Formalizing a generalized diagonalization argument in Isabelle/HOL

KAĞAN GÜLSÜM, University of Twente, The Netherlands

Abstract

One of the most important theorems of 19^{th} century mathematics is Cantor's theorem. The theorem set forth new areas of study within mathematics; it influenced set theory, shaped important aspects of mathematics philosophy, and affected many more areas. A powerful tool first used by Cantor in his theorem was the diagonalization argument, which can be applied to different contexts through category-theoretic or set-theoretic abstraction, as shown by Lawvere and Yanofsky, respectively. For instance, an interesting context that leverages this argument is Turing's Halting problem. In this research, we express a set-theoretic conception of the argument with a proof assistant, namely Isabelle/HOL, and formalize various theorems which touch upon it in some ways. Then we realize certain weaknesses of our abstraction when trying to derive the Halting problem. In turn, we come up with a novel framework that circumvents those weaknesses.

1 INTRODUCTION

Proof assistants or interactive theorem provers are software tools that formalize mathematics. They present an expressive syntax that can capture mathematical statements and, via logical calculus, try to verify their correctness. [2] One of the most widely used interactive theorem prover is Isabelle, written in Standard ML and Scala. There are different logical calculi Isabelle variants build on. We use the Higher Order Logic variant, Isabelle/HOL. On top, Isabelle constitutes a formal proof language called Isar; essentially, it is a generic framework for creating human-readable proof documents with full proof checking. [8]

Using Isabelle, one can show, for example, Cantor's theorem. Cantor's theorem is considered to be one of the most important theorems of 19^{th} century mathematics, as it opened up doors to vastly different studies. The theorem originally states that there are different sizes of infinity. [1] An argument used within the theorem inspired many groundbreaking theorems later on. The argument is called the diagonalization argument, which can be seen to be leveraged from Gödel's incompleteness theorems [4] to Turing's Halting problem [6].

The fact that Cantor initially put forth a line of reasoning that can be used in different contexts prompted possibilities to obtain an abstract version of the argument. Lawvere initially wrote down a general version using category theory. [4] Afterwards, Yanofsky simmered that down to a set-theoretic language. [10]

In this research, we use Isabelle to express the set-theoretic abstraction of the diagonalization argument formally. Afterwards, we instantiate this abstraction with different mathematical objects in order to realize various theorems. Then, we spot certain shortcomings in the framework and construct a stronger one to capture Turing's Halting problem.

Contribution

The contributions of this paper are the formalization of the following mathematical results in Isabelle/HOL:

(1) Yanofsky's set-theoretic abstraction of the diagonalization argument.

(2) Instances of the abstracted diagonalization argument, such as Cantor's theorem, Russell's Paradox, and the existence of non-recursively enumerable languages.

(3) A stronger version of the argument with minimally structured carrier sets.

(4) Turing's Halting problem via instantiating the stronger version.

All of which can be found in the following link:
https://github.com/Chmlgy/generalized_cantor

2 PRELIMINARY KNOWLEDGE

2.1 Mathematics

We use the classic representation of power sets as a set of characteristic functions. That is, for any set A, $\mathcal{P}(A) \equiv \{f \mid f : A \to \mathbf{2}\}$, where $\mathbf{2} = \{0, 1\}$. One can think of each function in the set as a characteristic function for a subset of A. The function that maps each member to 1 admits A; the function that maps each member to 0 admits $\varnothing$, etc.

2.2 Isabelle/HOL

2.2.1 *Isabelle.*

Isabelle's text has two parts: inner syntax (inside "") and outer syntax.

The outer syntax has important keywords to know. **theorem** and **lemma** specify that what follows is a statement which will be proven. After the theorem/lemma statement, we can fix certain variable's type using the **fixes** keyword. Following that, we can assume certain facts using **assumes**. And after fixing and assuming, we can tell what the theorem eventually proves via the keyword **shows**. Within the proof state, we can use the keyword **apply** to command Isabelle to use a certain proof method, like unification. **by** similarly commands Isabelle, but the proof state is completed right after. **definition** specifies a non-recursive function definition. **locale**

TScIT 39, July 7, 2023, Enschede, The Netherlands

specifies a parametric theory. It fixes certain constructions and assumes their behaviour; one can prove things in that specific context using them. A locale can be instantiated and used as a framework for different varieties of mathematical objects that satisfy its context.

The inner syntax is ML-based. So familiarity with a functional programming language makes it easier to read statements. λ-abstractions, if-then-else, and case statements work the same way as they would in any other functional PL. The data structure syntax is easily understandable. The symbol `@` is list comprehension. $\bigwedge$ can simply be interpreted as the universal quantifier. $[\![\cdot]\!]$ is used to list assumptions for a theorem/lemma followed by an implication. `SOME` is Hilbert's choice operator.

2.2.2 *Universal Turing Machine library.*

We make use of the Universal Turing Machine library, which formalizes Turing machines. [9] The type of Turing machines we work with is `tprog0`. They are simply an instruction list. The tapes are formalized as a pair of cell lists. Cells can be in one of two states, blank (`Bk`) or occupied (`Oc`). ↑ is used to create repeating cells, for example, `Bk ↑ 2 = [Bk, Bk]`. Tapes consist of a pair of lists because the head of the Turing machine is assumed to be at the beginning of the latter list.

A Gödel encoding for Turing machines exists – a function called `tm_to_nat`. An encoder from natural numbers to cell lists exists – shown as `<n>`; this encoder can also encode pairs of natural numbers as `<(n1, n2)>` by putting a `Bk` in between them. This encoding is heavily used to create input tapes.

