PRIMITIVE RECURSIVE PROGRAM TRANSFORMATION *

R. S. Boyer, J S. Moore, and R. E. Shostak

ABSTRACT

We describe how to transform certain flowchart programs into equivalent explicit primitive recursive programs. The input/output correctness conditions for the transformed programs are more amenable to proof than the verification conditions for the corresponding flowchart programs. In particular, the transformed correctness conditions can often be verified automatically by the theorem prover developed by Boyer and Moore [1].

KEY WORDS

flowcharts, LISP, program verification, structural induction, theorem proving.

INTRODUCTION

Experiments with the theorem prover developed by R. Boyer and J Moore [1] have shown that structural induction in combination with symbolic evaluation and some generalization heuristics can be used to prove properties of a wide variety of LISP functions completely automatically. The key property of these functions making them amenable to induction is their explicit primitive recursive specification. Roughly speaking, the explicit primitive recursive form produces the effect that when the formula to be proved in the induction conclusion is symbolically evaluated, it assumes the form of the induction hypothesis.

In order to use the theorem prover on flowchart programs, it is necessary to translate the flowcharts into functional form. The easiest approach is that described in McCarthy [3] which produces partial recursive specifications. One is then forced either to extend the theorem prover to cope with a limited class of partial recursive specifications (as Moore does in [4]) or to further transform these specifications (where possible) into explicit primitive recursion. In this paper we are concerned with the second approach.

Of course, not all programs compute primitive recursive functions (for example, programs that compute Ackermann's function or that interpret FORTRAN programs compute partial recursive, but not primitive recursive functions.) Furthermore, it is undecidable whether a function for which a partial recursive definition is given is primitive recursive. Thus, the method described here is not applicable to arbitrary flowchart programs, but only to those fitting certain schemes known to describe primitive recursive functions.

AN EXAMPLE

Our approach is best outlined with an example. Although we have restricted our presentation to the domain of lists and numbers, the general ideas are more broadly applicable.

Figure 1 shows a flowchart program computing the function int(x) that converts a binary number represented as a list of 1's and 0's into an integer. The program scans the input list from left to right. At each position scanned, it doubles the value of an accumulator A and adds the value of the scanned bit. After all bits have been scanned, the value of the accumulator is returned.

Consider the theorem stating that left-shifting a binary number (i.e., tacking a 0 onto the right end) has the effect of doubling that number's value:

```
(1)  int(append(L,list(0))) = 2*int(L),
```

where L is understood to be a universally quantified variable ranging over all lists of 1's and 0's.

The first step in proving the theorem is to convert the flowchart program into functional form. McCarthy [3] has shown that one can do this in a mechanical way for arbitrary flowchart programs by introducing a new recursive function for each tag point. In the above example, one obtains:

```
int(x) = int1(x,0),
```

* This work was supported in part by the Office of Naval Research under Contract No. N00014-75-C-0816, by the Air Force Office of Scientific Research under Contract No. F44620-73-C-0068, and by Xerox Palo Alto Research Center.

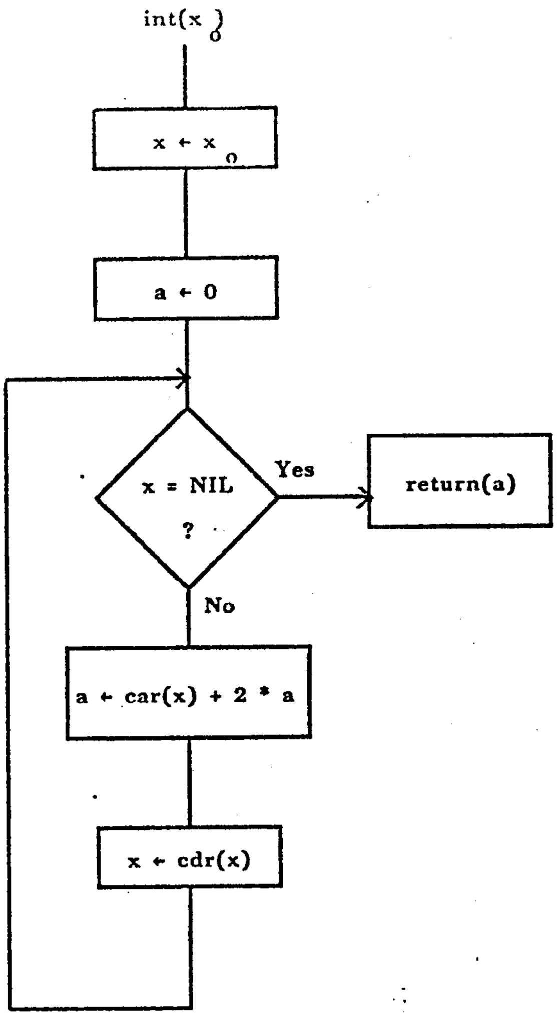

FIGURE 1

where

```
intl(x,a) =
          if x = NIL
             then a;
             else intl(cdr(x),car(x)+2*a).
```

The theorem to be proved can now be stated:

(2) intl(append(L,list(0)),0) = 2*intl(L,0).

