A Computational Induction Principle

Scott F. Smith*
The Johns Hopkins University

July 11, 1991

Abstract

It is critical to have an induction method for reasoning about recursive programs expressed as fixed points, for otherwise our reasoning ability is severely impaired. The fixed point induction rule developed by deBakker and Scott is one such well known principle.

Here we propose a new induction method, *computational induction*, which is an induction on the computation process. Computational induction is founded on different principles than the fixed point induction principle—it can only be defined in deterministic settings, and it cannot be modeled denotationally. Fixed point induction and computational induction prove many of the same facts; the relation between the two is examined in detail.

1 Introduction

In a theory for reasoning about programs, in particular recursive functions defined via fixed points, there must exist induction principles if interesting results are to be proven. A number of different induction principles have been developed; for a review, see [Man74].

Author's address: Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218 USA. Email: scott@cs.jhu.edu. Fax: 301.338.6134. Phone: 301.338.5299

One method of induction is *well-founded* or noetherian induction, induction over some well-founded ordering. Since inductively defined data structures may be well-ordered, structural induction may be viewed as an instance of well-founded induction. Well-founded induction is useful in proving functions terminate, because if the recursive calls of a function are getting smaller with respect to some well-founded ordering, the function terminates. Well-founded induction is for instance the form of induction used in the Boyer-Moore theorem prover [BM79].

With well-founded induction alone, however, no interesting partial correctness results may be proven. Computations that may not terminate by definition do not have interesting well-founded properties.

This paper considers induction methods that may be incorporated into programming logics to allow partial correctness properties of functions to be proven. In the late 60's and early 70's, many inductive methods for proving partial correctness of fixed points were developed (see [Man74]). The most widespread method today is fixed point induction, developed by deBakker and Scott [dS69]; it is for instance the method used in LCF [GMW79].

Fixed point induction is used to prove properties P about functions f by induction on the finite approximations $\overbrace{f(\ldots f}^{k}(\bot)\ldots)$ to the fixed point $fix(f)$:

PRINCIPLE 1.1 (FIXED POINT INDUCTION)

$$\frac{P(\bot), \forall g.\, P(g) \Rightarrow P(f(g))}{P(fix(f))}$$

where P is an admissible predicate.

If the predicate P is true for all finite approximations to the fixed point, this principle implies P is true of the fixed point. This principle is valid only for certain *admissible* predicates P. It is possible for P to be true at all approximations, but to fail to hold at the limit; for example,

$$f \stackrel{\text{def}}{=} \lambda y.\lambda x.\text{if } x = 0 \text{ then } 0 \text{ else } y(x-1)$$
$$P(y) \stackrel{\text{def}}{=} \exists x.\, y(x) = \bot.$$

$P(\overbrace{f(\ldots f}^{k}(\bot)\ldots))$ is true for all k, but $P(fix(f))$ is false; $\exists x.\, f(x) = \bot$ is thus not an admissible predicate. The largest class of admissible predicates

is undecidable, and the best that can be done is to approximate it with a decidable subclass. This is the major shortcoming of fixed point induction.

Here we define an induction method, *computational induction*, which may be used to prove partial correctness results. A general ordering on computations is defined which reflects the order in which computations are executed. For the terminating computations, this ordering is well-founded, and thus admits induction. Since a computation may be viewed as having a finite number n steps, computational induction amounts to induction on the length of the computation. However, such a principle could not be embedded in a standard programming logic (such as LCF) because it is impossible to reason about how many steps a computation takes in the logic. Here we give a method for encoding the principle of induction on computation length directly into a programming logic, but without the need for explicitly counting steps of a computation. The representation is thus both abstract in that steps need not be counted, but powerful in that many forms of induction on computation length can be expressed.

Although computational induction is founded on different principles than fixed point induction, the two methods prove many of the same facts. To conclude, the relation between the two will be examined.