Composability of Turing machines makes up an important part of the theory but is left out of the discussion in this paper. The only thing to know is the `composable_tm0` function that takes in a machine and returns True if it is composable. Assume the machines we work with are composable. The operator `|+|` is used to chain machines together. The machines are run from left to right when they are composed, unlike function composition.

Hoare triples are used to reason about the behaviour of machines. The syntax for it is: {[Q1]}p{[Q2]}. If the machine does not halt, then we show it by: {[Q1]}p↑. p is the machine. Q1 and Q2 are assertions on what the input and output tapes conform to, respectively. Assertions are usually λ-abstractions that take in a tape and specify its structure. Importantly, Hoare triples can be composed as follows:

{[Q1]}p{[Q2]} ∧ {[Q2]}q{[Q3]} ⟹ {[Q1]}p|+|q{[Q3]}
{[Q1]}p{[Q2]} ∧ {[Q2]}q↑ ⟹ {[Q1]}p|+|q↑

3 METHODOLOGY

The method of formalization is the Isabelle language. The subject of the formalization is the generalized diagonalization argument. The approach is to first clearly lay out the mathematical structure of the argument with the utmost clarity, as close to a formal proof [3] as it gets.

For this, consider the commutative diagram [10] below; it captures the essence of the argument. The framework we wish to formalize is built on this idea.

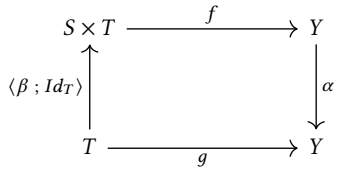

Fig. 1. The key insight of the set-theoretic generalized diagonal argument.

Let's curry the function f to understand what is happening here. We can call this function $f_{curried}$, which is of the form $S \to (T \to Y)$. The usual goal of the argument is to show that $f_{curried}$ does not map to g even though g is of the form $T \to Y$, i.e. we try to show that $f_{curried}$ is not surjective. To achieve that result, one must construct g carefully. As the commutative diagram shows, $g := \alpha \circ f \circ \langle \beta\,;\, Id_T \rangle$. Here $\langle \beta\,;\, Id_T \rangle$ stands for diagonalization – the input is fed into those two functions, and a pair is given as output. $\beta : T \to S$ is assumed to be surjective, meaning there exists a right inverse $\overline{\beta} : S \to T$ with $\beta \circ \overline{\beta} = Id_T$. α is a bit more interesting; usually, it is regarded as logical negation, but in the general case, it is any function from Y to Y that satisfies the condition $\forall y \in Y.\ y \neq \alpha(y)$. In other words, α has no fixed points. The possibility of defining such α makes Y a non-degenerate set, by definition.

Now we can prove the goal. We assume, for contradiction, $f_{curried}$ is surjective. If that is the case, a $s \in S$ exists such that $f_{curried}(s) = g$. Meaning we can write $f_{curried}(s)(\overline{\beta}(s)) = g(\overline{\beta}(s))$, as $\overline{\beta}(s) \in T$, but by definition of g that means:

$$\begin{aligned} f_{curried}(s)(\overline{\beta}(s)) &= (\alpha \circ f \circ \langle \beta\,;\, Id_T \rangle)(\overline{\beta}(s)) \\ &= (\alpha \circ f)(\beta(\overline{\beta}(s)), \overline{\beta}(s)) \\ &= \alpha(f(s, \overline{\beta}(s))) \end{aligned}$$

Uncurrying the left-hand side gives us $f(s, \overline{\beta}(s)) = \alpha(f(s, \overline{\beta}(s)))$. But α by assumption has no fixed points — contradiction.

We need to translate this proof into a formal one to express in Isabelle. A representation closer to formal proof is easier for a machine to realize, even though it can be unnecessary for humans.

Realize not much has been said for the underlying sets S, T and Y, except that Y is non-degenerate. This is the power of this framework. This way, the argument can be invoked in different contexts to prove insights for distinct objects. We can derive specific insights about the power of f, or – better phrased – the limited power of f. The impossibility of its surjectivity can often be regarded as a counting problem, giving us the statement $|S| < |\{h \mid h : T \to Y\}|$. And sometimes, the reason behind not being able to map to a specific object reveals the limits of what can exist in our set in the first place.

Later in the paper, we showcase some weaknesses of this method and devise a new one. For now, the following two sections implement the discussed method in Isabelle and interpret various mathematical results using it.

4 FORMALIZATION OF THE ARGUMENT

4.1 Abstracted Diagonalization Argument

The proof given in the above section can be formalized in Isabelle as follows:

```
theorem "abstracted_cantor":
  fixes f :: "'b ⇒ 'a ⇒ 'c" and α :: "'c ⇒ 'c"
  and β :: "'a ⇒ 'b" and β̄ :: "'b ⇒ 'a"
  assumes surjectivity: "surj f"
  and no_fixed_point: "∀y. α y ≠ y"
  and right_inverse: "∀s. β (β̄ s) = s"
  shows "False"
```

Here `'a` corresponds to T, `'b` to S, and `'c` to Y.

4.2 Instantiated Versions

We do not always need all the variables in the `fixes` clause of the theorem `abstracted_cantor`. For example, when we wish to prove the classic result of Cantor, we are interested in just showing $|T| < |\mathcal{P}(T)|$. Using the alternative representation of the power set, this boils down to $|T| < |\{h \mid h : T \to \mathbf{2}\}|$.

