Computation of recursive programs—Theory vs practice

by ZOHAR MANNA
Stanford University
Stanford, California

This note is actually an informal exposition of a part of a recent paper by Manna, Ness and Vuillemin. We have two main purposes in this note. First, we present some known results about computation of recursive programs, emphasizing some differences between the theoretical and practical approaches. Second, we introduce the computational induction method for proving properties of recursive programs. It turns out that most known methods for proving properties of programs are very closely related to the computational induction method. We illustrate this point by showing how Floyd's inductive assertions method for proving properties of "flowchart programs" can be expressed in terms of computational induction on recursive programs.

The reader should be aware that some of the results presented in this note hold only under certain restrictions which are ignored in this informal presentation.

RECURSIVE PROGRAMS

To simplify our discussion, we shall restrict ourselves to a particularly simple language, chosen because of its similarity to familiar languages such as ALGOL or LISP. A program in our language, called a recursive program, is of the form

\[ F(x) \leq_r [F](x) \]

where \([F](x)\) is a composition of known functions and the function variable \(F\), applied to the individual variables \(x = (x_1, x_2, \ldots, x_n)\). The following, for example, is a recursive program over the integers

\[ P_0: \quad F(x_1, x_2) \leq \begin{cases} x_1 = x_2 \text{ then } x_2+1 \\ \text{else } F(x_1, F(x_1-1, x_2+1)) \end{cases} \]

We allow our known functions to be partial, i.e., they may be undefined for some arguments. This is quite natural, since our known functions represent the result of some computation, and a computation process may in general give results for some arguments and run indefinitely for others. We include as limiting cases of partial functions, the partial functions defined for all arguments (called total functions) as well as the partial function undefined for all arguments.

Let us consider now the following partial functions:

\[ f_1(x_1, x_2) : x_1 + 1 \]
\[ f_2(x_1, x_2) : \begin{cases} x_1 \geq x_2 \text{ then } x_1 + 1 \text{ else } x_2 - 1, \text{ and} \\ f_3(x_1, x_2) : \begin{cases} (x_1 \geq x_2) \land (x_1 - x_2 \text{ even}) \text{ then } x_1 + 1 \\ \text{else undefined.} \end{cases} \]

These three functions have an interesting common property: For each \(i\) \((1 \leq i \leq 3)\), when we replace all occurrences of \(F\) in the program \(P_0\) by \(f_i\), the lefthand side and the righthand side of the \(\leq\) yield identical partial functions, namely,

\[ f_i(x_1, x_2) = \begin{cases} x_1 = x_2 \text{ then } x_2 + 1 \text{ else } f_i(x_1, f_i(x_1-1, x_2+1)) \end{cases} \]

It is straightforward to see that the equality holds for \(f_1(x_1, x_2)\), for example, since in this case we get

\[ x_1 + 1 = \begin{cases} x_1 = x_2 \text{ then } x_2 + 1 \text{ else } x_1 + 1, \end{cases} \]

which is clearly true. One can similarly verify that \(f_2\) and \(f_3\) have the same property. We therefore say that the functions \(f_1, f_2\) and \(f_3\) are fixpoints of the recursive program \(P_0\).

Among the three functions, \(f_3\) has one important special property: for any \((x_1, x_2)\) such that \(f_3(x_1, x_2)\) is defined, i.e., \((x_1 \geq x_2) \land (x_1 - x_2 \text{ even})\), both \(f_1(x_1, x_2)\) and \(f_2(x_1, x_2)\) are also defined and have the same value as \(f_3(x_1, x_2)\). For short, we say that \(f_3\) is "less defined" than \(f_1\) and \(f_2\) and denote this by \(f_3 \subseteq f_1\) and \(f_3 \subseteq f_2\). It can be shown that \(f_3\) has this property not only with respect to \(f_1\) and \(f_2\) but with respect to all fixpoints of the recursive program \(P_0\). Moreover, \(f_3(x_1, x_2)\) is the only function having this property; \(f_3\) is therefore said to be the least (defined) fixpoint of \(P_0\).

One of the most important results related to this topic is due to Kleene, who showed that every recursive program \(P\) has a unique least fixpoint (denoted by \(f_P\)).
In discussing our recursive programs, the key problem is: What is the partial function \textit{f} defined by a recursive program \textit{P}? There are two viewpoints.

