Continuation-Based Program Transformation Strategies

MITCHELL WAND

Indiana University, Bloomington, Indiana

ABSTRACT Program transformations often involve the generalization of a function to take additional arguments It is shown that in many cases such an additional variable arises as a representation of the continuation or global context in which the function is evaluated. By considering continuations, local transformation strategies can take advantage of global knowledge The general results are followed by two examples: the α-β tree pruning algorithm and an algorithm for the conversion of a propositional formula to conjunctive normal form

KEY WORDS AND PHRASES. program transformations, continuations, generalization, program manipulation, optimization, recursion, subgoal induction

CR CATEGORIES: 4 12, 4 22, 5.24

1. *Introduction*

The notion of program transformation is a programming paradigm which combines the notion of stepwise refinement [4, 10, 29] with traditional optimization techniques. Under this paradigm, one writes a clear, correct, though possibly inefficient, program, and then transforms it via correctness-preserving transformations into a program which is more efficient although probably less clear. Some of the classes of program transformations are local simplification, partial evaluation (or unfolding), abstraction (or folding), and generalization [28]. The generalization transformation replaces a function by some generalization which may be more amenable to subsequent manipulations. Typical generalizations include the introduction of additional variables or the extension of a function to deal with a list of inputs. Typical strategies for generalization involve pattern matching between compatible but nonidentical goals [1, 2, 28].

In this paper we present a different strategy for generalizations: the use of *continuations*. A continuation is a data structure which represents the future course of a computation. The use of continuations makes the global context of a computation available in the local context, and therefore allows the standard local transformations to use this global information. Our strategy is to obtain tractable closed forms for continuations. By studying the interactions between a function and its continuation, useful transformations can be made.

Many of these transformations are probably familiar to assembly-language programmers, since the machine-level programmer usually has access to a continuation variable. the run-time stack. Despite this fact, we believe that our account of these techniques is useful, since it liberates them from the realm of undocumented "coding tricks" and transports them to the source-language level where they can be used by a wider class of programmers. Furthermore, we will see that such heuristics as "add an accumulator" or "generalize to a list of arguments" may be derived from transformations on continuations, rather than being merely instances of isolated cleverness.

This research was supported in part by the National Science Foundation under Grant MCS75-06678A01

Author's address Computer Science Department, Indiana University, Bloomington, IN 47405

Section 2 of this paper presents our source language (a dialect of Lisp) and our major method of proof, subgoal induction [16]. Section 3 illustrates the notion of a continuation and presents examples of the kind of manipulations that are performed on them. In Section 4 these techniques are applied to the problem of a binary (or tree-structured) recursion pattern. In Section 5 our techniques are applied to two moderate-sized examples: the α-β tree search algorithm and the conversion of propositional formulas to conjunctive normal form. Section 6 presents a summary and conclusion.

2. *Preliminaries*

In this section we describe our language, a syntactically sugared dialect of Lisp, and we describe the verification method we employ, the method of subgoal induction [28].

2.1 THE SOURCE LANGUAGE. We define functions by recursion equations, e.g.,

```
F(x, y) ⇐ if p(x) then y      ,this is a comment
          else a(b(x), F(c(x), y))
```

This style of definition has a long history [14, 15, 21]. We often use sets of simultaneous recursion equations. We usually use uppercase for the names of functions we are defining (e.g., F above) and lowercase for functions which are assumed elementary (e.g., a, b, c, and p above). Such functions are often left unspecified. We refer to these functions as *trivial* and to the ones we define by equations as *serious* [19]. Occasionally we use the arithmetic functions, which we write in infix notation. End-of-line comments are preceded with semicolons. Side effects are not permitted.