Below is an instantiation of `abstracted_cantor` that gives us $|T| < |\{h \mid h : T \to Y\}|$, where we plug in the identity for β and $\overline{\beta}$. And interpret f's type not as $(S \times T) \to Y$ but as $(T \times T) \to Y$. Here g becomes $g := \alpha \circ f \circ \Delta$ where $\Delta = \langle Id; Id \rangle$. Δ comes up later in the paper; it simply duplicates any input it takes.

```
theorem "generalized_cantor":
  fixes f :: "'a ⇒ 'a ⇒ 'b" and α :: "'b ⇒ 'b"
  assumes surjectivity: "surj f"
  and no_fixed_point: "∀y. α y ≠ y"
  shows "False"
```

Then, we can instantiate `generalized_cantor` such that α becomes $\neg$ and `'b` becomes `bool` to reach exactly at the classic result of Cantor – that is for any set T, $|T| < |\{h \mid h : T \to \mathbf{2}\}|$, or $|T| < |\mathcal{P}(T)|$.

```
theorem "classic_cantor":
  fixes f :: "'a ⇒ 'a ⇒ bool"
  assumes surjectivity: "surj f"
  shows "False"
```

5 INSTANCES OF THE ARGUMENT

Now we have enough ingredients to reason about objects of interest by plugging them into our framework.

For starters, we dive into the world of different sizes of infinity. Then we realize the infamous Russell's paradox, which leverages self-reference. Finally, the abstracted framework entails a counting argument for languages and Turing machines.

5.1 $|\mathbb{N}| \lesssim |\mathcal{P}(\mathbb{N})|$

The most famous way to present that there are different sizes of infinity is to show that there are more elements in the power set of natural numbers than there are in natural numbers. Even though both sets are infinite, one is larger than the other. The result is reached by showing that there cannot be a surjective mapping from $\mathbb{N}$ to $\mathcal{P}(\mathbb{N})$. Take $\mathcal{P}(\mathbb{N})$ as the set of functions from $\mathbb{N}$ to `bool`.

Showing this result using our `classic_cantor` is as trivial as it gets.

```
theorem "classic_nat_cantor":
  fixes f :: "nat ⇒ nat ⇒ bool"
  assumes surjectivity: "surj f"
  shows "False"
```

What exactly is happening here can be nicely shown using a picture. Our function f acts like a grid of boolean values. The first natural number can be interpreted as the row and the second one as the column, wherein the intersection, the boolean value lies. Just filling in the first slot gives us an infinite sequence of binary digits, exactly a function from $\mathbb{N}$ to `bool`.

	0	**1**	**2**	**3**	**4**	**5**	**6**	⋯
0	1	0	1	0	1	0	1	⋯
1	0	1	0	1	0	1	0	⋯
2	0	0	0	0	0	0	0	⋯
3	1	1	1	0	1	0	0	⋯
4	0	0	1	1	1	1	0	⋯
5	1	0	0	1	0	0	1	⋯
6	0	1	0	0	0	1	0	⋯
⋮				⋮				⋱

The theorem tells us that this table should miss at least one infinite binary sequence. We construct the exact sequence as $g := \neg \circ f \circ \Delta$, pictorially g is the blue sequence below.

	0	**1**	**2**	**3**	**4**	**5**	**6**	⋯
0	1	0	1	0	1	0	1	⋯
1	0	1	0	1	0	1	0	⋯
2	0	0	0	0	0	0	0	⋯
3	1	1	1	0	1	0	0	⋯
4	0	0	1	1	1	1	0	⋯
5	1	0	0	1	0	0	1	⋯
6	0	1	0	0	0	1	0	⋯
⋮				⋮				⋱
	0	0	1	1	0	1	1	⋯

And we know for no $n \in \mathbb{N}$ that $f(n) = g$ as it always differs at the n^{th} spot.

5.2 Russell's Paradox

The second use of the framework is to show Russell's Paradox or, better said, how ZF axioms resolve it. But before we present in Isabelle, we give a quick reminder of Russell's Paradox.

Russell's Paradox happens when we unrestrictively use the set builder notation – which ZF axioms fix via the axiom of comprehension. Initially, we build a set of all sets that do not contain themselves, i.e. $R = \{x \mid x \notin x\}$. Then we ask whether R contains itself. If $R \in R$, the criterion tells us $R \notin R$; if $R \notin R$, we fulfil the criterion and $R \in R$. Paradox. [7]

ZF is formalized in Isabelle [5], and we can show how R cannot exist in this framework using `classic_cantor`. We achieve this by showing that the function `Elem :: ZF ⇒ ZF ⇒ bool (∈)` is not surjective.

```
lemma "russell's_paradox": "surj Elem ⟹ False"
apply (rule classic_cantor)
by simp
```

That looks much simpler than expected. A closer inspection is required to understand that this statement actually tells us R cannot exist in ZF. Let the function that `Elem` does not map to be g `:: ZF ⇒ bool`. Because of the unification with `classic_cantor` we also know that $g \coloneqq \neg \circ \texttt{Elem} \circ \Delta$. So for any set x, $g(x)$ is `True` if and only if $x \notin x$. Realize that this is exactly the characteristic function of R. So R's existence in ZF would imply `Elem` $R = g$. But as per the **lemma**, `Elem` does not map to g, meaning R does not exist in ZF.

5.3 Languages and Machines

Recursively enumerable (r.e.) languages are languages for which a Turing machine halts on and accepts each word in the language and halts on and rejects or runs forever for all other words.

