Generalization Heuristics for Theorems Related to Recursively Defined Functions

S. Kamal Abdali
Computer Research Lab
Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077

Jan Vytopil
BSO-AT
P.O. Box 8348
3503 RH Utrecht
The Netherlands

ABSTRACT

This paper is concerned with the problem of generalizing theorems about recursively defined functions, so as to make these theorems amenable to proof by induction. Some generalization heuristics are presented for certain special forms of theorems about functions specified by certain recursive schemas. The heuristics are based upon the analysis of computational sequences associated with the schemas. If applicable, the heuristics produce generalizations that are guaranteed to be theorems.

INTRODUCTION

This paper deals with the generalization of theorems arising from the analysis of recursive definitions. The theorems of concern here express the properties of functions computed by instances of certain recursive program schemas. To prove these theorems, one usually needs to invoke some form of induction. However, one often encounters cases when induction fails. That is, it turns out that, as originally posed, a given theorem is too weak to be useable in the induction hypothesis for carrying out the induction step. In such cases, it is of interest to find a more general theorem for which induction would succeed.

Manna and Waldinger [8] describe the theorem generalization problem, and mention the heuristic of replacing a constant by a variable and a variable by another variable. Aubin [1, 2] discusses in detail the method of attempting to replace some constant with an expression describing the possible values that the corresponding argument could assume. Another type of heuristics is exemplified by the method of Greif and Waldinger [5], in which the symbolic execution of the program for its first few terms is followed by pattern matching to find a closed form expression which generates the series so obtained. Much has been contributed to the generalization problem by Boyer and Moore [3, 4, 10]. In their work, expressions common to both sides of an equality are replaced with a new variable. This has turned out to be quite a powerful method in actual practice. Still other types of generalization methods are implicit in the work on the synthesis of program loop invariants such as [9, 12, 13], because there is a duality between the problems of finding loop invariants and finding theorem generalizations. Indeed, it is shown in [11] that for certain classes of functions and theorems, these problems are equivalent. Finally, the more recent work on rewrite rule-based methods for proving properties of recursively (equationally) defined functions (e.g., [6]) makes use of unification which can also be looked upon as a kind of generalization.

Two methods of theorem generalization are presented below. These are applicable to two special recursive schemas, and are based on an analysis of the computational sequence of these schemas. The use of these methods requires the existence (and discovery) of some functions that satisfy certain conditions dependent on the schemas.

NOTATION

To express programs and schemas, we use the notation for recursive schemas given in Manna[7, p. 319], with the following convention for letter usage: $u-z$ for individual variables (input, output, and program variables), c for individual constants, $F-H$, $f-h$, and n for function variables, and p, q for predicate variables. Subscripts and primes may also be added to these letters. In general, these symbols stand for vectors rather than single items. Following Moore [10], we distinguish two classes of program variables: "recursion variables", denoted by unsubscripted y or y subscripted with smaller integers, which are used in predicates (loop termination tests), and "accumulators", denoted by y subscripted with larger integers, which are used in forming results but not used in termination tests.

GENERALIZATION SCHEME I

Suppose we are given a recursive schema:

$$z = F(x, c), \text{ where} \qquad (1)$$

$$F(y_1, y_2) \leftarrow \text{if } p(y_1) \text{ then } y_2 \text{ else } F(f(y_1), g(y_1, y_2))$$

Further suppose that for a given interpretation of c, p, f, g the schema computes a known function G. That is, for all x for which $G(x)$ is defined, we would like to prove

$$F(x, c) = G(x) \tag{2}$$

We denote by $f^i(x), i>0$, the expression $f(f(...(f(x))...))$ in which there are i successive applications of f. Further, we denote by $y_1^{(i)}, y_2^{(i)}, i>0$, the values of the variables y_1, y_2 on the i-th iteration of (1), taking $y_1^{(0)}, y_2^{(0)}$ to mean x and c, respectively. If, for a given argument x, the value of $F(x, c)$ is defined, then there must exist an integer k such that each of $p(y_1^{(0)}), p(y_1^{(1)}),..., p(y_1^{(k)})$ is false and $p(y_1^{(k+1)})$ is true, and the following equalities hold:

$$y_1^{(i)} = f^i(x),$$

$$y_2^{(i)} = g(f^{i-1}(x), g(f^{i-2}(x),..., g(f(x), g(x, c))...))$$

for $0 \le i \le k$, and

$$F(x, c) = y_2^{(k+1)}$$

$$= g(f^k(x), g(f^{k-1}(x),..., g(f(x), g(x, c))...))$$

Note that the depth of recursion, that is the value of k, depends only on x and not on c.

