how to resolve a halting paradox

Nicholas Swenson
dart200@gmail.com

In 1936 Alan Turing published the groundwork math paradigms we still use today as our foundations for computing. He spent the first half of this paper describing the model we now call Turing machines, but the second half was dedicated to proofs attempting to establish inherent incompleteness in computing as a theory: including the halting problem. Since then the halting problem has stood as a relatively unquestioned fundamental limit to computing. The paradoxes encountered when hypothetically applying halting deciders in self-referential analysis are interpreted to be some kind of ultimate algorithmic limit to reality. This paper proposes alternatives to the accepted consensus on the matter, and attempts to demonstrate two methods in which we might circumvent those paradoxes through refining the interfaces we use in halting computation, *in order to make the programmatic forms of those paradoxes decidable*.

Both methods hinge on utilizing multiple *decision machines*, or *deciders*, in distinct ways, in order to mitigate attempts at creating self-defeating logic. This paper is focused on just resolving the paradoxes involved in halting analysis under self-reference, and to be clear: it is not then presenting a general halting algorithm. This paper does not attempt to present at depth arguments or reasons for why we should accept either of these proposals vs a more conventional perspective, it is mostly an objective description of the conceptions for further musing upon. Lastly, we will stick to solely the basic halting paradoxes found within computing. We will not try to address or apply these techniques to other problems of logical undecidability, either within computing, or greater math such as Gödel's Incompleteness.

Pseudocode notes:
Function definition is derived from lambda syntax used is modern languages like kotlin/typescript:
`function_name = (params) -> { code }`, curly braces optional
The typical ternary operator functions as expected:
`bool ? true_case : false_case`

1: the halting paradoxes

The halting decider is traditionally specified as computing a true/false function:

```
halts = (m: machine) -> {
  true: if m halts,
  false: otherwise,
}
```

The basic **undecidable paradox** can be expressed in a single line of logic:

```
und = () -> halts(und) ? und() : return
```

If one substitutes `true` for `halts(und)`, the program will loop forever recursively. If one substitutes `false` for `halts(und)`, the program will terminate immediately. Clearly it is not possible to coherently define a halting decision for `halts(und)`, and have it remain consistent in a seemingly unsolvable paradox. The halting decider is overspecified and bars anything (including us) from forming a coherent decision on the matter.

There is a second form of paradox, a **nondeterministic paradox**, that is rarely discussed:

```
ndt = () -> halts(ndt) ? return : ndt()
```

The flipped turnery cases, flips the behavior of `ndt` vs und. `ndt` will halt when `halts(ndt)` returns `true`, and loop forever recursively if `halts(ndt)` returns `false`. Either halting decision is valid ... so which is it supposed to be!? A decision machine is a computing machine, and a computing machine is a specific method to derive a computable function's singular output, from its input. Having two possible outputs is a contradiction to what the decider should be. While being overspecified in the previous example, the halting decider as declared is simultaneously demonstrating underspecification during `ndt`, leading to a situation with multiple valid decisions.

Now, the nondeterministic paradox is trivially resolvable, and can be done so with an algorithmic bias on the output, so this paper primarily focuses on resolving the undecidable paradox. However, the fact that two different paradoxes arise under self-analysis, out of the four possible scenarios of a simple if/else, suggests something might be quite wrong with how we define the halting deciders.

2: infinite adjacent deciders

The first proposed method of resolving the halting paradox involves almost identical deciders, differing only in name:

```
h1(m) -> true/false
h2(m) -> true/false
```

Both take on the original halting specification, returning `true` if their input `m` halts, and `false` otherwise. If they are caught in an undecidable situation where deciding a value is impossible, they will not return and instead just loop forever. Such behavior will be referred to as an “`undefined`” value for discussion's sake, but this value will never be returned. Only `true`/`false` are valid return values from either decider.