In our formalization, we inspect words represented as natural numbers. Accepting a word means returning a tape that encodes the number 1 with blanks to the left and right, and rejecting a word means returning any other tape or not returning anything. We can write a function that checks if a Turing machine accepts a word or not as follows:

```
definition "tprog0_accepts_num p n ≡
  {⟦λtap. tap = ([], <n>)⟧}
    p
  {⟦λtap. ∃k l. tap = (Bk ↑ k, <1> @ Bk ↑ l)⟧}"
```

Realize the type; it is `tprog0 ⇒ nat ⇒ bool`. Where `nat ⇒ bool` exactly admits a language. True for words within it and false otherwise.

To show that there exist non-r.e. languages, we can use similar reasoning to the one we employed to show $|\mathbb{N}| < |\mathcal{P}(\mathbb{N})|$. In this context, the picture changes slightly, as shown below:

	0	**1**	**2**	**3**	**4**	**5**	**6**	⋯
$\mathbf{M_0}$	1	0	1	0	1	0	1	⋯
$\mathbf{M_1}$	0	1	0	1	0	1	0	⋯
$\mathbf{M_2}$	0	0	0	0	0	0	0	⋯
$\mathbf{M_3}$	1	1	1	0	1	0	0	⋯
$\mathbf{M_4}$	0	0	1	1	1	1	0	⋯
$\mathbf{M_5}$	1	0	0	1	0	0	1	⋯
$\mathbf{M_6}$	0	1	0	0	0	1	0	⋯
⋮				⋮				⋱

However, before we construct the blue sequence, we need to mention one assumption we are glancing over. That is, we can enumerate Turing machines as $\mathbf{M_0}$, $\mathbf{M_1}$, $\mathbf{M_2}$, ⋯. Such an enumeration would be possible to imagine only if Turing machines were countable. Thankfully, they are. We used Isabelle's HOL library `Countable` to show this within our formalization.

So, `tprog0` is countable, and there exist conversion functions `to_nat` and `from_nat`, which are inverses of each other. With this, we can realize the picture above via our framework.

```
theorem "non-re_languages":
  assumes surjectivity: "surj tprog0_accepts_num"
  shows "False"
apply (rule abstracted_cantor[of
  tprog0_accepts_num ¬ from_nat to_nat])
```

Here we use the most abstract version of the argument because the picture has different types of rows and columns. Only `abstracted_cantor` can capture `tprog0` $\times \mathbb{N}$. The argument constructs g as the blue sequence below,

	0	**1**	**2**	**3**	**4**	**5**	**6**	⋯
$\mathbf{M_0}$	1	0	1	0	1	0	1	⋯
$\mathbf{M_1}$	0	1	0	1	0	1	0	⋯
$\mathbf{M_2}$	0	0	0	0	0	0	0	⋯
$\mathbf{M_3}$	1	1	1	0	1	0	0	⋯
$\mathbf{M_4}$	0	0	1	1	1	1	0	⋯
$\mathbf{M_5}$	1	0	0	1	0	0	1	⋯
$\mathbf{M_6}$	0	1	0	0	0	1	0	⋯
⋮				⋮				⋱
	0	0	1	1	0	1	1	⋯

and g admits a language for which no Turing machine recursively enumerates. Hence, non-r.e. languages exist. Alternatively:
`∃l. ∀p. tprog0_accepts_num p ≠ l`.

6 HALTING PROBLEM

As we work with Turing machines and diagonalization, it is natural to be reminded of the Halting problem. Not coincidentally, parallels between some of the above results and the Halting problem are well known. [6]

So, as an exercise, we first try to formalize the Halting problem without any reference to the general framework. And in the later section, we inspect possible ways to fit the framework in our formalization, only to realize that that might not be possible.

Before we dive in, let's state the Halting problem. The problem asks if a machine decides whether a given Turing machine halts on an arbitrary input. We can define what halting would mean for an arbitrary machine and input as follows:

```
definition "halts p n ≡
                ∃Q. {⟦λtap. tap = ([], <n>)⟧} p {⟦Q⟧}"
```

So we encode a natural number `n` into a tape and run `p` on it. If **any** tape is given in the end, it would mean the machine halted. It should be noted that we are restricting ourselves to numerical inputs when deciding whether a machine halts. This is a smaller set than arbitrary inputs, and if we cannot devise a machine deciding the Halting problem here, the arbitrary case immediately follows.

Using `halts`, we can define the behaviour of a machine that decides the Halting problem as follows:

```
definition "decides_halting H ≡ ∀p n.
  {⟦λtap. tap = ([Bk], <tm_to_nat p, n>)⟧}
    H
  {⟦λtap. ∃k l. tap = (Bk ↑ k, <(if halts p n then 0
                                  else 1)> @ Bk ↑ l)⟧}"
```

Basically, H is given a tape that encodes both the natural number encoding an arbitrary machine and a numerical input. Afterwards, it always halts with a tape that encodes the number 0 if the arbitrary machine halts on the numerical input and the number 1 otherwise.

The Halting problem turns out to be undecidable, i.e. no machine fulfils decides_halting; so we would like to show that "∄H. decides_halting H". To achieve this, we must talk about two other machines which do exist. The machines are named dither and copy. Their workings are quite simple and are explained below.

dither's behaviour is defined only for two inputs: on a tape that encodes the number 1, dither halts and returns the same exact tape; on a tape that encodes the number 0, dither runs forever.

copy's behaviour is known for any tape that encodes a number: on a tape that encodes the number n, it returns a tape that encodes <(n, n)>.