One might now be tempted to try to prove (2) using structural induction on L. The basis case, L = NIL, goes through easily because both sides of (2) symbolically evaluate to 0. The induction step, however, does not go through. For that step, one assumes the induction hypothesis (1), and tries to prove:

```
(3) intl(append(cons(B,L),list(0)),0)
                      =
          2*intl(cons(B,L),0),
```

where B is a variable ranging over the set {0, 1}.

Symbolically evaluating both sides of (3) gives:

(4) intl(append(L,list(0)),B) = 2*intl(L,B).

At this point, if all had gone well, we would have been able to invoke the induction hypothesis (2) and been done. But although (4) is similar to (2), it is not quite the same. Specifically, the second argument place of intl is filled with 0 in the one case and with B in the other.

The source of the difficulty is that the form of the definition of intl is not primitive recursive. In particular, the primitive recursive form requires all parameters but the "control" parameter (i.e., the one in the first argument position of intl) to be unmodified in the internal recursive calls. In the definition of intl, however, the second argument is changed from a to car(x)+2*a.

TRANSFORMATION TO PRIMITIVE RECURSION

The solution we propose here is to transform the non-primitive recursive definition of intl into one that is primitive recursive. The transformation works on all functions that are instances of the scheme:

```
f(x,y) = if p(x) then g(x,y)
                 else f(n(x),h(x,y)),
```

where p, g, n, and h are primitive recursive.

The primitive recursive transform of f is f':

```
f'(x,y) =
      g(finalx(x),finaly(rev(seqx(x)),y))
```

where finalx, finaly, and seqx are primitive recursive functions whose definitions are exhibited below:

```
finalx(x) = if p(x) then x
                    else finalx(n(x)),

finaly(xl,y) =
             if xl=NIL
                then y
                else h(car(xl),
                       finaly(cdr(xl),y),

seqx(x) = if p(x) then NIL
                  else cons(x,seqx(n(x))),
```

and rev is the primitive recursive function which reverses a list:

```
rev(x) =
      if x=NIL then NIL
               else append(rev(cdr(x)),
                           list(car(x))).
```

The justification for the theorem:

$$f(X_0,Y_0) = f'(X_0,Y_0)$$

is as follows: Let X_i denote $n^i(X_0)$ and let k be the smallest non-negative integer such that $p(X_k)$, then

$$f(X_0,Y_0) = g(X_k,\ h(X_{k-1},\ h(X_{k-2},\ldots,h(X_0,Y_0)\ldots))). \quad (5)$$

However,

$$\mathrm{finalx}(X_0) = X_k$$

and

$$\mathrm{seqx}(X_0) = (X_0\ X_1\ \ldots\ X_{k-1}).$$

Thus,

$$\mathrm{rev}(\mathrm{seqx}(X_0)) = (X_{k-1}\ X_{k-2}\ \ldots\ X_1\ X_0)$$

so that

$$\mathrm{finaly}(\mathrm{rev}(\mathrm{seqx}(X_0)),Y_0) = h(X_{k-1},h(X_{k-2},\ldots,h(X_0,Y_0)\ldots)).$$

Therefore,

$$f'(X_0,Y_0) = g(X_k,h(X_{k-1},h(X_{k-2},\ldots h(X_0,Y_0)\ldots))),$$

which is just $f(X_0,Y_0)$ by (5).

Informally, seqx constructs a list of the successive values x will take on during the computation of f. This list, in reverse order, is then given to finaly which computes the final value of the "accumulator" y. This value, and that of finalx which is the final value of x, is then given to g to compute the final output of f.

In the special case where p(x) is x=NIL, n(x) is cdr(x), and h(x,y) can be expressed as a function, h', of car(x) and y the transform is simpler:

```
f'(x,y) = g(NIL,finaly(rev(x),y)),
```

where we use h' for h in the definition of finaly. The informal justification of this is that if the final y can be computed only in terms of the car's of the successive values of x, then we need not compute the sequence of x values but merely the sequence of car(x) values. But if p and n are as above this sequence is just x.