(a) Fixpoint approach: Let it be the unique least fixpoint \( f_P \).
(b) Computational approach: Let it be the computed function \( f_C \) for some given computation rule \( C \) (such as “call by name” or “call by value”).

Now we come to a very interesting point. All the theory of proving properties of recursive programs is actually based on the assumption that the function defined by a recursive program is exactly the least fixpoint \( f_P \). That is, the fixpoint approach is adopted. Unfortunately, many commonly used programming languages imply computation rules for evaluating such recursive programs (such as “call by value!”) which do not necessarily lead to the least fixpoint.

Let us consider, for example, the following recursive program over the integers
\[
P_1: \quad F(x_1, x_2) \equiv \begin{cases} 
1 & \text{if } x_1 = 0 \\
F(x_1 - 1, F(x_1 - x_2, x_2)) & \text{else}
\end{cases}
\]

The least fixpoint \( f_P \), can be shown to be
\[
f_P(x_1, x_2) = \begin{cases} 
1 & \text{if } x_1 \geq 0 \\
\text{undefined} & \text{else}
\end{cases}
\]

However, the computed function \( f_C \) where \( C \) is “call by value,” turns out to be
\[
f_C(x_1, x_2) = \begin{cases} 
1 & \text{if } x_1 = 0 \lor [x_1 > 0 \land x_2 > 0 \land (x_2 \text{ divides } x_1)] \\
\text{undefined} & \text{else}
\end{cases}
\]

Thus, \( f_C \) is properly less defined than \( f_P \), e.g., \( f_C(1, 0) \) is undefined while \( f_P(1, 0) = 1 \).

There are two ways to view this problem: either (a) theoreticians are wasting their time by developing methods for proving properties of programs which “do not exist” in practice. They should concentrate their efforts in developing direct methods for proving properties of programs as they are actually executed; or (b) existing computer systems should be modified, since they are computing recursive programs in a way which prevents their user from benefiting from the results of the theory of computation. Language designers and implementors should look for efficient computation rules which always lead to the least fixpoint can be obtained by modifying “call by value” so that the evaluation of the arguments of the function variable \( F \) are delayed as long as possible.

One way to cope with the problem would be to develop translation techniques, so that for every given recursive program \( P \) and computation rule \( C \), we can construct a recursive program \( P' \) such that the computed function of \( P \) is identical to the least fixpoint of \( P' \). The verification techniques can then be applied to \( P' \). This is probably the way many verification systems are going to work in the future. The main reason for adopting this approach is the existence of a very powerful method, the computational induction method, for proving properties of the least fixpoint of recursive programs. Most known methods for proving properties of programs can be expressed in terms of the computational induction method, as illustrated later. The computational induction method has two important advantages over the other methods: First, it is very convenient for machine implementation; and second, termination and equivalence proofs can be handled in exactly the same way as correctness proofs.

The Computational Induction Method

We now describe the computational induction method for proving properties of recursive programs. The idea is essentially to prove properties of the least fixpoint \( f_P \) of a given recursive program \( P \) by induction on the level of recursion.

Let us consider, for example, the recursive program
\[
P_2: \quad F(x) \equiv \begin{cases} 
1 & \text{if } x = 0 \\
x \cdot F(x - 1) & \text{else}
\end{cases}
\]

over the natural numbers. The least fixpoint function \( f_P(x) \) of this recursive program is the factorial function \( x! \).