It is more convenient to talk about the behaviours of these machines in terms of their Hoare triples. We can state them easily as follows:

```
{⦇λtap. ∃k l. tap = (Bk ↑ k, <1> @ Bk ↑ l)⦈} dither
{⦇λtap. ∃k l. tap = (Bk ↑ k, <1> @ Bk ↑ l)⦈}

{⦇λtap. ∃k l. tap = (Bk ↑ k, <0> @ Bk ↑ l)⦈} dither
↑

{⦇λtap. tap = ([], <n>)⦈} copy
{⦇λtap. tap = ([Bk], <(n, n)>)⦈}
```

Additionally, remember that these machines can be made composable, and they would still preserve their Hoare triples. Now we have all the tools necessary to show the result.

Proof. For contradiction, assume that there exists a machine that decides Halting. Let's name this machine H, so we have: "decides_halting H".

Using this machine together with dither and copy, we can define a new machine as a composition. Let this machine be named contra defined as contra ≔ copy |+| H |+| dither. To save ink, we present the Gödel encoding of contra not as "tm_to_nat contra" but as "⌜contra⌝".

We now reason about the behaviour of contra on the numerical input of ⌜contra⌝. As with any machine, there are two cases: either it halts or runs forever. We use our halts definition to show these cases.

Case 1: Assume halts contra ⌜contra⌝

We can inspect what happens to contra when run on ⌜contra⌝ by looking at the behaviour of the machines that make it up. Firstly copy satisfies the following Hoare triple for a tape encoding ⌜contra⌝:

```
{⦇λtap. tap = ([], <⌜contra⌝>)⦈}
  copy
{⦇λtap. tap = ([Bk], <⌜contra⌝, ⌜contra⌝>)⦈}
```

H acts on this output in the following way:

```
{⦇λtap. tap = ([Bk], <⌜contra⌝, ⌜contra⌝>)⦈}
  H
{⦇λtap. ∃k l. tap = (Bk ↑ k, <0> @ Bk ↑ l)⦈}
```

The output tape encodes the number 0 by definition of H because we assumed contra halts on ⌜contra⌝. And as the final step, dither satisfies the following Hoare triple following its definition:

```
{⦇λtap. ∃k l. tap = (Bk ↑ k, <0> @ Bk ↑ l)⦈}
  dither
↑
```

We can chain these three Hoare triples from where their output matches the input and compose the machines accordingly. Which gives:

```
{⦇λtap. tap = ([], <⌜contra⌝>)⦈}
  copy |+| H |+| dither (=: contra)
↑
```

So, we let contra run on ⌜contra⌝ by tracing the behaviour of the machines that compose it and came to the conclusion that it does **not** halt – contradicting our assumption.

Case 2: Assume ¬ halts contra ⌜contra⌝

This case's proof is symmetrical to the above case's, so it is left out. □

In Isabelle, the formalized proof is given with the following theorem:

```
theorem "halting_problem": "∄H. decides_halting H"
```

7 SHORTCOMINGS AND LOCALES

As said at the beginning of the previous section, the Halting problem has many parallels with results that leverage diagonalization. This is not so hard to see now. The counterexample we came up with that shows the limits of the powers of the object we were after is constructed via diagonalization. We wanted to see if H exists; it didn't because of contra.

Remember how we defined g in our general framework to show the limited power of f. As $g \coloneqq \alpha \circ f \circ \Delta$. And here we defined contra as contra ≔ copy |+| H |+| dither. These two identities are completely analogous. We first diagonalize, then pass the result to the object of interest, and finally pass it to a construction with no fixed points.

Unfortunately, even though there are strong analogies between our abstraction and the Halting problem, they cannot fit together.

7.1 Failure of generalization

To understand why the generalization fails in the context of the Halting problem, we have to understand how much we limit the power of f and H. To derive the non-existence of f, we assume its surjectivity, and in the pictorial representation, that coincides with an entire infinite sequence which f cannot map to – like the blue sequences we inspected.

But in the case of the Halting problem, the pictorial representation, which can be seen below, failed not in an entire row but in just

an entry. Unlike the previous generalizations, where we picked an entire row, we had to pick both the row and the column to spot an undecidable entry in the matrix.

	0	**1**	**2**	**3**	**4**	⋯	**c**	⋯
$\mathbf{M_0}$	1	0	1	0	1	⋯	1	⋯
$\mathbf{M_1}$	0	1	0	1	0	⋯	0	⋯
$\mathbf{M_2}$	0	0	0	0	0	⋯	0	⋯
$\mathbf{M_3}$	1	1	1	0	1	⋯	0	⋯
$\mathbf{M_4}$	0	0	1	1	1	⋯	0	⋯
$\mathbf{M_5}$	1	0	0	1	0	⋯	1	⋯
$\mathbf{M_6}$	0	1	0	0	0	⋯	0	⋯
⋮			⋮			⋱	⋮	
$\mathbf{M_c}$	0	0	1	1	0	⋯	?	⋯
⋮			⋮				⋮	⋱

We need more structure than naked sets to capture this kind of proof using diagonalization.

7.2 A more structured abstraction

For a better generalization, we use a carrier set. It is basically a subset of a naked set which admits some additional structure. We call it `F`.

It is important to realize that with many instantiations of the argument, we were using countable objects. Consider the Halting problem; numeral inputs and Turing machines are both countable – so we can encode these types with `nat`. Additionally, we had numeral outputs, **sometimes**. So this set's members we wish to construct should get a numeral input and optionally return a numeral output. The type in Isabelle that corresponds to this is `(nat⇀nat) set` – the partial arrow indicates that the output is a `nat option`. (`option` monad represents results that might go wrong. It becomes `Some n` if the computation succeeded and returned n. Otherwise, it becomes `None`.)

