计算机暗示与库里的悖论

韦恩·艾特肯，杰弗里·A.巴雷特

摘要。理论上有局限性，即计算机可以实现的功能。在这篇论文中，我们关注计算机实施推断的强度的局限性。我们使用库里的悖论来得出这一结论。

1. 引言

计算机可以用于检查属性。事实上，计算机程序的属性可以通过适当设计的程序来检查。众所周知，有极限。测试终止属性就是其中之一。在这个论文中，我们关注计算机可以检查的另一个限制。在这里，我们对计算机实施推断的强度感兴趣。

一个推断程序是一个程序，它接受两个关于程序行为的声明，然后尝试根据指定的规则库推断第二个声明。它会一直查找，直到找到推断，因此它可能会永远不终止。如果它找到一个推断，它就会终止并输出1来表示已经找到了证明。推断程序可以用于证明关于推断程序本身的陈述。因为递归和部分递归函数可以被实现为计算机程序，所以推断程序也可以用于证明这些函数之间的关系。在本文中，我们假设程序是用固定语言为计算机写作，带有无限制的内存。

在本文中，我们指出推断程序的一个限制，即没有足够强大的推断程序可以包含不受限制的命题。更广泛地说，我们证明了有一些可以定义算法的规则，但是无法由给定的推断程序使用。在这里，我们提供一个非形式化的版本的论证。一个更正式、更严谨、因而更长的版本将在未来论文中给出。未来论文将是一个公理性证明，并证明这个结果不仅适用于计算机程序或图灵机，而且适用于更一般的组合性结构。未来论文还将讨论我们的结果与约翰·迈希尔的推断层次[3]之间的关系。

2. 陈述

一个陈述是[prog, in, out]的列表，其中prog是一个被认为是数据的程序，in是一个输入，out是期望的输出。一个陈述[prog, in, out]

日期：2003年8月17日。
is called \textit{true} if the program \textit{prog} halts with input \textit{in} and output \textit{out}. A statement which is not true is called \textit{false}.

There is a program to check but not test whether \([\textit{prog}, \textit{in}, \textit{out}]\) is a true statement. Given \([\textit{prog}, \textit{in}, \textit{out}]\) as an input, it first runs \textit{prog} as a subprocess with input \textit{in}. If and when \textit{prog} halts, it compares the actual output with \textit{out}. If they match then the program outputs 1; if they do not match, our program does something else (say, outputs 0). This program will output 1 if and only if \([\textit{prog}, \textit{in}, \textit{out}]\) is true, but it might not halt if \([\textit{prog}, \textit{in}, \textit{out}]\) is false. Due to the halting problem, no program can check for falsity.

In what follows, we continue to use 1 to signal a positive result. We use 0 to signal a negative result, but failure to halt also indicates (but does not signal to a real user) a negative result.

3. Valid Rules

A \textit{rule} is a program that takes as input a list of statements and outputs a list of statements. For convenience we require that the output list include the input list as a sublist, and that the rule halt for all input lists. A \textit{valid rule} is a rule with the property that whenever the input list consists of all true statements, the output list also consists of all true statements.

Consider the program \textit{and} which expects as an input a list of two statements \([A, B]\). The program first checks the truth of \textit{A} in the manner indicated above. If it determines that \textit{A} is true, then it checks the truth of \textit{B}. If \textit{B} is also true, it outputs 1. If either \textit{A} or \textit{B} is false, and fails to halt.

An example of a valid rule concerning \textit{and} is the program \textit{and-elim}. This program expects as input a list of statements (if the input is not of this form then \textit{and-elim} just outputs 0). It looks for statements of the form \([\textit{and}, [A, B], 1]\) on the input list. If no such statement is found, \textit{and-elim} outputs the list which was given as an input. Otherwise, \textit{and-elim} creates an output list consisting of every statement on the input list plus, for each statement of the form \([\textit{and}, [A, B], 1]\) appearing on the input list, the two statements \textit{A} and \textit{B}. Given the definition of \textit{and}, if all the statements on the input list are true and if \([\textit{and}, [A, B], 1]\) is on the input list, then \textit{A} and \textit{B} are also true. So the program \textit{and-elim} only adds true statements to the input list and hence is a valid rule.

