Objective and Subjective Specifications

Eric C.R. Hehner
Department of Computer Science, University of Toronto
hehner@cs.utoronto.ca

Abstract: We examine specifications for dependence on the agent that performs them. We look at the consequences for the Church-Turing Thesis and for the Halting Problem.

Introduction

The specifications considered in this paper are specifications of behavior, or activity. I include human behavior, computer behavior, and other behavior. To keep the examples simple, I will use specifications that say what the output, or final state, of the behavior should be. And I will use specifications that relate input, or initial state, to output, or final state. The conclusions apply also to specifications that say what the interactions during the behavior should be, but my examples will not be that complicated.

A specification may have the form of a question, for example "What is two plus two?". Or it may have the form of a command, for example "Tell me what is two plus two.". The question and the command are equivalent because they invoke the same behavior. A specification may describe the desired behavior, for example, "saying what is two plus two".

Definitions

A specification is objective if the specified behavior does not vary depending on the agent that performs it. For examples:
(0) Given a natural number, what is its square?
(1) What is the number of words in this question?
(2) What is the name of the first Turing Award winner?
For all three questions, the correct answer does not depend on who or what is answering.

A specification is subjective if the specified behavior varies depending on the agent that performs it. For examples:
(3) What is your name?
(4) What is your IP address?
The correct answer to question (3) depends on whom you ask. In this paper, "subjective" does not mean that the answer is a matter of disagreement, debate, doubt, or dishonesty. If we ask Alice what her name is, the answer "Alice" is correct, and all other answers are wrong. If we ask Bob the same question, the correct answer is different. Question (4) is similar to question (3), but applies to a computer rather than a human.

Subjectively and Objectively Inconsistent

Now consider this example:
(5) Lift Bob.
I am not interested in the variety of lifting techniques; I am interested only in the specified result: the agent lifts Bob. If we ask Hercules, who is very strong, to lift Bob, he can do so without difficulty. If we ask Alice, who is much smaller than Bob, she is not strong enough. The result is different, depending on who is trying to lift Bob. So it may seem that (5) is subjective. But the definition of subjective specification says "the specified behavior varies depending on the agent". When we ask Alice to lift Bob, we are asking for the same behavior (lifting Bob) as when we ask Hercules. So it may seem that (5) is objective. But suppose we

ask Bob to lift Bob. He cannot do so, but not due to lack of strength. He cannot do so because the specification does not make sense when we ask Bob to lift himself. The specification makes sense for some agent (anyone other than Bob), and makes no sense for some agent (Bob). For that reason, (5) is subjective. If we restrict the set of agents to exclude Bob, then (5) is objective.

(6) Can Carol correctly answer "no" to this question?
Let's ask Carol. If she says "yes", she's saying that "no" is the correct answer for her, so "yes" is incorrect. If she says "no", she's saying that she cannot correctly answer "no", which is her answer. So both answers are incorrect. Carol cannot answer the question correctly. Now let's ask Dave. He says "no", and he is correct because Carol cannot correctly answer "no". So (6) is subjective because it is a consistent, satisfiable specification for some agent (anyone other than Carol), and an inconsistent, unsatisfiable specification for some agent (Carol).

(7) Can any man correctly answer "no" to this question?
Let's ask Ed, who is a man. Suppose Ed says "no". Ed is saying that no man can correctly answer "no", and Ed, a man, is answering "no", so Ed is saying that his answer is incorrect. Suppose Ed says "yes". Ed is saying that some man, let's call him Frank, can correctly answer "no". But if Frank answers "no", he is saying that his own answer is incorrect. So Frank cannot say "no" correctly. So Ed's "yes" answer is incorrect. And the same goes for every man. But Gloria, who is not a man, can correctly say "no". Specification (7) is subjective because it is a consistent, satisfiable specification for some agent (anyone who is not a man), and an inconsistent, unsatisfiable specification for some agent (any man).

(8) Can anyone correctly answer "no" to this question?
If we ask Harry and he says "no", he is saying that his answer is incorrect. If he says "yes", he is saying that someone, let's say Irene, can correctly answer "no". But if Irene answers "no", she is saying that her answer is incorrect. So Harry can neither say "no" nor "yes" correctly. And the same goes for anyone else we ask. The correct answer to the question is therefore "no", but no-one can correctly say so (oops, I just did). I meant: no-one who is a possible agent can say so. I exclude myself from the set of possible agents just so that I can tell you that no possible agent can correctly answer "no". Specification (8) is objectively inconsistent.

Specifications (6), (7), and (8) are examples of twisted self-reference. The self-reference occurs when the specification talks about the agent who will perform the specification. The twist, in these examples, is the word "no". If we replace "no" with "yes" in these three specifications, then everyone can correctly answer "yes" to all of them, making them objectively consistent.