The elements of this set are sometimes interpreted as different objects. In the general framework, we had β and $\overline{\beta}$ to encode/decode our objects; in the Halting problem, we had Gödel encodings. To capture this transformation, we equip our set with an operation `pull_up` that is of type `(nat⇀nat)⇒nat`. Stylistically we sometimes show `pull_up` as `⌜·⌝`.

Our encodings were always injective, as we always needed to decode back to the original object in our proof without confusion. So we have to assume that our encoding is injective, at least on the elements of our carrier set. In Isabelle, that is: `inj_on pull_up F`.

Once we have an injective `pull_up`, constructing a decoder `push_down` becomes trivial. It would simply be equivalent to the inverse of `pull_up` into the elements of `F`. In Isabelle, that is shown by `inv_into F pull_up`. But at times, we might desire to push down the outputs of our functions, and in some cases, the output might be `None`. Also, even if it is not `None`, not all `nat`'s have a function that encodes to it. We assumed injectivity, not bijectivity or surjectivity; some numbers might never be encoded into. So we make a simple case distinction and a small check to define `push_down` as follows:

```
definition "naive_push_down ≡ inv_into F pull_up"
definition "push_down x = (case x of
  Some n ⇒ (if ∃f. ⌜f⌝ = n then naive_push_down n
                           else (λ_.None)) |
  None ⇒ (λ_.None))"
```

`(λ_.None)` stands for the bottom function, used when things don't make sense, like the `push_down` of a number for which no function encodes into. Again, stylistically we sometimes use `⌞·⌟` to represent `push_down`. And by definition, it follows that
`∀f ∈ F. ⌞Some ⌜f⌝⌟ = f`.

The last additional structure we need is some sort of chaining. Analogous to function composition in our general framework and machine composition in the Halting problem. We can quickly realize that we need a monadic composition. So if the output of a function is `Some n`, we pass `n` along the composition, but if it is `None`, we just return `None`. Point-wise, we define the composition as follows:

```
(f₁ ⊕ f₂) x = (case f₂ x of None ⇒ None |
                            Some n ⇒ f₁ n)
```

Of course, our carrier set should be closed under $\oplus$. To formalize this carrier set with the specified structure, we define the following locale in Isabelle:

```
locale computable_universe_carrier =
  fixes F :: "(nat⇀nat) set"
  fixes pull_up :: "(nat⇀nat) ⇒ nat"
  assumes countable: "inj_on pull_up F"
  assumes comp_closed: "⟦a ∈ F; b ∈ F⟧ ⟹ a ⊕ b ∈ F"
```

Now that we have a carrier set, we can specify the functions we need inside of it to invoke the diagonalization argument. Firstly, the analogue of `dither`, which is trivial. We name it α. It basically maps `1` to `None`, and `0` to `Some 1`.

However, when trying to create the analogue of `copy`, we realize a caveat. We cannot output two numbers. That is against the type of our carrier set. Also, we need a function that takes in two inputs; hence the table-like pictorial view, but our carrier set does not support that immediately. So, we devised two approaches to circumvent this issue, enabling us to distill the diagonalization argument. The first one makes use of a more general currying-inspired strategy; the second one simply encodes pairs into natural numbers.

Curry-like approach for Δ

We call the analogue of `copy`, Δ. We are only interested in Δ when it is composed to the left with some other function. This function should support taking two inputs via currying. It is fed the first input and gives out a number that can be interpreted as another function which in turn can eat up the second input. We know how to interpret an output as a function; it is `push_down`. So Δ acts as follows:

`⋀f. f ∈ F ⟹ (f ⊕ Δ) x = ⌞f x⌟ x`

So functions that take in two inputs have to be curried, and the output of the partially applied part needs to encode the rest of the function as a natural number.

Now we can define H, which is the analogue of H in the Halting problem. H partially applied should stay within the carrier set so we can push it down later. Formally:

```
⋀f h. ⟦f ∈ F; H ⌜f⌝ = Some ⌜h⌝⟧ ⟹ h ∈ F
```

The function that is encoded in the output should act as follows – completely analogous to the Halting machine:

```
∀f ∈ F. H ⌜f⌝ = Some ⌜H_f⌝, where
H_f c = (case f c of Some _ ⇒ Some 1 | None ⇒ Some 0)
```

With these, we can show that `H ∉ F`.

Proof. Assume for contradiction that `H ∈ F`. Then we can construct `contra := α ⊕ H ⊕ Δ`. We then inspect `contra ⌜contra⌝`.

```
contra ⌜contra⌝ = α ((H ⊕ Δ) ⌜contra⌝)
                = α (⌞H ⌜contra⌝⌟ ⌜contra⌝)
                = α (⌞Some ⌜H_contra⌝⌟ ⌜contra⌝)
                = α (H_contra ⌜contra⌝)                (*)
```

`H_contra ⌜contra⌝` by definition only has two results. Either `Some 1` or `Some 0`. So we make a case distinction.

Case 1: Assume `H_contra ⌜contra⌝ = Some 1`

From the case assumption, and following the definition of H_{contra}, we conclude that `contra ⌜contra⌝ = Some _`.

And from `(*)` it follows that `α (H_contra ⌜contra⌝) = Some _`.

But substituting the case assumption, we get `α (H_contra ⌜contra⌝) = α 1 = None`, contradicting the above line.

