# the Computability Hierarchy

Eric Hehner

Department of Computer Science, University of Toronto
hehner@cs.utoronto.ca

**Abstract** A computability hierarchy cannot be constructed by halting oracles.

## Introduction

In a 1937 paper [3], Alan Turing gave the first example of an incomputable function: the halting function. In his 1939 PhD thesis [4], he defined a hierarchy of computability that remains a standard part of theoretical computer science. It works like this.

Let's call your favorite programming language $L$ ; it should include an **if-then-else-fi** construct (the syntax may be different), function definition and call. A halting program $H$ for language $L$ is a program such that, for any program $p$ in $L$ , $H(p)$ is *true* if execution of $p$ terminates, and *false* if it is nonterminating. (I am omitting a coding detail that is not relevant for building the hierarchy; see [0] or [1].) Suppose that $H$ can be written in language $L$ . Define program $D$ as follows:

**if** $H(D)$ **then** *loop* **else** *stop* **fi**

where *loop* is an $L$ program whose execution is nonterminating, and *stop* is an $L$ program whose execution terminates. Then $D$ is an $L$ program. If $H(D)$ is *true* , then $D$ is equivalent to *loop* , and so $H(D)$ is *false* ; if $H(D)$ is *false* , then $D$ is equivalent to *stop* , and so $H(D)$ is *true* . This is inconsistent, so we conclude that $H$ is not in $L$ .

We propose a program $H$ (called an “oracle” for $L$ ) that is not in $L$ , that determines the halting status of all programs in $L$ . Now define language $L+ = L + H$ , by which I mean that $L+$ is all programs in $L$ plus $H$ plus all $L$ programs that call (make use of) $H$ , directly or indirectly. If we suppose that we can write a program $H+$ in language $L+$ that can determine the halting status of all programs in $L+$ , we find an inconsistency by the same argument as above (just add + signs). We propose a program $H+$ (an oracle for $L+$ ) that is not in $L+$ , that determines the halting status of all programs in $L+$ . Define language $L{+}{+} = (L+)+(H+)$ . If we suppose that we can write a program $H{+}{+}$ in language $L{+}{+}$ that can determine the halting status of all programs in $L{+}{+}$ , we find an inconsistency. We propose a program $H{+}{+}$ (an oracle for $L{+}{+}$ ) that is not in $L{+}{+}$ that determines the halting status of all programs in $L{+}{+}$ . And so on. We thus build a hierarchy of languages.

By the Church-Turing thesis, all our standard programming languages can compute the same things; the things they compute are called the computable functions. Let language $L$ be one of these languages. The halting function “implemented” by program $H$ , which determines the halting status of all programs in $L$ , is not computable. (The quotation marks around "implemented" are there because we cannot implement this function in any standard programming language. We "implement" it only by proposing an oracle.)

The hierarchy of languages generalizes computability to "relative computability". The function "implemented" by $H$ is not computable in $L$, but it is computable in $L+$. The function "implemented" by $H+$ is not computable in $L+$, but it is computable in $L++$. And so on. This gives us a hierarchy of computability of mathematical functions.

I believe that what I have said so far is standard, agreed by almost everyone. But I don't agree.

## Russell's Barber

For this story, the only important fact about a man is whom he shaves, so we formalize a man $p$ as a predicate that applies to men. If man $p$ shaves man $q$, then $p(q)$ is *true*; if man $p$ does not shave man $q$, then $p(q)$ is *false*. So we can read " $p(q)$ " as " $p$ shaves $q$ ". A town is a set of men, and there is a town strangely named $L$. A barber for town $L$ is a man who shaves all and only the men in $L$ who do not shave themselves. Suppose $H$ is a barber for $L$. Then, for all men $p$ in $L$,

$$H(p) \ = \ \textbf{if } p(p) \textbf{ then } \textit{false} \textbf{ else } \textit{true} \textbf{ fi}$$

(I could have written $H(p) \ = \ \neg\, p(p)$, but I emphasize the similarity to the halting problem.) If $H$ lives in $L$, then by specializing the previous line,

$$H(H) \ = \ \textbf{if } H(H) \textbf{ then } \textit{false} \textbf{ else } \textit{true} \textbf{ fi}$$

If $H(H)$ is *true*, then the right side of this equation is *false*, and so $H(H)$ is *false*. If $H(H)$ is *false*, then the right side of this equation is *true*, and so $H(H)$ is *true*. This is inconsistent, so we conclude that $H$ does not live in $L$. Let $L+ = L\cup\{H\}$ be a slightly larger town, and let $H+$ be a barber for $L+$. By the same argument, $H+$ does not live in $L+$. Let $L++ = (L+)\cup\{H+\}$ be a still larger town, and let $H++$ be a barber for $L++$. By the same argument, $H++$ does not live in $L++$. And so on up.