Church-Turing Thesis

If a specification can be computed by any one of:
- a Turing Machine (a kind of computer) [4]
- the lambda-calculus (a mathematical formalism) [0]
- general recursive functions (another mathematical formalism) [1][2]

then it can be computed by all of them; they all have the same computing power. The Church-Turing Thesis [5] says that each of these formalisms compute all that is computable. In a more modern version, the Church-Turing Thesis says that if a specification can be computed, then it can be computed by a program in any programming language. All programming languages provide the same computing power; each is equivalent to a Turing Machine.

Church and Turing were thinking of specifications of mathematical functions, like (0). It seems reasonable to me that the Church-Turing Thesis can be extended to all objective specifications. But its extension to subjective specifications comes up against a problem.

Reconsider subjective specification (7), but replace “man” with “L-program”, meaning a program written in programming language L. (L can be Pascal, or Python, or any other programming language. And we define “program” in such a way that the question whether *p* is a program in language L is decidable.)

(9) Can any L-program correctly answer “no” to this question?

It's easy to write an L-program that prints “no”. If that is the answer to (9), it is saying that there isn't an L-program that correctly answers “no” to the question, so in particular, the L-program that prints “no” doesn't give the correct answer. It's just as easy to write an L-program that prints “yes”. If that program is the answer to (9), it says that “no” is the correct answer, so the L-program that prints “yes” doesn't give the correct answer either. In fact, the correct answer to (9) is “no”, but no L-program can correctly say so. We can write a program in language M (which is another programming language) that prints “no” in answer to (9), and that answer is correct. No matter whether the agents are people or programs, the result is the same: one agent can satisfy the specification, but another can't.

The Church-Turing Thesis, in the version stated earlier, does not apply to subjective specifications. Specification (9) can be computed by a program in programming language M, but not by a program in programming language L.

Another version of the Church-Turing Thesis is that any program in any programming language can be translated to a program in any other programming language. We'll look at program translation in a moment, but first, here is a non-programming example to illustrate the translation problem.

(10) Is this question in French?

The correct answer is “no”. The question is easily translated into French.

(11) Cette question est-elle en français?

The correct answer is now “oui”. Before translation, when the question is put to someone who understands the language the question is in, it invokes one behavior: saying “no”. After an accurate translation, when the question is put to someone who understands the language the question is now in, it invokes a different behavior: saying “oui”. Specifications (10) and (11) refer to a language, and changing the language of the question affects the answer.

Similarly, when we write a program to compute a subjective specification, then translate it to another language, it may invoke different behavior. This can occur when the specification refers to a programming language. First, here's an objective specification that refers to a programming language.

(12) Is text *p* an L-program?

Every compiler answers the question whether its input text is a program in the language that it compiles. Whether we write the program that computes (12) in language L or in language M, for the same input *p* we should get the same answer. Specification (12) is objective, and the Church-Turing Thesis applies. Now replace the input with a self-reference.

(13) Is the program answering this question an L-program?

There are two ways to write an M-program to compute (13). The hard way is to give the program access to its own text, perform the lexical analysis and parsing and type checking and so on, just as a compiler would do, and then print the answer, which is “no”. The easy way is just to print “no” because that's the right answer. Now we translate our M-program to language L. If we programmed the hard way, the translated program accesses its own text, does the analysis, and prints the correct answer, which is “yes”. If we programmed the easy way, the translated program prints “no”, which is incorrect. Specification (13) is subjective, and the Church-Turing Thesis does not apply. Either the translation prints the correct answer by exhibiting different printing behavior, or the translation exhibits the same printing behavior and the answer is incorrect.

The same choice, whether to preserve the behavior or to preserve the specification, can occur

without translation, simply by renaming. For example,

(14) Given a text *p* representing a program, determine whether a call to the determining program appears within *p* .

Let's name the determining program *DoYouCallMe* . Given program *p* , it searches within *p* for a calling occurrence of *DoYouCallMe* , reporting "yes" if one is found, and "no" if not. Now let's rename the determining program *DoYouCallMe2* . If we just change the name of the program without changing what it is searching for, this name change preserves behavior, but the program no longer satisfies the specification (14). If we change both the program name and what it is searching for, the program still satisfies the specification (14), but its behavior changes: given the same input, *DoYouCallMe* and *DoYouCallMe2* may give different answers. If a program has access to its own name, then changing its name automatically changes what it is searching for; the result still satisfies (14), but has different behavior.