Let us denote now by \( f(x) \) the partial function indicating the “information” we have after the \( i \)th level of recursion. That is,
\[
f^0(x) \text{ is undefined (for all } x); \\
f^1(x) \text{ is if } x = 0 \text{ then } 1 \text{ else } x \cdot f^0(x - 1), \text{i.e.,} \\
\quad \text{if } x = 0 \text{ then } 1 \text{ else undefined}; \\
f^2(x) \text{ is if } x = 0 \text{ then } 1 \text{ else } x \cdot f^1(x - 1), \text{i.e.,} \\
\quad \text{if } x = 0 \text{ then } 1 \text{ else undefined (if } x - 1 = 0 \text{ then } 1 \text{ else undefined), or in short,} \\
\quad \text{if } x = 0 \lor x = 1 \text{ then } 1 \text{ else undefined; etc.}
\]
In general, for every \( i, i \geq 1 \),
\[ f^i(x) = \begin{cases} 
1 & \text{if } x = 0 \\
 x \cdot f^{i-1}(x-1) & \text{else}
\end{cases} \]
which is
\[ \begin{cases} 
1 & \text{if } x < i \\
 x! & \text{else undefined.}
\end{cases} \]
This sequence of functions has a limit which is exactly the least fixpoint of the recursive program; that is,
\[ \lim_{i \to \infty} f^i(x) = x! \]
The important point is that this is actually the case for any recursive program \( P \). That is, if \( P \) is a recursive program of the form
\[ F(x) \Rightarrow r[F](x), \]
and \( f^i(x) \) is defined by
\[ f^i(x) = \begin{cases} 
\text{undefined} & \text{if } x < i \\
r[f^{i-1}](x) & \text{else}
\end{cases} \]
then
\[ \lim_{i \to \infty} f^i(x) = \text{fp}(x). \]
This suggests an induction rule for proving properties of \( \text{fp} \).
To show that some property \( \varphi \) holds for \( \text{fp} \), i.e., \( \varphi(\text{fp}) \), we show that \( \varphi(Ji) \) holds for all \( i \geq 0 \), and therefore we may conclude that \( \varphi(\text{fp}) \), holds.
There are two ways to state this rule. Both are essentially equally powerful. These are actually the rules for simple and complete induction on the level of recursion.

(a) **Simple induction**
\[ \varphi(f^0) \text{ holds and } \forall i [\varphi(f^i) \Rightarrow \varphi(f^{i+1})] \text{ holds,} \]
then \( \varphi(f^i) \) holds.

(b) **Complete induction**
\[ \forall i [\forall j \text{ s.t. } j < i \varphi(f^j) \Rightarrow \varphi(f^i)] \text{ holds,} \]
then \( \varphi(f^i) \) holds.

The simple induction rule is essentially the "\( \mu \)-rule" suggested by deBakker and Scott,6 while the complete induction rule is the "truncation induction rule" of Morris.7

**Example:** Consider the two recursive programs7
\[ P_1: \quad F(x, y) \Leftrightarrow \begin{cases} 
1 & \text{if } p(x) \text{ then } y \text{ else } h(F(k(x), y)) \\
\text{undefined} & \text{if } x < 1 \text{ f. x. y}
\end{cases} \]
and
\[ P_2: \quad G(x, y) \Leftrightarrow \begin{cases} 
1 & \text{if } p(x) \text{ then } y \text{ else } G(k(x), h(y)) \\
\text{undefined} & \text{if } x < 1 \text{ f. x. y}
\end{cases} \]

For our purpose there is no need to specify the domain of the programs or the meaning of \( p, h \) and \( k \). We would like to prove, using the two forms of induction, that
\[ f_{P_1}(x, y) = g_{P_1}(x, y) \text{ for all } x \text{ and } y. \]

**Proof by simple induction**
If we restrict ourselves to simple induction, it is much more convenient to prove a stronger result than the desired one. This often simplifies proofs by induction by allowing a stronger induction assumption, even though we have to prove a stronger result. So, we actually show that
\[ \varphi(f_{P_1}, g_{P_1}) : \forall x \forall y [f_{P_1}(x, y) = g_{P_1}(x, y)] \]
holds. We proceed in two steps:

(a) \( \varphi(f^0, g^0) \), i.e., \[ \forall x \forall y [f^0(x, y) = g^0(x, y)] \]
\[ \wedge [g^0(x, h(y)) = h(g^0(x, y))]. \]
\[ \forall x \forall y [\text{undefined} = \text{undefined}] \]
\[ \wedge [\text{undefined} = \text{undefined}]. \]