Clearly, I intend this story to be analogous to the language and computability hierarchy. Some similarities are apparent. One difference is that the halting problem, if presented in more detail, requires programs to be encoded as data (as natural numbers, or as character strings) so that they can be arguments (or input) to the halting program. That difference can be eliminated by encoding men (as natural numbers, or as character strings like "Tom" and "George") so that men can be first-order functions. The other difference between the stories would seem to be that one is essentially about computing power, and the other is not. I will argue that this difference is inessential: they are the same story expressed in a different incidental vocabulary.

## Onward and Downward

The language and computability hierarchy, starting with $L$ and progressing upward to $L+$ and $L++$ and so on, can be extended in the downward direction. Although we cannot have a program in $L$ to determine halting in all of $L$, we can have a program $H-$ ( $H$ minus) in $L$ to determine halting in most of $L$. It can determine halting in all those programs of $L$

that don't call or make any use, direct or indirect, of $H$- . Let's name that language $L$- ( $L$ minus). Then $L$ = ($L$-)+($H$-) . Similarly we cannot have a program in $L$- to determine halting in all of $L$- , but we can have a program $H$-- in $L$- to determine halting in all those programs of $L$- that don't call or make any use of $H$-- . Let's name that language $L$-- . Then $L$- = ($L$--)+($H$--) . And so on down endlessly.

In the barber story, let $H$- be a resident of $L$ . Although $H$- cannot be a barber for $L$ , he can be a barber for a slightly smaller town $L$- that excludes him. And so on down. If the town $L$ is finite, then this downward sequence ends with an empty town. So, for the sake of making the towns analogous to the programming languages and the barbers analogous to the halting programs, let's say $L$ is an infinite town. Then the downward sequence can go on forever, just like the upward sequence.

## Shavability

The reason the barber for town $L$ cannot live in $L$ has nothing to do with his ability to shave people: a barber can shave anyone, including himself. The problem is that the task we have set for the barber is inconsistent. We ask the barber to shave those other men who do not shave themselves, which is a perfectly reasonable thing for a barber to do; but then we ask the barber to shave himself if and only if he doesn't shave himself, which is just logical nonsense.

It should be clear that the problem has nothing to do with shaving. If we say a chef is someone who cooks for all and only those people who do not cook for themselves, we can make the same argument. If we say a supersticker is a material that sticks to all and only those materials that do not stick to themselves, we can make the same argument. If we say that $R$ is a set that contains all and only those sets that do not contain themselves, we can make the same argument. We can make the same argument about any relation between elements of any set.

## Computability

The reason the halting program for language $L$ cannot be in $L$ has nothing to do with its ability to compute the halting status of programs. It can certainly compute the halting status of programs that do not invoke it. It can even compute its own halting status: it says *true* and then halts. But we ask it to do something logically impossible: for one program we ask it to report *true* if and only if it reports *false* . This logically impossible task is not a well-defined function that just can't be computed. Furthermore, this impossible task has nothing to do with halting. Every property of programs can be “proven” incomputable by exactly the same argument.

(Fine print: at least one program must have the property (needed for the **else**-part), and at least one program must not have the property (needed for the **then**-part); **if** *true* **then** $P$ **else** $Q$ **fi** and $P$ are equivalent with respect to the property; **if** *false* **then** $P$ **else** $Q$ **fi** and $Q$ are equivalent with respect to the property. See [2].)

For example, a calumating program $H$ for language $L$ is a program such that, for any program $p$ in $L$, $H(p)$ is *true* if execution of $p$ calumates, and *false* if it does not calumate. Suppose that $H$ can be written in language $L$. Define program $D$ as follows:

```
    if H(D) then noncal else cal fi
```

where *noncal* is an $L$ program whose execution does not calumate, and *cal* is an $L$ program whose execution calumates. Then $D$ is an $L$ program. If $H(D)$ is *true*, then $D$ is equivalent to *noncal*, and so $H(D)$ is *false*; if $H(D)$ is *false*, then $D$ is equivalent to *cal*, and so $H(D)$ is *true*. This is inconsistent, so we conclude that $H$ is not in $L$. Absurdly, we seem to have proven that calumation of programs in language $L$ cannot be computed by any program in $L$, and we don't even know what calumation means. It seems that we can build a hierarchy of languages and computability on calumation.

