Incomputability According to aPToP

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

Let \( S \) be a specification in state variables \( \sigma \) (in mathematical variables \( \sigma \) and \( \sigma' \)), not including a time variable, such that

1. \( \forall \sigma \cdot \exists \sigma' \cdot S \) \text{ " } S \text{ is implementable } \)
2. \( \forall \sigma \cdot \exists \sigma' \cdot \neg S \) \text{ " } \neg S \text{ is implementable } \)

For example, \( S \) could be \( x'=0 \).

Define

\[
D = \text{if } \forall \sigma, \sigma' \cdot S \leftarrow D \text{ then } \neg S \text{ else } S \text{ fi}
\]

The if-part, \( \forall \sigma, \sigma' \cdot S \leftarrow D \), says that \( D \) is an implementation of \( S \). It has no nonlocal variables; either it is the constant \( \top \) or it is the constant \( \bot \). Suppose it is \( \top \). Then

\[
\top \Rightarrow (\forall \sigma, \sigma' \cdot S \leftarrow D) \land (\forall \sigma, \sigma' \cdot D = \neg S) \land (\forall \sigma, \sigma' \cdot S \leftarrow D) \land (\forall \sigma, \sigma' \cdot D = S = \neg S) \Rightarrow \forall \sigma, \sigma' \cdot S \land (\forall \sigma \cdot \exists \sigma' \cdot \neg S) \Rightarrow \bot
\]

Suppose \( \forall \sigma, \sigma' \cdot S \leftarrow D \) is \( \bot \). Then

\[
\top \Rightarrow \neg (\forall \sigma, \sigma' \cdot S \leftarrow D) \land (\forall \sigma, \sigma' \cdot D = \text{if } \forall \sigma, \sigma' \cdot S \leftarrow D \text{ then } \neg S \text{ else } S \text{ fi}) \Rightarrow \neg (\forall \sigma, \sigma' \cdot S \leftarrow D) \land (\forall \sigma, \sigma' \cdot D = S) \Rightarrow \bot
\]

We have an inconsistency, introduced by definition (2). Assumption (0) is needed so that it is sensible to talk about implementing specification \( S \). Assumption (1) is needed to prove that (2) is an inconsistent definition.
Recursive Construction

Let's try recursive construction to see what it might tell us about definition (2).

\[
D_0 = \top \\
D_1 = \begin{cases} \neg S & \text{if } \forall \sigma, \sigma' \cdot S \Leftarrow D_0 \\
S & \text{else} \end{cases} \\
D_2 = \begin{cases} \neg S & \text{if } \forall \sigma, \sigma' \cdot S \Leftarrow D_1 \\
S & \text{else} \end{cases} \\
D_3 = \begin{cases} \neg S & \text{if } \forall \sigma, \sigma' \cdot S \Leftarrow D_2 \\
S & \text{else} \end{cases}
\]

We cannot form \( D_\infty \), and the limit axioms are uninformative about \( \lim n \cdot D_n \), as we would expect for an inconsistent definition. No specification \( D \) satisfies (2).

Relation to Halting Problem

If \( S \) is a specification (such as \( x' = 0 \)) and \( P \) is a program, then \( \forall \sigma, \sigma' \cdot S \Leftarrow P \) says that \( P \) refines \( S \), or in other words, execution of program \( P \) has property \( S \). If we could compute \( \forall \sigma, \sigma' \cdot S \Leftarrow P \), we would be computing whether execution of program \( P \) has property \( S \). This is related to the halting problem, but there are differences. One obvious difference is that I excluded time, so I cannot talk about termination. But even without time, specifications like \( x' = 0 \) are supposed to be incomputable (meaning there is no program to determine whether an arbitrary program refines it) by the same kind of argument that is supposed to prove that halting is incomputable.

Another difference between the preceding proof of inconsistency and the halting problem is that I have not assumed that \( \forall \sigma, \sigma' \cdot S \Leftarrow P \) can be computed for arbitrary program \( P \), and not supposed that \( D \) is a program. Nonetheless, we find an inconsistency: for specification \( D \), \( \forall \sigma, \sigma' \cdot S \Leftarrow D \) can neither be \( \top \) nor \( \bot \). Of course, (2) remains inconsistent if we assume that \( \forall \sigma, \sigma' \cdot S \Leftarrow P \) can be computed for arbitrary program \( P \), and that \( D \) is a program, but the inconsistency is not due to these assumptions.