Yet another version of the Church-Turing Thesis is that in any programming language, you can write an interpreter for programs in any other programming language. Interpretation is the same as executing a translation, and it is similarly limited to programs that satisfy objective specifications. Given a program that computes a subjective specification, its interpretation may produce behavior that differs from execution of the given program.

For objective specifications, I accept the Church-Turing Thesis that we can translate a program from any language to any other language preserving both the specification and the behavior. For subjective specifications we may not be able to preserve both. As we saw for specification (13), we may be able to choose which one we preserve. As we saw for specification (9), it may not be possible to preserve the specification. When a program's behavior depends on the language the program is written in, it may not be possible to preserve the behavior. This last statement might be best explained by making the programming language explicit. If program *p* is written in language L, and on input *x* computes output *y* , write $p\,(\mathrm{L}, x) = y$. Translating to language M, write $p\,(\mathrm{M}, x) = z$. Even though the input remains *x* , the output can change from *y* to *z* because the language has changed from L to M.

Halting Problem, Language-Based

When Alan Turing laid the foundation for computation in 1936 [4], he wanted to show what computation can do, and what it cannot do. For the latter, he invented a problem that we now call the "Halting Problem". Without loss of generality and without changing the character of the problem, I consider halting for programs with no input (input to a program can always be replaced by a definition within the program). In modern terms, it is as follows.

(15) Given a text *p* representing an L-program that requires no input, report "stops" if execution of *p* terminates, and "loops" if execution of *p* does not terminate.

The input *p* represents a program in programming language L. The agent that performs specification (15) must be a program, written in a programming language, running on a computer. (In fact, Turing used the word "machine" for the combination of program and computer, and he used the words "universal machine" for the combination of interpreter program and computer.) I am excluding distributed computations so that I can identify the agent.

First, let's ask for a program written in language L to perform (15), and let's call it *halts* . If there is such a program, then there is also a program in language L, let's call it *twist* , whose execution is as follows:

> *twist* calls *halts* ("*twist*") to determine if its own (*twist* 's) execution will terminate.
> If *halts* reports that *twist* 's execution will terminate,
> then *twist* 's execution becomes a nonterminating loop;
> otherwise *twist* 's execution terminates.

We assume there is a dictionary of function and procedure definitions that is accessible to *halts* , so that the call *halts* ("*twist*") allows *halts* to look up "*twist*" and "*halts*" in the dictionary, and retrieve their texts for analysis.

When programmed in language L, specification (15) is another twisted self-reference. The self-reference is indirect: *halts* applies to *twist* , and *twist* calls *halts* . The twist is supplied by *twist* . If *halts* reports that *twist* 's execution will terminate, then *twist* 's execution is a nonterminating loop. If *halts* reports that *twist* 's execution will not terminate, then *twist* 's execution terminates. Whatever *halts* reports about *twist* , it is wrong. Therefore specification (15) is inconsistent when we ask for a program written in language L to perform it [3].

Now let's ask for a program written in programming language M to perform (15), where M is such that programs written in L cannot call programs written in M. Can this M-program be written? Since L-programs cannot call M-programs, we cannot rule it out by a twisted self-reference. I present two possible answers to the question.

Answer O: Specification (15) is objective, like specification (12). But unlike (12), it is an inconsistent specification, no matter what language we use. If we could write an M-program to compute halting for all L-programs, we could translate it into L (or interpret it by an L-program), and because (15) is objective, the translation (or interpretation) would also compute halting correctly for all L-programs. But there is no L-program to compute halting for all L-programs. So there is no program in any language to compute halting for all L-programs.

Answer S: Specification (15) is subjective. Like specification (13), (15) refers to a programming language L. When programmed in L there is a twisted self-reference; when programmed in M there is no self-reference. There is an M-program to compute halting for all L-programs. Because (15) is subjective, its translation to L (or interpretation in L) does not compute halting for all L-programs. Perhaps the M-program says correctly that *twist* 's execution terminates, and its translation to L (or interpretation in L), which we call *halts* , says incorrectly that *twist* 's execution does not terminate, and that is why *twist* 's execution terminates.

Answer O has been almost unanimously accepted by computer scientists, but its acceptance is premature because (15) has never been shown to be objective, and Answer S has never been ruled out. I favor Answer S for the weak reason that I cannot see any inconsistency in asking for an M-program to compute halting for all L-programs. (Writing the Python program would prove consistency. A logician says that's building a model; the logician's modeling language might be some version of set theory.)

Halting Problem, Location-Based

The preceding discussion of halting is language-based. Here is a similar discussion that is location-based. First a trivial example.

(16) Is this sentence written on page 1?