Case 2: Assume `H_contra ⌜contra⌝ = Some 0`

This case's proof is symmetrical to the above case's, so it is left out. □

Pairing approach for Δ

In this approach, we define Δ using another small function `pair_to_nat` that encodes a pair of natural numbers into a natural number. Again just like the previous approach, we define Δ only in the context of a composition with another function to the left. It behaves as such:

```
⋀f. f ∈ F ⟹ (f ⊕ Δ) x = f (pair_to_nat (x, x))
```

The H with this approach also changes a bit, but it is still quite reminiscent. It is given as follows:

```
⋀f c. f ∈ F ⟹ H (pair_to_nat (⌜f⌝, c)) =
      (case f c of Some _ ⇒ Some 1 | None ⇒ Some 0)
```

Again, we prove that `H ∉ F`.

Proof. Assume for contradiction that `H ∈ F`. Then we can construct `contra := α ⊕ H ⊕ Δ`. And inspect `H (pair_to_nat (⌜contra⌝, ⌜contra⌝))`.

By H's definition, there are only two possible results: `Some 1` and `Some 0`. So we make a case distinction.

Case 1: Assume `H (pair_to_nat (⌜contra⌝, ⌜contra⌝)) = Some 1`

From the case assumption, and following the definition of H, we derive that `contra ⌜contra⌝ = Some _`. (*)

Expanding `contra ⌜contra⌝` gives us:

```
contra ⌜contra⌝ = α ((H ⊕ Δ) ⌜contra⌝)
                = α (H (pair_to_nat (⌜contra⌝,
                                     ⌜contra⌝)))
                = α 1 = None
```

But that contradicts `(*)`.

Case 2: Assume `H (pair_to_nat (⌜contra⌝, ⌜contra⌝)) = Some 0`

This case's proof is symmetrical to the above case's, so it is left out. □

7.3 The Halting problem revisited

We can get back to the Halting problem by instantiating this newly created parametric theory. We use the pairing approach variant for its simplicity.

To build a carrier set, we need special Turing machines to induce functions. We make it so that our Turing machines are composable as it will be easier to work with. Additionally, when given input tapes that encode numeric values, we want the output tapes to encode numeric values as well – in case the machines halt.

Elements of this specific set of Turing machines satisfy the function `numeral_composable_tm0`.

Then the functions we induce from these machines work as expected. We encode the given natural number onto a tape which we then run on the machine that induces the function. If the machine halts, it outputs a tape that encodes a number, which we return. If it does not halt, we return None. This feels like too strict of criteria on the machines and input/output formats, and it is, but it is just enough to realize the Halting problem.

We refrain from showing the Isabelle formalization of this set because of the complicated definition. The reader is referred to the `.thy` file to inspect the two below definitions with the paragraphs above as a guide.

```
definition induce_F_from_tprog0 ::
                          "tprog0 ⇒ nat ⇒ nat option"
definition turing_F :: "(nat⇀nat) set"
```

After inducing Turing machines into our carrier set, we need a `pull_up` back to natural numbers. There is a trivial way of doing this. We know a Turing machine gives us the function inside the carrier set, so we use Hilbert's choice to obtain this machine and get its Gödel encoding.

```
definition turing_pull_up :: "(nat⇀nat) ⇒ nat" where
"turing_pull_up f = (if f ∈ turing_F
  then tm_to_nat (SOME p. induce_F_from_tprog0 p = f)
  else 0)"
```

We can prove that turing_pull_up is injective on our carrier set using Isabelle's SMT solver.

Now that we have our functions, we need to show that they are closed under ⊕. We restricted the machines that induce these functions to be composable. So we can leverage this fact and show that there is a correspondence between ⊕ and |+|. Because we know composable Turing machines are closed under |+|, we can inherit this to show our carrier set functions are closed under ⊕. Formally, we are after the following lemma:

```
lemma seq_tm_oplus_correspondence: "⋀p1 p2.
  ⟦numeral_composable_tm0 p1;
    numeral_composable_tm0 p2⟧ ⟹
  induce_F_from_tprog0 (p2 |+| p1) =
  induce_F_from_tprog0 p1 ⊕ induce_F_from_tprog0 p2"
```

The way these functions are induced makes it clear that this lemma is correct. The numeral outputs are passed along by machine composition the same way they are passed along by function composition. The restriction on numeral outputs plays a crucial role here; if that was not the case, machine composition could have passed pairs along, which ⊕ cannot, and the correspondence would fail. However, by our restriction, we manage to circumvent that issue. Lastly, a couple of technical steps are required to show this in Isabelle, which is left incomplete in the formalization.

All of the above definitions and lemmas conform to the required structure we specified in the **locale** computable_universe_carrier. The only thing left to show the undecidability of the Halting problem is to devise an α and Δ. We can manage this using dither and a modified copy.

dither already takes in numeral inputs and only gives numeral outputs or just runs forever, so it is easier to instantiate into the framework.

But fitting copy into the behaviour of Δ is not immediately clear, as the output tape of copy encodes not a single number but a pair. The way we work around this is to change copy to simply double the numeral input. If it takes in a tape encoding x, it outputs a tape encoding $x + x$.

So the pair_to_nat function we have defined becomes an addition operation to mimic the modified copy's behaviour.

A modified_copy machine has been defined as the composition of copy with a machine that takes in a pair and adds them. The proof for a Hoare triple showing its behaviour is left incomplete. But assuming the behaviour holds, the induced function is just what we need to instantiate Δ – with pair_to_nat being addition.