(b) \( \forall i [\varphi(f^i, g^i) \Rightarrow \varphi(f^{i+1}, g^{i+1})] \).
We assume
\[ \forall x \forall y [f^i(x, y) = g^i(x, y)] \]
\[ \wedge [g^i(x, h(y)) = h(g^i(x, y))], \]
and prove
\[ \forall x \forall y [f^{i+1}(x, y) = g^{i+1}(x, y)] \]
\[ \wedge [g^{i+1}(x, h(y)) = h(g^{i+1}(x, y))]. \]
\[ f^{i+1}(x, y) \equiv \begin{cases} 
1 & \text{if } p(x) \text{ then } y \text{ else } f^i(k(x), y) \\
\text{undefined} & \text{if } x < 1 \text{ f. x. y}
\end{cases} \]
\[ = \begin{cases} 
1 & \text{if } p(x) \text{ then } y \text{ else } g^i(k(x), y) \\
\text{undefined} & \text{if } x < 1 \text{ f. x. y}
\end{cases} \]
\[ = g^{i+1}(x, y). \]
\[ g^{i+1}(x, h(y)) \equiv \begin{cases} 
1 & \text{if } p(x) \text{ then } y \text{ else } g^i(k(x), h^2(y)) \\
\text{undefined} & \text{if } x < 1 \text{ f. x. y}
\end{cases} \]
\[ = \begin{cases} 
1 & \text{if } p(x) \text{ then } y \text{ else } h(g^i(k(x), h(y))) \\
\text{undefined} & \text{if } x < 1 \text{ f. x. y}
\end{cases} \]
\[ = h(g^{i+1}(x, y)). \]

**Proof by complete induction**
Using complete induction we can prove the desired result directly; that is, we prove that
\[ \varphi(f_{P_1}, g_{P_1}) : \forall x \forall y [f_{P_1}(x, y) = g_{P_1}(x, y)] \]

* Where \( F[f^{i-1}] \) is the result of replacing \( f^{i-1} \) for all occurrences of \( F \) in \( JF \).
** Note that this indicates implicitly the need to prove \( \varphi(f^0) \) separately, since for \( i = 0 \) there is no \( j \) s.t. \( j < i \).
holds. This is done by showing first that \( \varphi(f_0, g_0) \) and \( \varphi(f_i, g_i) \) hold, and then that \( \varphi(f_i, g_i) \) holds for all \( i \geq 2 \). (We treat the cases for \( i = 0 \) and \( i = 1 \) separately, since to prove \( \varphi(j_i, g_i) \) we use the induction hypothesis for both \( i - 1 \) and \( i - 2 \).)

(a) \( \varphi(f_0, g_0) \), i.e., \( \forall x \forall y [f_0(x, y) \Rightarrow g_0(x, y)] \).

(b) \( \varphi(f_1, g_1) \), i.e., \( \forall x \forall y [f_1(x, y) \Rightarrow g_1(x, y)] \).

(c) \( \forall i \geq 2 [\varphi(f_i-2, g_i-2) \land \varphi(f_i-1, g_i-1) \Rightarrow \varphi(f_i, g_i)] \)

We assume

\( \forall x \forall y [f_i-2(x, y) \Rightarrow g_i-2(x, y)] \)

and

\( \forall x \forall y [f_i-1(x, y) \Rightarrow g_i-1(x, y)] \)

and deduce

\( \forall x \forall y [f_i(x, y) \Rightarrow g_i(x, y)] \).

THE INDUCTIVE ASSERTIONS METHOD

The most widely used method for proving properties of "flowchart programs" is presently the inductive assertions method, suggested by Floyd\(^a\) and Naur\(^b\). It can be shown that for any proof by inductive assertions, there is a naturally corresponding computational induction proof. We shall illustrate the inductive assertion method and its relation to computational induction on the following simple flowchart program:

We wish first to use the inductive assertions method to show that the above flowchart program over the natural numbers computes the factorial function, i.e., \( z = x! \), whenever it terminates. To do this, we associate a predicate \( Q(x, y_1, y_2) \), called an "inductive assertion," with the point labelled \( \alpha \) in the program, and show that \( Q \) must be true for the values of the variables \( x, y_1, y_2 \) whenever execution of the program reaches point \( \alpha \). Thus, we must show (a) that the assertion holds when point \( \alpha \) is first reached after starting execution (i.e., that \( Q(x, 0, 1) \) holds) and (b) that it remains true when one goes around the loop from \( \alpha \) to \( \alpha \) (i.e., that \( y_1 \leq x \land Q(x, y_1, y_2) \) implies \( y_1 + 1, (y_1 + 1) \cdot y_2 \)). To prove the desired result we finally show (c) that \( z = x! \) follows from the assertion \( Q(x, y_1, y_2) \) when the program terminates (i.e., that \( y_1 = x \land Q(x, y_1, y_2) \) implies \( y_2 = x! \)).