It is easy to see that intl is an instance of the scheme described by f, and in fact is an example of the simpler case, since we can let:

```
p(x) = x=NIL
g(x,y) = y
n(x) = cdr(x)
h(x,y) = car(x) + 2*y.
```

Thus, we get:

```
intl'(x,a) = finala(rev(x),a).
```

where finala is:

```
finala(x,a) =
        if x=NIL
           then a
           else
           car(x)+2*finala(cdr(x),a).
```

Given this definition of intl', the example theorem:

```
(2)  intl(append(L,list(0)),0) = 2*intl(L,0),
```

becomes:

```
finala(rev(app(L,list(0))),0)
              =
      2*finala(rev(L),0).
```

While this theorem is somewhat more complicated (syntactically) than (2), all of the functions in it are primitive recursive and it can be proved immediately by the theorem prover described in [1].

DISCUSSION

The idea that flowchart programs can sometimes be replaced by equivalent explicit primitive recursive functions was first mentioned in the 1934 work of R. Peter [5]. To quote from Peter ([5], pp. 69):

"5. It may be seen in a similar way that in general a recursion of the form

$$\varphi(0,a) = \alpha(a),$$

$$\varphi(n+1,a) = \beta(n,a,\varphi(n,\gamma_1(n,a)),\varphi(n,\gamma_2(n,a)),\ldots,\varphi(n,\gamma_k(n,\alpha)))$$

and even a definition of the form

$$\varphi(0,\alpha_1,\ldots,\alpha_r) = \alpha(\alpha_1,\ldots,\alpha_r),$$

$$\varphi(n+1,\alpha_1,\ldots,\alpha_r) =$$

$$
\begin{aligned}
&\beta(n,\alpha_1,\ldots,\alpha_r,\\
&\quad \varphi(n,\gamma_{11}(n,\alpha_1,\ldots,\alpha_r),\ldots,\gamma_{1r}(n,\alpha_1,\ldots,\alpha_r)),\\
&\quad \varphi(n,\gamma_{21}(n,\alpha_1,\ldots,\alpha_r),\ldots,\gamma_{2r}(n,\alpha_1,\ldots,\alpha_r)),\\
&\quad \ldots,\\
&\quad \varphi(n,\gamma_{k1}(n,\alpha_1,\ldots,\alpha_r),\ldots,\gamma_{kr}(n,\alpha_1,\ldots,\alpha_r)))
\end{aligned}
$$

of a function with arbitrarily many argument places does not lead out from the class of primitive recursive functions."

Peter's result for the two argument case is easily seen, since it is just the theorem:

```
f(X0,Y0) = f'(X0,Y0),
```

justified above. This theorem can actually be proved completely automatically by the modified theorem prover described in [4]. Peter's proof of the theorem is somewhat complicated because she carries it out in number theory where a Goedel enumeration method must be used to express the notion of the list of x's used in the computation of f[4].

As noted in the introduction, it is possible to avoid the translation of the partial recursive specifications into primitive recursive ones and still prove many theorems. Moore [4] describes how. Roughly stated, Moore's approach requires two enrichments of the original theorem prover. First, the induction principle must be strengthened so that to prove φ(X,Y) for all X and Y, where Y is used as an accumulator in some function f in φ, one first proves φ(0,Y) for all Y, and then inductively assumes φ(X,e(X+1,Y)) for any expression e, and proves φ(X+1,Y). Moore explains why this is a valid induction principle (also cf. Goodstein [2], pp. 123). The choice of the expression e is left to the theorem proving process, and Moore explains how it can be determined from the *definition of f. The second augmentation required in* [4] is the extension of the generalization heuristic so that accumulator argument positions which initially contain constants can be replaced by expressions containing free variables, allowing the use of the induction method above. This generalization introduces a new function, called the "accumulator function", into the accumulator positions. It turns out that Moore's accumulator function is just our finaly.

The method of translation into primitive recursive form proposed in this paper does not require either the enriched form of induction or the subtle generalization.

We have found that proofs of many theorems involving f' are complicated by the introduction of terms such as finaly(append(x,y),z), due to the expansion of the rev call in f'. However, use of the lemma:

```
finaly(append(x,y),z)
          =
finaly(x,finaly(y,z)),
```

(which can be proved by the theorem prover) allows these terms to be further simplified.

In fact, the use of this equality usually yields the same lemma produced by accumulator generalization and induction in [4].

REFERENCES

[1] Boyer, R.S. and Moore, J Strother. Proving theorems about LISP functions. *J. ACM 22*, 1 (January 1975), pp. 83-105.

[2] Goodstein, R. L. Studies in logic. North-Holland Publishing Company, Amsterdam, 1964.

[3] McCarthy, J. Recursive functions of symbolic functions and their computation by machine. *C. ACM*, 3 (April 1960).

[4] Moore, J Strother. Introducing iteration into the Pure LISP Theorem Prover. *IEEE Transactions on Software Engineering*, SE-1, No. 3 (September 1975), pp. 328-338.

[5] Peter, R. Recursive functions. Academic Press, New York, 1967, pp. 63-69.

[4] This is a good example of why future number theory courses should be taught using Pure LISP as the metalanguage.