Suppose we can find two functions h_1 and h_2 with the property

$$g(u, h_1(v, w)) = h_1(g(u, v), h_2(u, v, w)). \tag{3}$$

If we wish to generalize (2) by replacing c with $h_1(c, z)$, then the final value of y_2 will be

$$g(f^k(x), g(f^{k-1}(x), \ldots, g(x, h_1(c, z)))).$$

Using (3) repeatedly to move h_1 outwards across g, we can rewrite this as

$$h_1(y_2^{(k+1)}, \tag{4}$$
$$h_2(f^k(x), y_2^{(k)}, h_2(f^{k-1}(x), y_2^{(k-1)}, \ldots,$$
$$h_2(f^2(x), y_2^{(2)}, h_2(f(x), g(x, c), h_2(x, c, z))))$$

The first argument of h_1, $y_2^{(k+1)}$, is equal to $F(x, c)$, and the second argument of h_1 is the iteration of h_2 with the first and second arguments having the same values as they have during the evaluation of F. Using the definition of F, we can define a new function H so that $H(x, c, z)$ equals the second argument of h_1 in (4). This H is defined as follows:

$$H(y_1, y_2, y_3) \leftarrow \text{if } p(y_1) \text{ then } y_3$$
$$\text{else } H(f(y_1), g(y_1, y_2), h_2(y_1, y_2, y_3))$$

The generalization of a theorem should be an expression (relation) which

a) we believe is in fact a theorem

b) has the original theorem as an instance

c) is easier to prove.

We propose

$$F(x, h_1(c, z)) = h_1(G(x), H(x, c, z)) \tag{5}$$

as a generalization of (2). The following result states that the condition (a) is satisfied, that is, (5) is indeed true.

Theorem 1: Let f, g, G be some previously defined functions, and F, H be defined by the schemas

$$F(y_1, y_2) \leftarrow \text{if } p(y_1) \text{ then } y_2 \text{ else } F(f(y_1), g(y_1, y_2))$$

$$H(y_1, y_2, y_3) \leftarrow \text{if } p(y_1) \text{ then } y_3$$
$$\text{else } H(f(y_1), g(y_1, y_2), h_2(y_1, y_2, y_3))$$

If

$$F(x, c) = G(x)$$

and there exist functions h_1 and h_2 satisfying

$$g(u, h_1(v, w)) = h_1(g(u, v), h_2(u, v, w))$$

then

$$F(x, h_1(c, z)) = h_1(G(x), H(x, c, z))$$

holds. ||

This theorem can be proved by first showing that under its conditions and definitions, it is the case that

$$F(x, h_1(w, z)) = h_1(F(x, w), H(x, w, z)).$$

The condition

$$g(u, h_1(v, w)) = h_1(g(u, v), h_2(u, v, w))$$

does not guarantee that the original theorem is an instance of the generalized theorem. In general, it is difficult to state how to derive h_1 and h_2 so that the original theorem is an instance of the more general equation (5). Nevertheless, we can state a sufficient condition for it:

Theorem 2: Under the definitions and conditions of Theorem 1, a sufficient condition that $F(x, c) = G(x)$ is an instance of $F(x, h_1(c, z)) = h_1(G(x), H(x, c, z))$ is that there exist an r such that

$$h_1(u, r) = u \text{ and } h_2(u, v, r) = r \tag{6}$$

for all u and v. ||

Although (6) is not a necessary condition, experience suggests that it is a natural and easily satisfied requirement. Also, although at present there is no systematic way of finding h_1 and h_2 (if they exist at all), there are often natural candidates for these functions, such as addition, multiplication or append for h_1 and projection functions for h_2. Note also that while we have only considered theorems of the type $F(x, c) = G(x)$ for simplicity, the method can be used for some more complicated cases such as $F(m(x), k(x)) = G(x)$.