In the previous paragraph, I mentioned two assumptions: that \( \forall \sigma, \sigma' \cdot S \Leftarrow P \) can be computed for arbitrary program \( P \), and that \( D \) is a program. The halting problem argument makes only one assumption: that \( \forall \sigma, \sigma' \cdot S \Leftarrow P \) can be computed for arbitrary program \( P \). Then a definition is displayed, and claimed to be a program. For example, in one variable \( x \),

\[
D = \begin{cases} \text{if } (\text{expression computing } \forall x, x' \cdot x' = 0 \Leftarrow D) \text{ then } x := 1 \text{ else } x := 0 \end{cases}
\]

Under the assumption that we can compute \( \forall x, x' \cdot x' = 0 \Leftarrow D \), \( D \) is clearly a program. But that is not the assumption. The assumption is that we can compute \( \forall x, x' \cdot x' = 0 \Leftarrow P \) for all programs \( P \). If \( D \) is a program, then we can compute \( \forall x, x' \cdot x' = 0 \Leftarrow D \), and therefore \( D \) is a program. But if \( D \) is not a program, we have not assumed that \( \forall x, x' \cdot x' = 0 \Leftarrow D \) can be computed, and so we cannot say that \( D \) is a program.
Simplest Instance

I would now like to look at the simplest instance. Let \( b \) be a binary state variable (\( b \) and \( b' \) are binary mathematical variables); there are no other state variables. There are only 16 specifications in 2 binary variables \( b \) and \( b' \). Of these, only 9 are implementable: \( b' \), \( \neg b' \), \( b \Rightarrow b' \), \( \neg b \Rightarrow b' \), \( b = b' \), \( b \oplus b' \), \( \top \). Of these, only 4 are deterministic: \( b' \), \( \neg b' \), \( b = b' \), \( b \oplus b' \). These are the only specifications that are implementable and whose negations are implementable. They are the only specifications that can be written as programs:

\[
b := \top, \quad b := \bot, \quad \text{ok, } b := \neg b.
\]

For specification \( S \), I arbitrarily choose \( b' \). Define

\[
(3) \quad D = \text{if } \forall b, b' : b' \Leftarrow D \text{ then } b := \bot \text{ else } b := \top \text{ fi}
\]

The incomputability argument is as follows:

Assume we can compute \( \forall b, b' : b' \Leftarrow D \) and that we replace it in (3) with an expression to compute it. Then \( D \) is a program. Since \( \forall b, b' : b' \Leftarrow D \) has no nonlocal variables, either it is the constant \( \top \) or it is the constant \( \bot \). Suppose is it \( \top \).

\[
\begin{align*}
\top &= (\forall b, b' : b' \Leftarrow D) \land (\forall b, b' : D = \text{if } \forall b, b' : b' \Leftarrow D \text{ then } b := \bot \text{ else } b := \top \text{ fi}) \quad \text{context} \\
\Rightarrow & (\forall b, b' : b' \Leftarrow D) \land (\forall b, b' : D = (b := \bot)) \\
\Rightarrow & \forall b, b' : b' \Leftarrow (b := \bot) \\
\Rightarrow & \bot
\end{align*}
\]

Suppose \( \forall b, b' : b' \Leftarrow D \) is \( \bot \). Then

\[
\begin{align*}
\top &= \neg(\forall b, b' : b' \Leftarrow D) \land (\forall b, b' : D = \text{if } \forall b, b' : b' \Leftarrow D \text{ then } b := \bot \text{ else } b := \top \text{ fi}) \quad \text{context} \\
\Rightarrow & \neg(\forall b, b' : b' \Leftarrow D) \land (\forall b, b' : D = (b := \top)) \\
\Rightarrow & \neg \forall b, b' : b' \Leftarrow (b := \top) \\
\Rightarrow & \bot
\end{align*}
\]

We have an inconsistency. Therefore (according to the incomputability argument) the assumption that we can compute \( \forall b, b' : b' \Leftarrow D \) is wrong.

The inconsistency just proven is an instance of the inconsistency proven more generally earlier, without the assumption that \( \forall b, b' : b' \Leftarrow D \) is computable. There are infinitely many programs on this state space, but each of them can be algorithmically reduced to one of the basic 4, and \( \forall b, b' : b' \Leftarrow D \) is easily computable: when \( D \) is \( b := \top \) it is \( \top \), and for the other 3 programs it is \( \bot \).
Discussion

For Turing, and for most of the programming world today, even for most of the formal methods community, programs and specifications are different things. They have separate languages in which they are expressed. Programs are instructions to a computer, not mathematical expressions. (Turing used the word “machine” to mean the combination of computer and program.) Formal specifications are mathematical expressions, not instructions to a computer. Specifications are used to reason about programs.

To the majority of people, the Halting Problem is about programs, not about specifications. The computability assumption is essential; we have to assume the existence of a program to compute halting so that we can reason about it. An inconsistency is created by the assumption that such a program exists. The reasoning that there is an inconsistency uses specifications, but, to the majority of people, the specifications are talking about programs.

In the aPToP world [0], programs are a subset of specifications. We do not specify programs; we specify computation, or computer behavior. Programs are those specifications that a computer can execute. If we prove something about all specifications, that includes programs. If we find an inconsistency in a collection of specifications, that finding is unaffected by an assumption that one or more of the specifications are programs. An inconsistency in a specification cannot be caused by the assumption that the specification is a program.

Maybe a revision of the Halting Problem will have to wait (possibly forever) until the aPToP view of programming is accepted.

Reference