We will have occasion to use list processing, for which we use notation adapted from [7]. If x, y, and z are variables, $[x\ y\ z]$ denotes a list whose three elements are the values of x, y, and z. If the value of y is a list of N elements, then the value of $[x\,!\,y]$ is a list of $N + 1$ elements whose first element is x and the remainder of whose elements are those of y. We use *hd* and *tl* to select the first element of a list and its remainder; thus

$$hd([x\,!\,y]) = x, \qquad tl([x\,!\,y]) = y,$$
$$hd([x\ y\ z]) = x, \qquad tl([x\ y\ z]) = [y\ z].$$

Similarly, $[x\ y\,!\,z]$ denotes a list whose first two elements are equal to x and y, and whose remainder is equal to z. [ ] denotes the empty list. The function *append*(x, y) concatenates two lists nondestructively; *append* is associative, so we will sometimes write *append*(x, y, z).

We will also occasionally use temporary functional objects passed as arguments. Such objects, called *closures*, are created using λ-notation. For example, the definition

```
F(x, y) ⇐ MAPHD(λz [x ! z], y)
MAPHD(f, x) ⇐ if x = [ ] then [ ]
                else [f(hd(x)) ! MAPHD(f, tl(x))]
```

passes a functional object as a parameter to *MAPHD*. The value of x inside the closure is the value at the time the closure was created; that is, we use lexical or static scoping [17, 26]. Thus $F(3, [[1]\ [2]\ [3]])$ evaluates to $[[3\ 1]\ [3\ 2]\ [3\ 3]]$. *MAPHD* is a generally useful function which we shall take as primitive. Another useful function is *MAP!* which is defined as

```
MAP!(f, a, x) ⇐ if x = [ ] then a
                 else f(hd(x), MAP!(f, a, tl(x)))
```

where f is a function of two arguments; *MAP!*$(sum, 0, x)$ returns the sum of the elements of x.

2.2 SUBGOAL INDUCTION. Our major method for proving properties about recursion equations is the method of subgoal induction [16], which is an easily understood refinement

of earlier methods [13, 14]. For each serious function $F(x, y)$, we introduce an input-output specification $\psi_F(x, y; z)$ which is a condition on the input parameters x, y and the output value z. The principle of subgoal induction states that if the verification conditions (which we are about to describe) are true, then the function satisfies its specifications, that is, if F halts on input (x, y), then $\psi_F(x, y, F(x, y))$ is true. There is one verification condition for each branch of the equation. The verification condition for each branch has the form $A \,\&\, B \,\&\, C \Rightarrow D$, where A is the condition for taking that branch, B says that all serious function calls on this branch satisfy their specifications, C says that equal arguments to a serious function give equal answers, and D says that the final value on this branch satisfies the desired specification. New variables are introduced throughout to eliminate all occurrences of serious function symbols. For example,

$F(x) \Leftarrow$ **if** $p(x)$ **then** $a(x)$
else $b(F(l(x)), F(r(x)))$

produces the verification conditions

$$\forall x[p(x) \Rightarrow \psi(x; a(x))],$$
$$\forall x \forall z_1 \forall z_2[\sim p(x) \,\&\, \psi(l(x); z_1) \,\&\, \psi(r(x); z_2) \,\&\, [l(x) = r(x) \Rightarrow z_1 = z_2] \Rightarrow \psi(x; b(z_1, z_2))].$$

In the second condition the second and third conjuncts comprise formula B. The "functionality" condition C is not used in this paper but is useful for specifications which would otherwise be too weak to support the induction. Extensions to multiple equations are obvious, as are those to multiple-valued and nondeterministic functions. In our experience this is a powerful and natural method for explaining the correctness of recursive programs. Although subgoal induction is a partial correctness method, it can be extended to prove total correctness by including a performance measure in the specifications, just as the inductive assertion method can be so augmented [11].