The problem with computing the halting function is not about halting, and it isn't about computing either. Assuming $H$ is in $L$, we ask $H$ to compute halting for $L$, and it can't. Assuming $H+$ is in $L+$, we ask $H+$ to report halting for $L+$, but we don't ask it to compute halting by any understood means of computing; it's an oracle and we let it work by magic. Still it cannot do its job, not because of computing limitations (magic is unlimited), but because its job is logically inconsistent. Likewise in $L$ the problem was a logical inconsistency, not a computing limitation.

## Recruiting a Barber

On a particular day, the mayor of town $L$ was concerned that his town had no barber, so he recruited someone named $H-$ from far away. The mayor said to $H-$ that his job would be to shave all and only the men of $L$ who do not shave themselves, including all men currently living there, and all men who might move there in the future, except that his job requirement does not apply to himself; he can shave himself or not as he wishes. So $H-$ accepted the job, and moved to $L$. Town $L$ now includes $H-$ (the barber); $L$ excluding $H-$ is called $L-$ (his customer base). And they all lived happily ever after. Let's call that story $S-$, and compare it to the following story, which we'll call $S$.

Story $S$ starts on the same day as story $S-$, with the same town $L$, same mayor, same lack of barber, same recruiting trip. But in story $S$ the mayor found someone named $H$, and when describing the job, the mayor said that the job requirement applies to the barber himself. $H$ accepted the job, but could not move into $L$ and fulfill his job requirements, so he lived just outside of $L$, in a superset town named $L+$ that consists of $L$ (his customer base) plus $H$ (the barber).

Now compare the stories: the two barbers in their alternate universes have exactly the same task. They have exactly the same customer base, and they shave exactly the same set of people, although the barbers and their customer bases are labeled differently.

## Writing a Halting Program

On a particular day, a manager approached her top programmer, and asked him to write a program, to be called $H-$ when completed, that would determine the halting status of all programs in language $L$ that have ever been written, plus those that have yet to be written but excluding $H$- and those programs that will invoke $H$-, directly or indirectly. She called this subset of $L$ programs the $L$- programs. The programmer realized that, thanks to the exclusion, he can write that program in language $L$, so he set about his task.

By coincidence, on that same day, a different manager in a different company asked her top programmer to write a program, to be called $H$ when completed, that would determine the halting status of all programs in language $L$ that have ever been written, plus those that have yet to be written. She made no exclusion, so the programmer realized that he cannot write the program in $L$. But he didn't just give up and say it would have to be a magic oracle. He knew that the problem with writing the program in $L$ was not lack of computing power, but simply an inconsistency of specification. So he decided to invent a new language that he called $L$+, which is identical to $L$ except in one respect. Identifiers in $L$ are formed by starting with a letter and continuing with zero or more letters and digits; identifiers in $L$+ are formed by starting with a letter and continuing with zero or more letters, digits, and underscores. It should be clear that $L$+ properly includes $L$, that the difference between them is superficial, and that no computing power is gained by adding the new identifiers. When writing $H$, our clever programmer made sure to use an identifier with an underscore in it, so that it would not be an $L$ program.

The point of these stories is that the two programmers have been given exactly the same task, and are writing almost exactly the same program (just one identifier different) in almost exactly the same language. It is of no concern to the $H$ programmer that someone else is writing an $L$ program called $H$-, because $H$- can't invoke $H$, because $H$ isn't in $L$.

## Conclusion

We started with town $L$, but it was not special in any way. We ended with an infinite sequence of nested towns, each one with the same relationship to its enclosing and enclosed towns. No town in this sequence deserves the title of "largest shavable town", meaning that it and all subtowns are shavable, and larger towns are unshavable.

No one of the languages ..., $L$--, $L$-, $L$, $L$+, $L$++, ... has any more claim to defining what's computable than any other. Fortunately, all these languages are Church-Turing equivalent. There is a language hierarchy here, but only in a very shallow sense, and no computability hierarchy. In language $L$ we can't compute $H$, not because $H$ implements a well-defined function that is not computable in $L$, but because the specification of $H$ is inconsistent. To restore consistency, we appear to have two choices. One choice is to ask $H$ to compute a little less; we rename it $H$- and ask it to work on just those $L$ programs that don't invoke $H$-, a subset we call $L$-. The other choice is to say that $H$ isn't in $L$, but in a superset language $L$+ consisting of $L$ plus $H$ and all programs that invoke $H$. The difference between these two choices is superficial; essentially the same program with the two names $H$- and $H$ works on the exact same set of programs with the two names $L$- and $L$.