Let’s consider the undecidable paradoxes for these deciders:

```
und1 = () -> h1(und1) ? und1() : return
und2 = () -> h2(und2) ? und2() : return
```

In both cases, the associated decider has an `undefined` value and will loop forever when executed:

```
h1(und1) -> undefined
h2(und2) -> undefined
```

However, because neither paradox involves the adjacent decider, they can cover for the other's missing value:

```
h1(und2) -> false
h2(und1) -> false
```

We've found a method to deal with single decider paradoxes: just ask the other decider. But now we have two deciders, so let's consider trying to disprove both of them with a single paradox:

```
und3 = () -> ( h1(und3) && h2(und3) ) ? und3() : return
```

We can establish a truth table for the possible results:

h1 value	h2 value	und3 execution
true	true	loops
true	false	halts
true	undefined	loops
false	true	halts
false	false	halts
false	undefined	halts
undefined	true	loops
undefined	false	loops
undefined	undefined	loops

There is only one case where: (a) no decider returns an invalid value, (b) at least one decider produces a valid result, and that is when `h1(und3)->undefined` and `h2(und3)->false`. When `und3` is executed h1 will fail in nondecision, looping forever. This allows h2 to escape the paradox because it is never actually evaluated during the execution of `und3`, such that it can take the value of `false` to accurately describe `und3`'s runtime.

The ability for one decider to stall execution if any form of halting paradox (whether undecidable or nondeterministic) be detected, allows the second decider to then take on a `false` value and accurately decide/predict the stalled runtime, without being held liable to answer to the constructed paradox. A combination use of these deciders should be able to describe the halting behavior for any function without being subject to a disproving paradox. However, one cannot reduce the combination to a single value, such a single value does not have the same descriptive power as the separated evaluations, and could be subject to undecidable inputs.

It is, however, quite unfortunate to have a decision machine that stalls on undefined values. The values are usable in other machines, but guaranteeing the use of the value can only be done by running simultaneous machines each using one of the adjacent deciders. If one wanted to use

the halting behavior of und3, a machine would need to be run using h1(und3), and another the other using h2(und3), and both machines would need to be evaluated separately in order to ensure a result without stalling on the undefined h1(und3) value.

Furthermore, such an approach may have an achilles heel that can still lead to unresolvable contexts. One may try to evaluate these deciders concurrently on the same machine, in for example separate threads/coroutines, and return whichever value resolves first. Doing so will subvert the stalling ability of these deciders that allows for at least one of them to decide on paradoxical contexts, and could lead to another undecidable situation:

```
und4 = () -> {
  p1 = Promise(() -> h1(und4))
  p2 = Promise(() -> h2(und4))
  // Promise.race() resolves to whichever promise resolves first
  decision = await Promise.race([p1, p2])
  decision ? und4() : return
}
```

But then one can consider the fact that our adjacent deciders are identical except in name, and because of this it is certainly quite possible to define and generate an unbounded amount of them: while neither h1 nor h2 can give a defined value for und4, an h3 could since it is not named in und4. More generally: *Any malicious machine undn that tries paradox n deciders h1...hn through the use of concurrent evaluation, there is still an decider hn+1 that could define a halting value for that machine*.

However, concurrent evaluation may extend the paradox even to infinite adjacent deciders. This next example endlessly initiates and tests an infinite amount of deciders, and can be extended to any set of deciders that is countable. If any generated decider ever actually resolves to a `true` or `false` decision, then it will be paradoxed at some point during the execution:

```
und∀ = () -> {
  promises = []
  for (i = 0; true; i++) {
    for (p in promises) {        // test each promise every loop
      if (p.is_resolved()) {
        decision = await p
        decision ? und∀() : return
      }
    }
    hi = Halts(i)                // init new concurrent analysis
    pi = Promise(() -> hi(und∀))
    promises.push(pi)
  }
}
```

This may seem to be a definite end for the adjacent deciders proposal, but there is one last idea worth considering: *make the set of adjacent deciders uncountably infinite by declaring the existence of an adjacent decider for every real number.* 🤯 As demonstrated by Cantor's diagonal, such a set would not be countable, and could not be stepped thru by any method/computation, and therefore no single machine like und∀ could paradox all of them in one execution. Leaving even one decider out would make the machine decidable by at least one decider, and such is the goal of the opposing decider technique. This is perhaps why we can understand and discuss how und∀ will operate, even if none of the countable deciders can actually decide upon it. But it may prove hard to actually compute via these uncountable deciders, as merely referencing to one by some id would put that decider in the set of countable deciders, and therefore unable to deal with und∀.