A more restrictive heuristic than that given by Theorems 1 and 2, but one which is particularly simple to use, is described by the following:

Theorem 3: Let f, g, G be some previously defined functions, and F be defined by the schema

$$F(y_1, y_2) \leftarrow \textbf{if } p(y_1) \textbf{ then } y_2 \textbf{ else } F(f(y_1), g(y_1, y_2)).$$

If

$$F(x, c) = G(x) \tag{7}$$

and there exists a function h and a constant r such that

$$g(u, h(v, w)) = h(g(u, v), w)$$

$$h(u, r) = u, \text{ for all } u,$$

then it is the case that

$$F(x, h(c, z)) = h(G(x), z). \tag{8}$$

Furthermore, (7) is an instance of (8). ||

Example 1: Let the function $rev(x)$ be defined by

$$rev(y_1, y_2) \leftarrow \textbf{if } null(y_1) \textbf{ then } y_2$$
$$\textbf{else } rev(cdr(y_1), cons(car(y_1), y_2)).$$

When we try to prove the property

$$rev(x, nil) = reverse(x) \tag{9}$$

by induction on x (*reverse* being the usual LISP function), we find that the induction step cannot be carried through. To apply the generalization method given above, we observe that

$$p(y_1) = null(y_1), \quad f(y_1) = cdr(y_1),$$

$$g(y_1, y_2) = cons(car(y_1), y_2).$$

We now look for a function h satisfying

$$g(u, h(v, w)) = h(g(u, v), w)$$

that is,

$$cons(car(u), h(v, w)) = h(cons(car(u), v), w)$$

and also, for some r and all u,

$$h(u, r) = u.$$

The simple choice $h(u, v) = append(u, v)$ (with *nil* for r) satisfies the above conditions. The generalization of property (9) is thus found to be

$$rev(x, append(nil, z)) = append(reverse(x), z),$$

that is,

$$rev(x, z) = append(reverse(x), z).$$

This property is provable by induction on x with the usual definitions of *append* and *reverse*.

Example 2: Let $F(y_1, y_2, y_3)$ be defined by

$$F(y_1, y_2, y_3) \leftarrow \textbf{if } (y_1 = 0) \textbf{ then } y_3 \textbf{ else}$$
$$F(y_1 \textbf{ div } 2, y_2 * y_2, \textbf{ if } odd(y_1) \textbf{ then } y_2 * y_3 \textbf{ else } y_3)$$

Here, y_1 is the recursion variable, and y_2, y_3 are accumulators. We have

$$f(y_1, y_2) = \langle y_1 \textbf{ div } 2, \ y_2 * y_2 \rangle$$

$$g(y_1, y_2, y_3) = \textbf{if } odd(y_1) \textbf{ then } y_2 * y_3 \textbf{ else } y_3.$$

We look for a function h which satisfies

$$g(u, v, h(w, z)) = h(g(u, v, w), z),$$

that is,

$$\textbf{if } odd(u) \textbf{ then } v * h(w, z) \textbf{ else } h(w, z)$$
$$= h(\textbf{if } odd(u) \textbf{ then } v * w \textbf{ else } w, z).$$

Also, there must exist an r such that for all u, $h(u, r) = u$. To satisfy these requirements, we can simply take $h(u, v) = u * v$ (with r=1). It is now easy to see that the property

$$F(x_1, x_2, 1) = x_2^{x_1}$$

generalizes to the induction-provable property

$$F(x_1, x_2, z) = (x_2^{x_1}) * z.$$

GENERALIZATION SCHEME II

Suppose we have to prove

$$F(x, c, c) = G(x), \textbf{ where} \tag{10}$$
$$F(y_1, y_2, y_3) \leftarrow \textbf{if } (y_1 = y_2) \textbf{ then } y_3$$
$$\textbf{else } F(y_1, f(y_2), g(y_2, y_3))$$

To prove this theorem, we need to carry out induction on y_2. But this would be impossible because the initial value of y_2 is constant. Therefore we must generalize (10) by replacing c with a more general term. In the previous heuristics, we replaced a constant c with a function $h(c, z)$ and then tried to determine the influence of this change of initial value on the final result. It was easy to see how the change of initial value of an accumulator propagated through the entire computation, because of the limited role played by an accumulator in the function evaluation.