2 Computational induction

In a deterministic computation system we know how to evaluate computations and know that there is no other way for the evaluation to be carried out, meaning the path of computation is unique. This uniqueness gives rise to an ordering on computations: if we are computing a and require the result of some computation b before the result of a can be completely computed, the computations a and b are linked in a very precise way, because every time a is computed, b will in turn need to be computed. We express this as $a \succ b$ (read "a induces b"). For example, take the function:

$$fact(n) \stackrel{\text{def}}{=} \text{if } n = 0 \text{ then } 1 \text{ else } fact(n-1) * n.$$

Then, $fact(5) \succ fact(5-1)$, because when we evaluate $fact(5)$, we find that $fact(5-1)$ needs to be evaluated to arrive at the result. In general, if $k \neq 0$, $fact(k) \succ fact(k-1)$.

This ordering gives rise to an induction principle for reasoning about computations. An induction principle follows directly if we can define a well-founded ordering, but $\succ$ is not itself well-founded: diverging computations produce infinite chains $a_1 \succ a_2 \succ \ldots$. However, $\succ$ defined over terminating computations *is* well-founded: informally, if it weren't well-founded, there would be an infinite descending chain, but this cannot be the case because infinite chains arise only from nonterminating computations.

Induction over this ordering is called *computational induction*, because it exploits the inductive structure of the actual computation process. A detailed comparison with other methods of induction will be given below, but we point out now that fixed point induction is substantially different from computational induction: fixed point induction is induction on the finite approximants of a fixed point, and this has only an indirect bearing on the actual computation process.

The computational induction principle was originally developed for constructive type theory [Mar82, CAB+86] in [Smi88], but the use of types is not critical. In this paper the method is demonstrated by defining inducement $\succ$ and the computational induction principle for the pure untyped λ-calculus over a call-by-name deterministic evaluator.

For space considerations, rules axiomitizing inducement and computational induction will not be presented; instead, a semantic interpretation will be given, and from this the principles that would make part of an axiom system may be seen. It is not necessary to use any advanced semantic methods to show soundness; in fact, we show that computational induction cannot be shown sound using domains-theoretic models of programs; this partially explains why to date they have not been considered. In this paper an operational interpretation of programs and inducement is given to show the ideas are sound. The mathematical tools are all elementary, and the development is very naturally expressed in purely constructive logic. By observing the method of development for this simple language, we implicitly argue its applicability to languages with more advanced features, including typed and imperative languages. One class of languages it is apparently not very useful for is nondeterministic languages.

First the evaluation relation $a \mapsto b$, meaning a evaluates to b, is defined. Inducement $a \succ b$ is defined in terms of evaluation. We then show how an induction principle may be derived for $\succ$.

The call-by-name λ-calculus consists of expressions of the form $\lambda x.b$, $a(b)$ and x, where a and b are expressions, and x is a variable. λ binds free variables in its body, and $a[b/x]$ denotes substitution of free occurrences of x in a by b. $a = b$ means a and b are identical modulo α-equivalence.

Functions $\lambda x.a$ are *values*, because they are not computed; applications $a(b)$ are *computations*. This general terminology is used because the principles herein apply to languages with other values, such as numbers and pairs; see [Smi88] for this development over such a language.

DEFINITION 2.1 *$a \mapsto v$, meaning a evaluates to v, iff*

CASE *a is $\lambda x.b$:*	$v = a$
CASE *a is $f(c)$:*	$f \mapsto \lambda x.b$ *and* $b[c/x] \mapsto v$

We may make the following simple observations:

LEMMA 2.2
(i) *If $v = \lambda x.b$ then $v \mapsto v$.*
(ii) *If $a \mapsto v$ then $v \mapsto v$.*
(iii) *If $a \mapsto v$ and $a \mapsto v'$ then $v = v'$.*

Terminating computations are those computations that have a value.