4. An Implication Program

A \textit{library} is the list of rules used by an implication program in its proofs. We assume here that the library is finite at any given time. A \textit{valid library} is one that contains only valid rules.

Consider the implication program \(\Rightarrow\) defined as follows. The program \(\Rightarrow\) expects as input a list of two statements \([A, B]\). Then it sets up and manipulates a list of sentences called the \textit{consequence list}. The consequence list begins as a singleton list consisting only of \textit{A}. The program \(\Rightarrow\) then goes to the library and chooses a rule. It applies the rule to the consequence list, and the result becomes the new consequence list. Since rules are required to include the input list as a sublist of the output list, once a statement appears on any consequence list it will appear on all subsequent consequence lists. After applying a rule, the program \(\Rightarrow\) checks whether the consequent \textit{B} is on the new consequence list. If so, it outputs 1; otherwise it chooses another rule, applies it to update the consequence list, and checks for \textit{B}
on the new consequence list. It continues to apply the rules in an exhaustive way until \( B \) is found, in which case \( \Rightarrow \) outputs 1. If the consequent \( B \) is never found, the implication program \( \Rightarrow \) does not halt.

It does not matter how the program \( \Rightarrow \) selects rules to apply to the consequence list as long as each rule is repeatedly applied. The program \( \Rightarrow \) might, for example, use the sequence 1,1,2,1,2,3,1,2,3,4,... to determine which order to apply the rules. This sequence directs \( \Rightarrow \) to first use the first rule in the library, then the first rule again, then the second rule in the library, etc. If it is time to use the \( n \)th rule, but there is no \( n \)th rule in the library, then it just ignores that step and goes to the next.

If one adds new rules to the library, a currently running implication program might be designed to dynamically incorporate the new rules in its search for the consequent. An implication program need not have this feature for the considerations below to hold. But adding new rules and using the sequence described above allows one to consider a countable infinite number of rules if needed.

5. Modus Ponens

Consider *Modus Ponens Program* \( mp \). The program \( mp \) expects as an input a list of statements. It begins by forming an empty result list. It then searches the input list for all statements of the form \([ \Rightarrow, [A, B], 1] \) where \( A \) and \( B \) are statements. For each such statement, it searches to see if \( A \) also appears as a statement on the input list. If \( A \) is found, then \( mp \) adds \( B \) to the result list. After constructing the result list, the program \( mp \) outputs a list consisting of all the statements in the input list followed by all the statements of the result list (if any).

The program \( mp \) is a rule since its output list contains every statement on its input list. A rule is valid if, for an input list of true statements, it only adds true statements. From the definition of \( \Rightarrow \), if \([ \Rightarrow, [A, B], 1] \) and \( A \) are on the input list and if they are both true and if the library is valid, then \( B \) will be true. So \( mp \) is a valid rule if the library used by \( \Rightarrow \) is valid.

6. The Curry Program

Some statements are clearly false. Consider the program \( eq \). It expects as input a list \([m, n]\) of two natural numbers. If \( m = n \) then \( eq \) outputs 1, otherwise it outputs 0. Let \( false \) be the false statement \([eq, [0, 1], 1] \).

Consider the program \( curry \) defined as follows. It expects a program \( X \) as input. Then it runs \( \Rightarrow \) as a subprocess with input \([ [X, X, 1], false] \). The output of the subprocess (if any) is then used as the output of \( curry \).

If \( X \) checks for a particular property of programs, then the statement \([X, X, 1]\) asserts that the program \( X \) has the very property for which it checks. The program \( CURRY \) when applied to program \( X \) can be thought of as trying to find a proof by contradiction that the statement \([X, X, 1]\) does not hold.

7. The Ad Hoc Rule

From the definition of \( curry \), the only way that \( curry \) could output 1 with input \( X \) is if \( \Rightarrow \) outputs 1 with input \([ [X, X, 1], false] \). In other words, if \([CURRY, X, 1]\) is a true statement, then so is \([ \Rightarrow, [ [X, X, 1], false], 1] \). This is the idea behind the Ad Hoc Rule.
The Ad Hoc Rule AH expects as input a list of statements. It begins by producing an empty result list. Then it checks its input for statements of the form \([\text{curry}, X, 1]\) where \(X\) is a program. For each such statement on the input list, AH adds the statement \([\Rightarrow, ([X, X, 1], \text{false}], 1]\) to the result list. When AH is finished constructing the result list, it outputs a list consisting of the statements in the input list followed by the statements of the result list (if any).