It is harder to apply the same strategy to a recursion variable. The initial value of a recursion variable determines the depth of recursion, and at each level of recursion, the value of a recursion variable also affects the final result of computation. Consequently, the change in the initial value of a recursion variable has a much more complicated influence on the computation. So the function h must be chosen more carefully, and in fact our choice now is quite limited. A good strategy would be to replace c with an expression describing all possible values that this recursion variable can take on during the computation of $F(x, c, c')$. Suppose we can derive the values of the recursion variable and the accumulator at the recursion depth z, say $h(z)$ and $H(z)$, respectively. If (10) holds, then

$$F(x, h(z), H(z)) = G(x)$$

would be a good generalization, likely provable by induction on z (that is, on the depth of recursion). To find suitable candidates for $h(z)$ and $H(z)$, we observe that

$$\begin{aligned} F(x, c, c') &= F(x, f(c), g(c, c')), \text{ if } c \neq x, \\ &= F(x, f^2(c), g(f(c), g(c, c'))), \text{ if } f(c) \neq x, \\ &= \cdots = \\ &F(x, f^i(c), g(f^{i-1}(c), g(f^{i-2}(c), \ldots, g(c, c')\ldots))) \quad (11) \\ &\text{if } i \leq i_{\min} = \min\{j \mid f^j(c) = x\}. \end{aligned}$$

Thus, $H(i) = g(f^{i-1}(c), g(f^{i-2}(c), \ldots, g(c, c')))$. On the other hand, for $i \leq i_{\min}$, we also have

$$G(f^i(x)) = F(f^i(x), c, c'))) =$$

$$F(f^i(c), f^i(c), g(f^{i-1}(c), g(f^{i-2}(c), \ldots, g(c, c')))).$$

$$= g(f^{i-1}(c), g(f^{i-2}(c), \ldots, g(c, c')\ldots))$$

So $H(i)$ can simply be replaced by $G(f^i(c))$. We are thus led to the following

Theorem 4: Let F be defined by (10), G be some previously defined function, and let

$$i_{\min} = \min\{j \mid f^j(c) = x\},$$

$$h(z) \leftarrow \text{ if } z = 0 \text{ then } c \text{ else } f(h(z-1)).$$

Then

$$F(x, h(z), G(h(z))) = G(x) \text{ for } 0 \leq z \leq i_{\min} \quad (12)$$

$$\text{iff } F(x, c, c') = G(x).$$

Example 3: Let the definition of a function F be given as

$$F(y_1, y_2, y_3) \leftarrow \text{ if } (y_1 = y_2) \text{ then } y_3$$

$$\text{else } F(y_1, y_2+1, (y_2+1)*y_3)$$

We define h by

$$h(z) \leftarrow \text{ if } z = 0 \text{ then } 0 \text{ else } h(z-1)+1,$$

obtaining, simply, $h(z) = z$. Also in this case we have

$$i_{\min} = \min\{i \mid (i = x)\} = x.$$

So we can generalize

$$F(x, 0, 1) = x!$$

into

$$F(x, z, z!) = x! \quad \text{for } 0 \leq z \leq x.$$

In the above discussion, we have used the simplest case of the theorem in (10). Now suppose that the initial value of y_2 is not a constant but some function $M(x)$. We can derive the value of the accumulator at the recursion depth z if we assume the case

$$M(h(z, x)) = M(x) \quad \text{for all } 0 \leq z \leq i_{\min},$$

where h is defined by

$$h(y_1, y_2) \leftarrow \text{ if } (y_2 = 0) \text{ then } M(y_1)$$

$$\text{else } f(h(y_1, y_2-1)).$$

For example, an $M(x)$ with this property is given by

$$M(y_1) \leftarrow \text{ if } p(y_1) \text{ then } y_1 \text{ else } M(\bar{f}(y_1)),$$

where $\bar{f}$ is the functional inverse of f, that is, $f(\bar{f}(y)) = y$. Now we can write

$$M(x) = M(\bar{f}(x)) = M(\bar{f}^2(x)) = \cdots$$

$$= M(\bar{f}^{(i_{\min})}(x)) = \bar{f}^{(i_{\min})}(x)$$

$$h(x, z) = f^z(\bar{f}^{(i_{\min})}(x)) = \bar{f}^{(i_{\min}-z)}(x),$$

$$\text{for all } 0 \leq z \leq i_{\min}.$$

$$M(h(x, z)) = M(\bar{f}^{(i_{\min}-z)}(x)) = \bar{f}^{(i_{\min})}(x) = M(x).$$

We can therefore generalize

$$F(x, M(x), c') = G(x)$$

into

$$F(x, h(x, z), G(h(x, z)) = G(x), \text{ for } 0 \leq z \leq i_{\min}.$$

Using the relation between f and its inverse $\bar{f}$, the above can be rewritten as

$$F(x, h'(x, z), G(h'(x, z))) = G(x),$$

$$\text{for } 0 \leq z \leq i'_{\min},$$

where

$$h'(x, z) \leftarrow \text{ if } z = 0 \text{ then } x \text{ else } \bar{f}(h'(x)),$$

$$i'_{\min} = \min\{i \mid p(h'(x, i))\}.$$

Example 4: Let the functions M and F be defined as follows:

$M(y_1, y_2) \leftarrow$ **if** $y_1 < y_2$ **then** y_2 **else** $M(y_1, 2*y_2)$

$F(y_1, y_2, y_3, y_4) \leftarrow$ **if** $(y_1 = y_2)$ **then** $\langle y_3, y_4 \rangle$

elseif $(y_3 \geq y_2 \text{ div } 2)$

then $F(y_1, y_2 \text{ div } 2, y_3 - y_2 \text{ div } 2, 2*y_4+1)$

else $F(y_1, y_2 \text{ div } 2, y_3, 2y_4)$.

Since $(2*y_2) \text{ div } 2 = y_2$, we can apply the above method, generalizing

$$F(x_2, M(x_1, x_2), x_1, 0) = \langle x_1 \bmod x_2,\ x_1 \text{ div } x_2 \rangle$$

into

$$F(x_2, h(x_2, z), x_1 \bmod h(x_2, z), x_1 \text{ div } h(x_2, z)) = \langle x_1 \bmod x_2,\ x_1 \text{ div } x_2 \rangle \quad \text{for } 0 \leq z \leq i_{\min},$$

where

$h(y_1, y_2) \leftarrow$ **if** $y_2 = 0$ **then** y_1 **else** $2*h(y_1, y_2-1)$,

that is,

$$h(y_1, y_2) = 2^{y_2} * y_1.$$

CONCLUSION

We have discussed the problem of generalizing theorems about recursively defined functions in such a way that the generalized form of the theorems is more amenable to proof by induction. We have presented, motivated, and illustrated some heuristics for carrying out the generalization for certain patterns of theorems and recursive definitions.

Our heuristics are given in terms of definitional schemas. Given a functional definition, we try to find a matching schema and look for certain auxiliary functions satisfying some conditions dependent on the schema. This seems to be a systematic approach, as opposed to the ad hoc approach of, say, replacing constants by expressions. Furthermore, our generalization heuristics have been derived by analyzing recursive computational sequences. Whenever these heuristics apply, the generalized theorems are true if and only if the original theorems are true. This is not the case with the heuristics currently found in the literature. It may be possible to apply similar generalization methods to other types of definitional schemas, and develop a catalog of heuristics for different patterns of recursive definitions.

REFERENCES

1. Aubin, R.: "Some generalization heuristics in proofs by induction", *Proc. IRIA Colloq. on Proving and Improving Programs*, Arc et Senans, pp. 197–208 (July 1975).
2. Aubin, R.: "Mechanizing structural induction", Ph.D. dissertation, University of Edinburgh (1976).
3. Boyer, R.S. and Moore, J.S.: "Proving theorems about LISP functions", *JACM 22*, 1, pp. 129–144 (January 1975).
4. Boyer, R.S. and Moore, J.S.: *A Computational Logic*, Academic Press, New York (1979).
5. Greif, I. and Waldinger, R.S.: "A more mechanical heuristic approach to program verification", *Proc. Intl. Symp. on Programming*, Paris, pp. 83–90 (April 1974).
6. Huet, G.P. and Huillot, J.-M.: "Proof by induction in equational theories with constructors", *JCSS 25*, 2, pp. 239–266 (Oct. 1982).
7. Manna, Z.: *Mathematical Theory of Computation*, McGraw Hill, New York (1974).
8. Manna, Z. and Waldinger, R.: "Synthesis: dreams → programs", *IEEE Trans. Software Engineering, SE-5*, 4, pp. 294–328 (July 1979).
9. Misra, J.: "Some aspects of the verification of loop computations", *IEEE Trans. Software Engineering, SE-4*, 6, pp. 478–486 (Nov. 1978).
10. Moore, J.S.: "Introducing iteration into the pure LISP theorem prover", *IEEE Trans. Software Engineering, SE-1*, 3, pp. 328–338 (May 1975).
11. Morris, H.J. and Wegbreit, B.: "Subgoal induction", *CACM 20*, 4, pp.209–222 (April 1977).
12. Tamir, M.: "ADI: Automatic derivation of invaraiants", *IEEE Trans. Software Engineering, SE-6*, 1, pp. 40–48 (Jan. 1980).
13. Wegbreit, B.: "The synthesis of loop predicates", *CACM 17*, 2, pp. 102–112 (Feb. 1974).