DEFINITION 2.3 *$a\downarrow$ iff there exists v such that $a \mapsto v$.*

Inducement is defined in terms of evaluation: $a \succ b$ means that the evaluation of a necessarily entails the evaluations of b. First *direct inducement*, $a \succ_1 b$, is defined. The expressions that are directly induced by a computation a may be read off of the definition of evaluation:

DEFINITION 2.4 *$a \succ_1 b$ iff*

CASE *a is $\lambda x.c$:*	*false*
CASE *a is $f(c)$:*	$b =_\alpha f$ *or* $f \mapsto \lambda x.d$ *and* $b =_\alpha d[c/x]$

The inducement relation is the transitive closure of direct inducement.

DEFINITION 2.5 *$\succ$ is the transitive closure of $\succ_1$.*

Computations induced by terminating computations must themselves terminate:

LEMMA 2.6 *If $a\downarrow$ and $a \succ b$, then $b\downarrow$.*

This fact is useful in showing some computation b induced by a terminates given the fact that a terminates. It is particularly useful for constructive logics of programs, because some termination properties that are provable classically are not provable constructively without an inducement relation. For instance, if $a + 1\downarrow$, $a\downarrow$. This can be proven classically by supposing $a\uparrow$: $a + 1\downarrow$ then may be proven false, giving a contradiction, so $a\downarrow$ by Markov's principle.

The computational induction principle can now be derived. First, the well-founded part of $\succ$ is defined:

DEFINITION 2.7 *$a\Downarrow$ (read "a is founded") is the least property of expressions such that*

> *$a\Downarrow$ iff for all b, $a \succ b$ implies $b\Downarrow$.*

To show an expression is founded it suffices to show it terminates.

LEMMA 2.8 *For all a, $a\downarrow$ implies $a\Downarrow$.*

Since the terminating expressions are well-founded over $\succ$, well-founded induction is valid.

PRINCIPLE 2.9 (COMPUTATIONAL INDUCTION) *Given some arbitrary predicate on expressions P,*

> *if $a\downarrow$ and*
> *$\forall a'.\, a'\downarrow$ and $(\forall a''.\, a' \succ a''$ implies $P(a''))$ implies $P(a')$,*
> *then $P(a)$ is true.*

This principle is the most general computational induction principle; for reasoning about specific sorts of computations, specific induction principles may be derived. For example, for reasoning about fixed point computations $\mathit{fix}(f)$*, we may derive

*fix may be defined as Turing's fixed point combinator, $(\lambda zx.x(zzx))(\lambda zx.x(zzx))$, or added as a primitive.

PRINCIPLE 2.10 (COMPUTATIONAL INDUCTION FOR FIXED POINTS)
Given a two-place predicate P,

$$\begin{array}{l} \textit{if } \mathit{fix}(f)(a)\downarrow \textit{ and} \\ \forall a'.\, \mathit{fix}(f)(a')\downarrow \textit{ and} \\ \qquad (\forall a''.\, \mathit{fix}(f)(a') \succ \mathit{fix}(f)(a'') \textit{ implies } P(\mathit{fix}(f), a'')) \\ \qquad\qquad \textit{implies } P(\mathit{fix}(f), a'), \\ \textit{then } P(\mathit{fix}(f), a) \textit{ is true.} \end{array}$$

2.1 Properties

The inducement relation and computational induction have some properties worth discussing.

If we take $\succ$ to be an observable property of computation, then the collection of observables cannot be modeled by many of the standard methods. In mathematical models of computation such as domains [Sco76, GS90], or functions on sets, computations with the same value are identified, so the structure of $\succ$ is lost. For example, $\mathit{fact}(5)$ and 120 are interpreted by an identical point in a domain, but $\mathit{fact}(5) \succ \mathit{fact}(5-1)$ while $120 \not\succ \mathit{fact}(5-1)$. $\succ$ is a *strongly intensional* property of computations, because it does not respect value-equivalence of computations.

If the evaluation mechanism was not deterministic, then $a \succ b$ would hold only when all possible computations of a led to the computation of b. It is much more difficult to reason about all possible computations than about a single computation path, so $\succ$ is not useful in settings with implicit nondeterminism such as the λ-calculus defined as a rewrite system, allowing redices to be reduced in any order [Bar84].