Alas, the next section suggests further refinement of the specification for the halting deciders, such that neither undefined values, stalling on those undefined values, nor the resulting game of infinite cats vs an uncountable hoard of mice... is necessary to decide a halting evaluation for any given machine.

3: binary opposing deciders

```
halts = (m: function) -> {
  true: iff (m does halt && m remains halting after decision),
  false: otherwise,
}

loops = (m: function) -> {
  true: iff (m does not halt && m remains looping after decision),
  false: otherwise,
}
```

The second proposal establishes binary deciders in opposed decision rather than adjacent: `halts` and `loops`. These specifications differ substantially in nuance from the original halting function. For the most part, they will operate like one expects, but there is an additional qualification that the `true` value must remain truthful during the remainder of its **decision context**, which is the scope of code that the decider is asked to decide upon. Such is the point of the added predicates: `&& m remains halting/looping after decision`. What this means is the decider will only return `true` iff such decision will stay truthful after the decider returns the decision.

Conversely, `false` values do not make such a guarantee, and can be returned even if the decider would return `true` when executed outside its decision context. This brings about a **context sensitive** nature to these deciders. The context sensitive nature only matters when the decider is executing within its own decision context**,** and can essentially be ignored otherwise. But when executing within such decision context, having an escape is integral to building a decidable runtime under such self-analysis. The reason for having opposing deciders is to have an interface that grants the best possible guarantee for returning truth of the prediction one is most interested in, as each decider can only guarantee certain truth for the `true` decision of

that decider. As a final note, to prevent nondeterminism from arising, these deciders do operate with opposing decision biases, each preferring to return `true` whenever possible.

Let's consider how this would work in a basic undecidable paradox:

```
0 und = () -> halts(und) ? loop_forever() : return
1 main = () -> halts(und)
```

Given the context-dependent nature of the return values, we need line numbers to refer to the **call context** of the returned decision. If `und` is run, `halts@L0(und)` will determine that it cannot return `true` without it then being immediately contradicted by the following `loop_forever()`, so it will return `false` causing the program to halt immediately. When `halts` is run externally to `und` like `halts@L1`, the halting decider is free to return `true`, which is the objective runtime nature of `und` execution.

While this might seem a bit arbitrary at first, the complex interplay between various deciders operating self-referentially is both precise, and decidable. Let us consider a more complex paradox:

```
 0 paradox = () -> {
 1  if ( halts(paradox) || loops(paradox) ) {
 2     if ( halts(paradox) )
 3       loop_forever()
 4     elif ( loops(paradox) )
 5       return
 6     else
 7       loop_forever()
 8   }
 9 }
10 main = () -> {
11   halts(paradox)
12   loops(paradox)
13 }
```

If one tries to utilize the naive decider specification which ignores context sensitivity, `paradox` will surely be computably undecidable (the reader is left to work such out). Let us go through and detail how this can be decidable with the more refined context-dependent specifications, considering the decider calls in the necessary order to decide this runtime:

➢ `loops@L4(paradox) -> false`

> The first decider call we shall consider, since the execution after involves no further decider calls. If `true` is returned, L5 will run next causing termination and contradicting the `true` value, so `false` will be returned, causing L7 to run an infinite loop.

➢ `halts@L2(paradox) -> false`

> If `true` is returned, L3 will run next causing an infinite loop, and contradicting the `true` value, so `false` will be returned, leading to the execution of L4, L7, and ultimately an infinite loop.

➢ `loops@L1(paradox) -> true`

> In this case returning `true` will cause L2 to be executed next, and ultimately the resulting infinite loop on L7, so should execution reach this point, `true` will be returned.