If (16) is written on page 1, the correct answer is "yes"; if it is written on page 2, the correct answer is "no". Although the answer depends on the location of the question, the answer does not depend on the agent answering, so it is an objective specification. We can create a subjective specification by creating a question that depends on the location of the agent answering.

There are some people in location A, and some other people in different location B. The question is:

(17) Can a person in location A correctly answer "no" to this question?

Anyone in location A who answers "no" to (17) contradicts themself. But Ingrid, who is

standing in location B, can correctly answer "no" to (17) without self-contradiction. When Ingrid walks over to location A, she can no longer correctly answer "no" to (17). The question refers to Ingrid when Ingrid is at A; the question did not refer to Ingrid when Ingrid was at B. Even though she is the exact same person, with the same reasoning power, in either location, a correct answer in one location becomes incorrect in the other.

Suppose there are two identical disconnected computers C and D, and all programs are written in Pascal, and all programs can run on either computer. Both computers have enough memory so that memory limitation is not an issue. (Two computers are necessarily in different locations.)

(18) Given a text *p* representing a Pascal program that requires no input, loaded on computer C, report *true* if execution of *p* terminates, and *false* if execution of *p* does not terminate.

The agent that performs specification (18) must be a Pascal program running on either C or D. Once again, I exclude distributed computing so that I can identify the agent, and once again I assume there is a dictionary of function and procedure definitions on each computer.

First, let's ask for a Pascal program running on computer C to perform (18), and let's call it *halts* . If there is such a program, then we can write another program, let's call it *twist* , exactly as before, and we can load this program onto computer C. As before, *twist* calls *halts* to report on *twist* , and then *twist* does the opposite; so whatever *halts* reports, it is wrong. Specification (18) is inconsistent when we ask for a Pascal program running on computer C to perform it.

Now let's ask for a Pascal program running on computer D to perform (18). Can this program be written? Since programs on C cannot call programs on D (the computers are disconnected), we cannot rule it out by a twisted self-reference. As in the language-based case, we have the same two possible answers to the question: Answer O and Answer S.

Answer O: Specification (18) is objective. It is an inconsistent specification, no matter what computer we use. There is no program on any computer to compute halting for programs on computer C.

Answer S: Specification (18) is subjective. There is a Pascal program on computer D, and again let's call it *halts* , to compute halting for all Pascal programs on computer C. We can carry the *halts* program from D to C and run it there. But when we run it on C, it does not compute halting for all Pascal programs on C. This is quite counter-intuitive. When *halts* applies to *twist* , and *twist* calls *halts* , it matters whether the *halts* that applies (the first occurrence of *halts* in this sentence) is the same instance as the *halts* that is called (the second occurrence of *halts* in this sentence). In one case, there is a twisted self-reference, and in the other, there isn't, and that can affect the computation.

Normally, a program running on one computer will give the same answers to the same questions, with equal validity, as the exact same program running on another computer. This seems obvious, perhaps because it is true for objective specifications. But it is not always true for subjective specifications. The halting specification (18) is a twisted self-reference if the program answering it is on computer C, but not a self-reference if the program answering it is on computer D. So it seems probable that halting is subjective. Even if the program answering it is the exact same one on C and on D, a correct answer from the program running on D may be incorrect from the same program running on C. Furthermore, the same program running on C and D, with the same input, can give different answers to a question that refers to the location of the program.

Conclusion

A specification is objective if the specified behavior does not depend on the agent that performs it, and subjective if it does. The Church-Turing Thesis applies to objective specifications, not to subjective ones. If an objective specification can be implemented as a program in a programming language, it can translated to a program in any other programming language, preserving both the specification and the behavior. If a subjective specification is implemented as a program in a programming language, it may not be possible to translate it to a program in another programming language, preserving both the specification and the behavior.

Let X and Y be two programming languages, or two computers, or two locations. It is inconsistent to ask for an X-program to compute halting for all X-programs due to a twisted self-reference. Twisted self-reference is characteristic of subjective specifications. So it may be consistent and satisfiable to ask for a Y-program to compute halting for all X-programs. At least it has not yet been proven impossible.

Acknowledgement

I thank Bill Stoddart for stimulating discussions.

References

[0] A.Church: *the Calculi of Lambda Conversion*, Princeton University Press, 1941
[1] S.B.Cooper: *Computability Theory*, Chapman&Hall, 2004
[2] K.Gödel, reported by S.C.Kleene: *Introduction to Metamathematics*, North-Holland, 1951
[3] W.Stoddart: “the Halting Paradox”, FACS FACTS: the Newsletter of the Formal Aspects of Computing Science Specialist Group, 2018 January
[4] 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
[5] A.M.Turing: Systems of Logic based on Ordinals, p.8, Ph.D. thesis, Princeton University, 1939

other papers on halting