Computational induction is an abstraction of induction on the number of computation steps: if $a \succ b$, then a must have been computed for more steps than b. Computational induction can then be derived from natural number induction. However, for a programming logic to have the capability for reasoning by induction on step count would mean the whole computation system would have to be reflected in the logic, and this might not be desirable. Computational induction aims to be the most abstract statement of this induction principle.

A logical theory which includes as a component $\succ$ and computational induction may be defined; see [Smi88]. Because of the intensional nature of

$\succ$, it is a challenge to incorporate these ideas in a natural yet usable fashion.

3 Induction methods compared

fixed point induction and the computational induction principle are both general induction principles for computations; it is worthwhile then to compare the two.

First, we consider how the two principles are used in proofs. Consider proving the following predicate on $\mathit{fix}(f)$:

$$\forall n.\, \mathit{fix}(f)(n) \sqsubseteq g(n)$$
$$\text{where } a \sqsubseteq b \stackrel{\text{def}}{=} a\downarrow \Rightarrow a = b.$$

Using fixed point induction, this predicate is admissible, so the rule may be directly applied: if this fact is true at all finite approximations to the fixed point, it will be true at the fixed point. With computational induction, we need to have a terminating computation before applying the rule, so we take n to be arbitrary, and assume $\mathit{fix}(f)(n)\downarrow$; we need to show $\mathit{fix}(f)(n) = g(n)$. Since we make the assumption that $\mathit{fix}(f)(n)$ terminates, computational induction may be applied to prove this statement.

This example illustrates the disparity between how the two principles are used in proofs: if a predicate is admissible, we may immediately apply fixed point induction. To prove the same predicate using computational induction, we first need an assumption that some computation terminates, which entails analyzing the predicate.

The difference in how the two principles are applied may be traced to the differing assumptions about computations they are based on. fixed point induction makes assumptions that computations must be well-behaved; for instance, if the halting problem were solvable, fixed point induction could not be defined [CS88]. Computational induction makes no presumption that the halting problem is not solvable—computational induction could for instance be defined for primitive recursive computation systems. On the other hand, computational induction assumes the existence of a single method of evaluation of computations. fixed point induction needs no assumptions for how the computation is executed—for instance, domains may model fixed point induction.

The biggest weakness of fixed point induction is the problem of admissibility. It is in general undecidable whether a predicate is admissible, so the best we can do is to approximate this class. However, any approximation will miss some admissible predicates, causing some correct proofs to fail. There are some philosophical and methodological shortcomings to admissibility as well. fixed point induction is difficult to accept if we take the view that axioms should be inherently justifiable statements, because admissibility conditions have involved justifications. Computational induction, on the other hand, can be defined with a collection of easily justifiable axioms. As a consequence of this philosophical shortcoming, the admissibility restrictions are problematic methodologically because it is impossible to predict when a predicate will fail the admissibility test. This is an important issue to address in logic design: any time we are unexpectedly surprised to find something cannot be proved, we should curse our lack of insight and not the shortcomings of the logic implementation.

Acknowledgements

Robert Constable and Stuart Allen provided useful comments on this work.

References

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. ACM Monograph series. Academic Press, New York, 1979.

[CAB+86] R. L. Constable, S. F. Allen, H. Bromley, W. R. Cleveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. P. Mendler, P. Panangaden, J. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[CS88] R. L. Constable and S. F. Smith. Computational foundations of basic recursive function theory. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*. IEEE, 1988.

[dS69] J. W. deBakker and D. Scott. A theory of programs. unpublished notes, 1969.

[GMW79] M. J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture notes in Computer Science*. Springer-Verlag, 1979.

[GS90] C. Gunter and D. Scott. Semantic domains. In *Handbook of Theoretical Computer Science*, volume 2, pages 635–674. MIT/Elsevier, 1990.

[Man74] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[Mar82] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[Sco76] D. Scott. Data types as lattices. *SIAM J. Computing*, 5:522–587, 1976.

[Smi88] S. F. Smith. Partial objects in type theory. Technical Report 88-938, Department of Computer Science, Cornell University, August 1988. Ph.D. Thesis.