We take \( Q(x, y_1, y_2) \) to be \( y_2 = x! \). Then:

(a) \( Q(x, 0, 1) \) is \( 1 = 0! \).

(b) We assume \( y_1 \neq x \) and \( Q(x, y_1, y_2) \), i.e., \( y_2 = y_1 \). Then \( Q(x, y_1 + 1, (y_1 + 1) \cdot y_2) \) is \( (y_1 + 1) \cdot y_2 = (y_1 + 1)! \), i.e., \( (y_1 + 1) \cdot y_1 = (y_1 + 1)! \).

(c) We assume \( y_1 = x \) and \( Q(x, y_1, y_2) \), i.e., \( y_2 = y_1 \), then \( y_2 = y_1 = x! \) as desired.

To show the relation between this method and computational induction, we must first translate the flowchart program into a recursive program. Following the technique of McCarthy\(^c\), we find that the above

\[ Y_1 = x + 1 \]
\[ Y_2 = Y_1 \cdot Y_2 \]

\[ \text{Halt} \]
The flowchart program is \( f_{P_5}(x, 0, 1) \), i.e., the least fixpoint of the recursive program \( P_5 \) (with \( y_1 = 0 \) and \( y_2 = 1 \)) where

\[
P_5: \quad F(x, y_1, y_2) = \begin{cases} \text{if } y_1 = x \text{ then } y_2 \\ \text{else } F(x, y_1 + 1, (y_1 + 1) \cdot y_2) \end{cases}.
\]

We shall prove by (simple) computational induction that \( f_{P_5}(x, 0, 1) \leq x! \), i.e., the value of \( f_{P_5}(x, 0, 1) \) is \( x! \) whenever \( f_{P_5}(x, 0, 1) \) is defined, which is precisely what the above proof by inductive assertions showed.

We take \( \varphi(F) \) to be the following predicate:

\[
(\forall x, y_1, y_2) \left\{ \begin{array}{l} Q(x, y_1, y_2) := [F(x, y_1, y_2) \leq x!] \end{array} \right\},
\]

where \( Q(x, y_1, y_2) \) is \( y_2 = y_1! \), the induction assertion used before. Obviously \( \varphi(F^i) \) holds. To show that \( \varphi(F^i) \) implies \( \varphi(F^{i+1}) \) for \( i \geq 0 \), we consider two cases: either \( y_1 = x \), in which case the proof follows directly from (c) above, or \( y_1 \neq x \), which follows directly from (b).

By computational induction we therefore have \( \varphi(f_P) \), i.e., \( Q(x, y_1, y_2) := [f_P(x, y_1, y_2) \leq x!] \) for all \( x, y_1, y_2 \). But since \( Q(x, 0, 1) \) is known from (a) to hold, we conclude that \( f_P(x, 0, 1) \leq x! \), as desired.

REFERENCES

1. Z. Manna, S. Ness, J. Vuillemin
   *Inductive methods for proving properties of programs*

2. S. C. Kleene
   *Introduction to meta-mathematics*
   Van Nostrand Princeton New Jersey 1952

3. J. M. Cadou
   *Recursive definitions of partial functions and their computations*
   PhD Thesis Computer Science Dept Stanford University To appear

4. J. Vuillemin
   *Proof techniques for recursive programs*
   PhD Thesis Computer Science Dept Stanford University To appear

5. R. Milner
   *Implementation and applications of Scott's logic for computable functions*

6. W. deBakker, D. Scott
   *A theory of programs*
   Unpublished memo August 1969

7. J. H. Morris
   *Another recursion induction principle*
   CACM Vol 14 No 5 pp 351-354 May 1971

8. R. W. Floyd
   *Assigning meanings to programs*

9. P. Naur
   *Proof of algorithms by general snapshots*
   BIT Vol 6 pp 310-316 1966

10. J. McCarthy
    *Towards a mathematical science of computation*