text{BitStreamPred} \to \text{BitStream}$$

$$\text{forall} : \text{BitStreamPred} \to \text{bool}$$

$$\text{exists} : \text{BitStreamPred} \to \text{bool}$$

Find an example satisfying this predicate, otherwise return junk

Escardó's and Simpson's approach

We'll break this into defining the following functions:

$$\mathsf{search} : \mathsf{BitStreamPred} \to \mathsf{BitStream}$$

$$\mathsf{forall} : \mathsf{BitStreamPred} \to \mathsf{bool}$$

$$\mathsf{exists} : \mathsf{BitStreamPred} \to \mathsf{bool}$$

Check whether a predicate is always true

Escardó's and Simpson's approach

We'll break this into defining the following functions:

$$\text{search} : \text{BitStreamPred} \rightarrow \text{BitStream}$$

$$\text{forall} : \text{BitStreamPred} \rightarrow \text{bool}$$

$$\text{exists} : \text{BitStreamPred} \rightarrow \text{bool}$$

Check whether a predicate is ever true

Deciding equality

Suppose that we have search, forall, and exists:

$$\text{eq} : \text{BitStreamPred} \to \text{BitStreamPred} \to \texttt{bool}$$
$$\text{eq}\ \Phi\ \Psi = \text{forall}(\texttt{fn}\ s \to \Phi(s) = \Psi(s))$$

Moral: two predicates being equal everywhere can be expressed as a third predicate!

Deciding equality

Suppose that we have search, forall, and exists:

$$\mathrm{eq} : \mathrm{BitStreamPred} \to \mathrm{BitStreamPred} \to \texttt{bool}$$

$$\mathrm{eq}\ \Phi\ \Psi = \mathrm{forall}(\texttt{fn}\ s \to \Phi(s) = \Psi(s))$$

Moral: two predicates being equal everywhere can be expressed as a third predicate!

search **to** exists **to** forall

Let's assume search for a bit:

$$\text{exists}, \text{forall} : \text{BitStreamPred} \to \texttt{bool}$$

$$\text{exists}\,\Phi = \Phi(\text{search}(\Phi))$$

$$\text{forall}\,\Phi = \text{not}(\text{exists}(\texttt{fn}\ s \to \text{not}(\Phi(s))))$$

Let's assume search for a bit:

$$\text{exists}, \text{forall} : \text{BitStreamPred} \to \texttt{bool}$$

$$\text{exists}\,\Phi = \Phi(\text{search}(\Phi))$$

$$\text{forall}\,\Phi = \text{not}(\text{exists}(\texttt{fn}\ s \to \text{not}(\Phi(s))))$$

Something is always true if it's not the case that it's ever false.

Questions? 5 minute break?

The main act: search

Now, at last, we arrive at search.

Big idea:

- First, run search on $\texttt{fn}\ s \to \Phi(\texttt{false} \rhd s)$[2]
- If the result s actually satisfies $\Phi(\texttt{false} \rhd -)$, return $\texttt{false} \rhd s$.
- Otherwise, return whatever we can find for $\texttt{fn}\ s \to \Phi(\texttt{true} \rhd s)$ with $\texttt{true}$ append.

[2] $\rhd$ appends something to the start: $(b \rhd s)\, n = \texttt{if}\ \texttt{isZero}(n)\ \texttt{then}\ b\ \texttt{else}\ s(\texttt{dec}(n))$

The main act: search

Now, at last, we arrive at search.

Big idea:

- First, run search on $\texttt{fn}\ s \to \Phi(\texttt{false} \triangleright s)$[2]
- If the result s actually satisfies $\Phi(\texttt{false} \triangleright -)$, return $\texttt{false} \triangleright s$.
- Otherwise, return whatever we can find for $\texttt{fn}\ s \to \Phi(\texttt{true} \triangleright s)$ with $\texttt{true}$ append.

$$
\begin{aligned}
&\texttt{let}\ \text{search}\ s = \\
&\quad \texttt{if}\ \Phi(\text{search}(\texttt{fn}\ s \to \Phi(\texttt{false} \triangleright s))) \\
&\quad \texttt{then false} \triangleright \text{search}(\texttt{fn}\ s \to \Phi(\texttt{false} \triangleright s)) \\
&\quad \texttt{else true} \triangleright \text{search}(\texttt{fn}\ s \to \Phi(\texttt{true} \triangleright s))
\end{aligned}
$$

[2] $\triangleright$ appends something to the start: $(b \triangleright s)\ n = \texttt{if}\ \texttt{isZero}(n)\ \texttt{then}\ b\ \texttt{else}\ s(\texttt{dec}(n))$

Example 1: a constant predicate

If $\Phi(s) = \texttt{true}$, what happens?