A computability hierarchy cannot be constructed by halting oracles.

## References

[0] E.C.R.Hehner: Problems with the Halting Problem, *Advances in Computer Science and Engineering* v.10 n.1 p.31-60, 2013, hehner.ca/PHP.pdf

[1] E.C.R.Hehner: Reconstructing the Halting Problem, 2013, hehner.ca/RHP.pdf

[2] H.G.Rice: Classes of Recursively Enumerable Sets and their Decision Problems, *Transactions of the American Mathematical Society* v.74 p.358-366, 1953

[3] A.M.Turing: on Computable Numbers with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society* s.2 v.42 p.230-265, 1936; correction s.2 v.43 p.544-546, 1937

[4] A.M.Turing: Systems of Logic Based on Ordinals, *Proceedings of the London Mathematical Society* s.2 v.45 p.161-228, 1939

## Appendix in answer to a question, added 2014-4-16

In the section titled "Writing a Halting Program", I claimed that we can write a program *H* in language *L+* , with at least one underscore in it somewhere so it is not in language *L* , that determines the halting status of all programs in *L* . To be definite, let's say that *H* has in it just one identifier with an underscore, namely *x_* . And suppose the identifier *x* does not appear in *H* .

In *L* , there is a program *H-* that is almost identical to *H* . The difference is just that all occurrences of *x_* are replaced by *x* . It may seem that these two programs, *H* and *H-* , must be functionally equivalent. If *H* determines the halting status of all programs in *L* , then it may seem that *H-* does too. But we know there is no program in *L* that determines the halting status of all programs in *L* . This may seem to refute my claim that there is a program in *L+* that determines the halting status of all programs in *L* .

Even though programs *H-* and *H* have identical code except for renaming one identifier, they are not equivalent. The difference between them is that programs in *L* can call *H-* (which is in *L* ), but programs in *L* cannot call *H* (which is not in *L* ). The programs that give *H-* trouble are programs in *L* that call *H-* . There is no program in *L* that calls *H* to give *H* trouble.

Program *H* applies to all programs in *L* , and none of them can call *H* , neither directly nor indirectly. If we apply program *H-* to just those programs in *L* that do not call *H-* , neither directly nor indirectly, then on that reduced domain *L-* , *H-* is equivalent to *H* .

## **Addendum** to the previous appendix, added 2014-9-28

If I say “My name is Eric Hehner.”, I am telling the truth. If Margaret Jackson says exactly the same words, she is saying something false. Due to the self-reference, the truth of that sentence depends on who says the sentence.

A town consists of three men: A, B, and C. The job description for a barber says “Shave all and only the men of the town who do not shave themselves.”. D, who lives just outside of town, can do the job. B says that if D can do the job, then so can he; he just has to do exactly what D would do. If $x$ is a man in town whom D would shave, then B will shave $x$. If $x$ is a man in town whom D would not shave, then B will not shave $x$. If D's actions would fulfil the job description, then B reasons that by doing exactly the same actions, he too will fulfil the job description. But B is wrong. If B shaves B, it is an instance of shaving yourself; if D shaves B (same action, different agent), it is not an instance of shaving yourself. Due to the self-reference, whether the job description is consistent depends on who is doing the job.

In the Pascal programming language as originally defined, identifiers cannot have underscores in them. Let me use the name Pascal_ for the language that is identical to Pascal except that identifiers can have underscores in them. We can write a Pascal_ program, let's call it *halts_* , that computes the halting status of all Pascal programs. We don't need to use any identifiers within *halts_* that have underscores; its name prevents it from being a Pascal program. There is a Pascal program, let's call it *halts* , that is identical to *halts_* except for its name. The two programs are functionally identical; whatever result *halts_* computes for a Pascal program, *halts* computes the same result. So if *halts_* fulfils the specification “Compute the halting status of all Pascal programs.”, it may seem that *halts* must fulfil the same specification. But *halts* does not fulfil the same specification. When *halts* is applied to *diag* , it is an instance of applying to a procedure that calls you back; when *halts_* is applied to *diag* , it is not an instance of applying to a procedure that calls you back. No Pascal procedure can call back to *halts_* due to the underscore. It is exactly this construction, applying to a procedure that calls you back, that is used to create the inconsistency. Due to the indirect self-reference, whether this specification is consistent depends on which function, *halts_* or *halts* , is doing the job.

other papers on halting