If the statements on the input list are true, then AH only adds true statements to form the output list. So AH is a valid rule.

The valid rule AH is ad hoc since it is designed specifically for curry. With a little more work, the same effect can be achieved by using a collection of more natural, logically motivated rules, each independent of curry. This more natural approach is pursued in the promised future paper.

8. Curry’s Paradox

We now describe a version of Curry’s paradox\(^1\) for the implication program \(\Rightarrow\). We assume that the library is valid and contains the Modus Ponens Rule \(\text{mp}\) and the Ad Hoc Rule AH.

Consider what happens when we run \(\Rightarrow\) with input \([\text{curry}, \text{curry}, 1], \text{false}\]. First a consequence list containing the statement \([\text{curry}, \text{curry}, 1]\) is set-up. Next rules from the library are applied to the consequence list. At some point the Ad Hoc Rule AH is applied and, since \([\text{curry}, \text{curry}, 1]\) is on the consequence list, \([\Rightarrow, ([\text{curry}, \text{curry}, 1], \text{false}], 1]\) is added to the consequence list. Because of this, when \(\text{mp}\) is next applied to the consequence list, \(\text{false}\) will be added to the list. Since the initial input had the statement \(\text{false}\) as the second item on the input list, \(\Rightarrow\) will halt with output 1 when \(\text{false}\) appears on the consequence list.

So \(\Rightarrow\) outputs 1 with input \([\text{curry}, \text{curry}, 1], \text{false}\]. From the definition of \(\text{curry}\), this implies that \(\text{curry}\) will output 1 when \(\text{curry}\) is given as an input. That is, the statement \([\text{curry}, \text{curry}, 1]\) is true.

Consider again what happens when \(\Rightarrow\) is applied to \([\text{curry}, \text{curry}, 1], \text{false}\]. Since the antecedent \([\text{curry}, \text{curry}, 1]\) is true, every statement added to the consequence list is also true. Above we noted that \(\text{false}\) will eventually appear on the consequence list. Therefore, \(\text{false}\) is true, a contradiction.

Curry’s paradox typically arises in logical systems designed to capture natural inference, and in such cases there are several assumptions one might question. But here the paradox occurs in the concrete setting of perfectly well-defined programs and seemingly careful reasoning about their expected behavior. So what went wrong?

9. The Resolution

We assume above that the inference library is valid and that it contains the Modus Ponens Rule \(\text{mp}\). We showed that if the library is valid, then the rule \(\text{mp}\) is in fact valid, but this does not mean that \(\text{mp}\) is in the valid library. Indeed, the argument above can be regarded as a proof (by contradiction) that \(\text{mp}\) is not in any valid library containing the rule AH (or containing rules sufficiently strong to mimic AH).

\(^1\)Named for its use in [2]. See [1] for more information.
We conclude from this that, given an implication program, there are valid inference rules (including the associated Modus Ponens Rule $\text{mp}$) that are valid only so long as they are not included in the library of rules used by the implication. Informally, we can say that there are valid rules that one is not allowed to use (in an unrestricted manner) in one’s proofs.

In the discussion above we allowed for the possibility of an open library; that is, a library where valid rules can be added as developed. It is now clear that a rule’s validity is not itself a sufficient condition for it being safe to add it to a valid library. Rather, in order to maintain a valid open library, one must check that the rule is valid and that it remains valid when added to the library. Call a rule independently valid if it is valid regardless of which library is used by the implication program. Examples of independently valid rules include AH and AND-ELIM. Clearly, any library consisting only of independently valid rules is valid.

The rule $\text{mp}$ is not independently valid; its validity is contingent on the nature of the library. In fact, Curry’s paradox gives examples of libraries for which $\text{mp}$ is not valid. Our proof that $\text{mp}$ was valid only works for some libraries, namely the valid libraries.

References


Cal. State, San Marcos, CA 92096, USA
E-mail address: waitken@csusm.edu

UC Irvine, Irvine, CA 92697, USA
E-mail address: jabarret@uci.edu