- $\Phi(\ldots)$ will always be true, so immediately get to `then` clause.
- Now return $\texttt{false} \triangleright \text{search}(\texttt{fn}\ \_ \to \texttt{true})$.
- Clearly HT: just going to keep yield the stream of `false`s

Example 1: a constant predicate

If $\Phi(s) = \texttt{true}$, what happens?

- $\Phi(\ldots)$ will always be true, so immediately get to `then` clause.
- Now return $\texttt{false} \triangleright \text{search}(\texttt{fn}\ \_ \to \texttt{true})$.
- Clearly HT: just going to keep yield the stream of `false`s

Moment of thought: $\text{search}(\texttt{fn}\ \_ \to \texttt{false})$ yields a stream of `true`s.

Example 2: depth 1 predicate

If $\Phi(s) = s(0)$, what happens?

- $\Phi(\texttt{false} \triangleright -) = \texttt{fn \_} \to \texttt{false}$, so first `if` will send us to `else`
- We're now computing $\texttt{true} \triangleright \text{search}(\texttt{fn \_} \to \texttt{true})$.
- Back to the previous case: now have `true` followed by only `false`s

Proof sketch of hereditary termination

The general argument:

- We are making recursive calls to search $\Phi(\texttt{false} \triangleright -)$,$\Phi(\texttt{false} \triangleright -)$
- If Φ is depth d, these are depth $d-1$.
- We can inductively argue hereditary termination from this.

Crucial point: since every Φ has a modulus of uniform continuity, all have finite depth.[3]

[3] We actually need the *intensional* version of the modulus of uniform continuity. Don't worry about it.

What goes wrong with `nat` $\to$ `nat`?

Here is a function `nat` $\to$ `nat` which is not uniformly continuous:

$$f\ \ s = s(s(0))$$

This is where the argument breaks down for deciding (`nat` $\to$ `nat`) $\to$ `bool`.

Questions?

Interlude: some OCaml code

In which Daniel bravely attempts to do some live coding.

How on Earth did people work this out?

How did Escardó come up with this code/the more complex searches?

- not (just) by meditating on functional programs
- there is actually mathematical reasoning behind it!

In fact, a lot of what we've just argued stems from a foundational topological result:

Theorem

The Cantor space $C \subseteq [0, 1]$ is compact.

Psych, it was a math-y talk after all

Key Ideas

Effective computation is *continuous*

In the case of PCF, we have an (adequate) model where:

- HT elements of BitStream are roughly C
- HT elements of BitStreamPred are roughly continuous functions $C \to \{0, 1\}$

Compactness upgrades "continuous" to "uniformly continuous" and the rest unfolds.

The role of topology

From Escardó:

> *Thus, in a more abstract level, topology is applied as a paradigm for discovering unforeseen notions, algorithms and theorems in computability theory.*

The role of topology

From Escardó:

> *Thus, in a more abstract level, topology is applied as a paradigm for discovering unforeseen notions, algorithms and theorems in computability theory.*

- Very much ongoing! (Algebraic/differential geometry, stone spaces, ∞-categories)
- The connection between computation and geometry is deep & profound.

Curious?

Where I learned of this (by Martín Escardó)

- https://math.andrej.com/2007/09/28/seemingly-impossible-functional-programs/
- *Infinite sets that admit fast exhaustive search*
- *Exhaustible sets in higher-type computation*

Lots of relevant and interesting stuff on Andrej Bauer's blog!

Very curious?

- Gunter: *Semantics of Programming Languages*
- Vickers: *Topology via Logic*
- Abramsky & Jung: *Domain Theory*
- Longley & Normann: *Higher-order Computability*
- Van Oosten: *Realizability theory: an introduction to its categorical side*
- Pratchett: *Going postal*

Curious and don't like reading?

My office is Turing 127. Always happy to chat ☺

Thanks

Berger

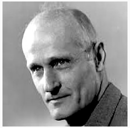
Church

Curry

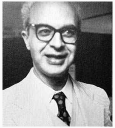
Ershov

Escardó

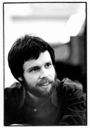
Gödel



Kleene



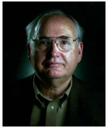
Kreisel



Longley

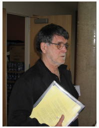
Normann

Post



Scott

Simpson



Tait