➢ `halts@L1(paradox) -> false`

> Like the `loops@L1` on this line, returning `true` will cause an infinite loop on L7, so it will return `false`. This, of course, does not stop the infinite loop from happening, as the subsequent `loops@L1` call will return `true`.

At this point we can determine that the chain of decider calls within `paradox` will cause it to loop into eternity. What was an undecidable mess becomes rectifiable, *without deciders that potentially stall indefinitely*. The last two calls from `main` can now be decided upon. Neither of these are executing within their decision context, so they both just give objective overall runtime behavior without further issue.

➢ `halts@L11(paradox) -> false`
➢ `loops@L12(paradox) -> true`

A major concern many will have with this approach is the fact these deciders give different responses in different contexts, which can be seen as computing functions that aren't *well-defined*. How can a decision machine return different values for the same input, and still be called a function, let alone a Turing machine? To this it can be suggested there is actually a hidden parameter to the function being computed: the *call context*.

This call context is not user-controllable by parameter, but is implicitly provided when the decider is called. The technical details on how to provide the context is outside the scope of this paper, and would depend on specific implementation. But for example: if one is imagining Turing machines, then the point at which one machine calls into (or more technically starts simulating) the decider, it puts the rest of the machine into a static state, that the executing decider would then need full access to for examination: including both the state-machine and tape state.

Again, one may still be wondering how this doesn't violate our intuitive notion that either a machine halts or it doesn't. But that intuition hasn't been violated, for every possible input machine, there is a `null` context value which does indeed provide the objective machine runtime. If `halts@null(paradox)` were to be run (meaning directly with no callee context that will be resumed after), `false` will always be returned. This `null` context value will be returned in all decider executions outside the decider's decision context. The naive halting function (simple `true`/`false`) is still found within this extended, context based halting map, but

we've constrained how it is accessed such that its computability doesn't just disappear in a puff of self-defeating logic.

> actual halting function: `(machine, context) -> true/false`
> naive halting function: `(machine, null) -> true/false`

This doesn't make decider executions within their decision context useless, they still act as completely functional branch guards. The execution branches gated by these deciders will not be run unless the associated branch execution remains truthfully in accordance with a `true` decision. In fact, the decider will only return `true` if the objective truth is also `true`. Let us construct a truth table for how these deciders return under various conditions vs the objective truth on the matter:

objective truth	coherent call context	paradoxical call context
`true`	`true`	`false`
`false`	`false`	`false`

Coherent contexts include any call context external to the decision context, as well as any within where a truthful return isn't contradicted. Paradoxical contexts are those tricky call locations within the decision context, where an objectively truthful return would be ultimately contradicted after the decision is returned.

4: why tho?

As stated in the abstract, this paper will not go into depth for why we might want to care about possible resolution strategies to the halting paradox vs conventional thought, and is focused entirely on just disseminating potential resolution strategies.

But I like to consider that should a resolution to halting paradoxes be found and agreed upon, it can likely be generalized to all the non-trivial properties of a computing machine, claimed to be undecidable by Rice's Theorem. If so this could mean that the behavior of computing machines may prove to be generally computable.

This of course wouldn't make the behavior trivially decidable. Generally deciding would still be at least of np-complete complexity, if for example the decision between halting and looping was based on running an np-complete decision problem (like traveling salesman). There are likely harder cases that can be constructed, and quite probably a general halting algorithm may be as hard as the hardest class of computable problems.

But we do not need to let perfection get in the way of doing a lot better. *I propose that we should be able to find reasonable algorithms that can generally prove the halting behavior for any program we claim to understand*. Such practices should be part of our general engineering toolchains, and we should not usually be deploying programs where the behavioral properties of the computation (like halting) have not been algorithmically proven.

Of course such sentiments are not entirely new, but from the perspective of an emerging theorist on the matter: it looks as though we have some unresolved paradoxes to fix before we can collectively commit to such sentiment, with the heart we all know it deserves. This paper was written with the explicit intention of advancing us towards that resolution.