Using this guideline, we can induce the necessary functions to instantiate the rest of the locale that uses the pairing approach. Some parts of this instantiation are yet to be completed, but a clear strategy for completing it is noted down. When complete, the non-existence of a Halting deciding machine within turing_F falls automatically as the proof of H ∉ F within the locale is analogous to the Halting problem.

8 DISCUSSION

In this research, we started off by formalizing a set-theoretic conception of the diagonalization argument owed to Yanofsky. [10] We realized that there could be certain restrictions on the free variables of the abstracted framework to derive a toolbox. Afterwards, we employed our toolbox by instantiating the framework with different mathematical objects to reach important mathematical results.

Looking back at the Contributions. We showed Cantor's theorem and proved that different sizes of infinities exist. We showed a built-in limit of the Elem (∈) relation within the ZF set theory, enabling modern set theory to circumvent the infamous Russell's paradox. We showed the existence of non-recursively enumerable languages by invoking the framework with Turing machines and words within languages as natural numbers.

After getting comfortable with Turing machines, we wished to show the undecidability of the Halting problem – as there are known parallels between the diagonalization argument and that proof. [6] We were successful in formalizing this important result but failed to use our framework within. We quickly realized the toolbox could not fit into the context of the Halting problem as it is too coarse and limiting. This realization then opened up a possibility for a more nuanced abstraction using the parametric theories of Isabelle – **locale**s.

We created a carrier set with a structure just enough to realize an analogous result to the undecidability of the Halting problem without any reference to the mathematical context of Turing machines. Then, using the Turing machine formalisation, we built a promising but incomplete scaffolding to instantiate the parametric theory.

All of these interconnected, deeply insightful mathematical results were formalized in Isabelle. Additionally, a section on cardinalities of infinities was formalized but had to be dropped from the paper due to page-count limitations. The Isabelle code for all the results mentioned in the paper and more can be found in the repository given under Contributions.

9 CONCLUSION

In conclusion, Yanofsky's various set-theoretic generalizations [10] have been formalized in Isabelle. These encompass the diagonalization argument, Cantor's theorem [1], Russell's paradox [7], and the existence of non-recursively enumerable languages. Also included are the Halting problem, along with two novel diagonalization argument abstractions that utilize minimally structured carrier sets.

Apprehension of the deep connection these results share with each other has been gained. Some of the differences have also been spotted and studied. Future work on formalizing a category-theoretical abstraction is left open. Building on this research, especially on the discussed parametric theories, this extension is quite possible and would most likely prove even more powerful. Instantiating Gödel's incompleteness theorems can be seen as a goal mark, which was left untouched by this research's framework.

REFERENCES

[1] Cantor. 1874. Ueber eine Eigenschaft des Inbegriffs aller reellen algebraischen Zahlen. 1874, 77 (Jan. 1874), 258–262. https://doi.org/10.1515/crll.1874.77.258 Publisher: De Gruyter Section: Journal für die reine und angewandte Mathematik.

[2] H. Geuvers. 2009. Proof assistants: History, ideas and future. *Sadhana* 34, 1 (Feb. 2009), 3–25. https://doi.org/10.1007/s12046-009-0001-5

[3] John Harrison. 2008. Formal Proof—Theory and Practice. *Notices of the American Mathematical Society* 55 (Jan. 2008).

[4] F. William Lawvere. 1969. Diagonal arguments and cartesian closed categories. In *Category Theory, Homology Theory and their Applications II (Lecture Notes in Mathematics)*, Barry Mitchell, Jan-Erik Roos, Friedrich Ulmer, Hans-Berndt Brinkmann, Stephen U. Chase, Paul Dedecker, R. R. Douglas, P. J. Hilton, F. Sigrist, Charles Ehresmann, K. W. Gruenberg, Max A. Knus, F. William Lawvere, and Saunders Mac Lane (Eds.). Springer, Berlin, Heidelberg, 134–145. https://doi.org/10.1007/BFb0080769

[5] Lawrence C. Paulson. 2019. Zermelo Fraenkel Set Theory in Higher-Order Logic. *Archive of Formal Proofs* (Oct. 2019).

[6] Mikhail Prokopenko, Michael Harré, Joseph Lizier, Fabio Boschetti, Pavlos Peppas, and Stuart Kauffman. 2019. Self-referential basis of undecidable dynamics: From the Liar paradox and the halting problem to the edge of chaos. *Physics of Life Reviews* 31 (Dec. 2019), 134–156. https://doi.org/10.1016/j.plrev.2018.12.003

[7] Bertrand Russell. 2010. *Principles of mathematics.* Routledge, London.

[8] Makarius Wenzel. 2007. Isabelle/Isar — a Generic Framework for Human-Readable Proof Documents. https://www.semanticscholar.org/paper/Isabelle-Isar-%E2%80%94-a-Generic-Framework-for-Proof-Wenzel/1bd3c0543ebb6caf38676f2dfeb16451d20a569e

[9] Jian Xu, Xingyuan Zhang, and Christian Urban. 2013. Mechanising Turing Machines and Computability Theory in Isabelle/HOL. In *Interactive Theorem Proving (Lecture Notes in Computer Science)*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer, Berlin, Heidelberg, 147–162. https://doi.org/10.1007/978-3-642-39634-2_13

[10] Noson S. Yanofsky. 2003. A Universal Approach to Self-Referential Paradoxes, Incompleteness and Fixed Points. *The Bulletin of Symbolic Logic* 9, 3 (2003), 362–386. https://www.jstor.org/stable/3109884