3. *Manipulating Continuations*

To illustrate the notion of a continuation, let us consider a function which reverses a list:

Program 3.1

$REV(x) \Leftarrow$ **if** $x = [\,]$ **then** $[\,]$
else $append(REV(tl(x)), [hd(x)])$

If REV is called with a nonnil list x, it proceeds to call REV on $tl(x)$; given the value of $REV(tl(x))$, say w, the resulting answer is $append(w, [hd(x)])$. Another way of expressing this idea is that the function $\lambda w.append(w, [hd(x)])$ is applied to the argument $REV(tl(x))$. The function $\lambda w.append(w, [hd(x)])$ is called the *continuation* [5, 6, 19, 20, 22–24, 26]. We can use this idea to rewrite REV in the so-called continuation-passing style:

Program 3.2

$REV2(x) \Leftarrow REVC(x, \lambda z\ z)$
$REVC(x, \gamma) \Leftarrow$,= $\gamma(REV(x))$ (see footnote 1)
if $x = [\,]$ **then** $\gamma([\,])$
else $REVC(tl(x), \lambda w\ \gamma(append(w, [hd(x)])))$

Here the intended input-output specification is included as a comment.

In order to fix our ideas about subgoal induction as a device for program explanation/verification, let us prove:

PROPOSITION 3.1. $REVC(x, \gamma) = \gamma(REV(x))$.

[1] ,= may be read "is intended to equal."

PROOF. The specification $\psi_{REVC}(x, \gamma; z)$ is $z = \gamma(REV(x))$. The verification conditions are

(i) $(x = [\,]) \Rightarrow \psi_{REVC}(x, \gamma; \gamma([\,]))$ and
(ii) $(x \neq [\,]) \;\&\; \psi_{REVC}(tl(x), \lambda w.\gamma(append(w, [hd(x)])); z) \Rightarrow \psi_{REVC}(x, \gamma; z)$.

Substituting the definition of ψ_{REVC}, we have to show

(i) $(x = [\,]) \Rightarrow (\gamma([\,]) = \gamma(REV(x)))$,
(ii) $(x \neq [\,]) \;\&\; (z = (\lambda w.\gamma(append(w, [hd(x)])))(REV(tl(x)))) \Rightarrow (z = \gamma(REV(x)))$.

Verification condition (i) follows immediately from the definition of *REV*. Condition (ii) is proved as follows:

$$\begin{aligned}
&(x \neq [\,]) \;\&\; (z = (\lambda w.(append(w, [hd(x)])))(REV(tl(x)))) \\
&\quad \Rightarrow (x \neq [\,]) \;\&\; (z = \gamma(append(REV(tl(x)), [hd(x)]))) \quad (\beta\text{-reduction}^2) \\
&\quad \Rightarrow z = \gamma(REV(x)) \quad (\text{definition of } REV, \text{ using } x \neq [\,]).
\end{aligned}$$

□

The use of subgoal induction lets us prove the correctness of *REVC* essentially "line by line," referring to the definition of *REV* and doing some simple manipulations. Henceforth, proofs of this sort will be left to the reader; the relevant input-output specification will be included as a comment.[3] Similarly, it is evident that if *REV* terminates (which it always does), then so does *REVC*. This may be proved by considering the performance specification

$\psi_{REVC}(x, \gamma; z) \equiv$ if y occurs as a first argument to *REVC* during the computation of $REVC(x, \gamma)$, then y occurs as a first argument to *REV* during the computation of $REV(x)$.

A brief consideration of the usual operational semantics of recursion equations (using, say, call-by-value) reveals that this specification is inconsistent with nontermination. Similar arguments throughout will be left to the diligent reader.

Let us now make another observation about the operational semantics of *REVC*: In the computation of $REV2(x)$, every value of γ supplied to *REVC* is of the form $\lambda v.append(v, a)$ for some a. To prove this, we observe that $\lambda z.z$ is of this form (with $a = [\,]$), and if $\gamma = \lambda v.append(v, a)$, then

$$\begin{aligned}
&\lambda w.\gamma(append(w, [hd(x)])) \\
&\quad = \lambda w.((\lambda v.append(v, a))(append(w, [hd(x)]))) && (\text{definition of } \gamma) \\
&\quad = \lambda w.append(append(w, [hd(x)]), a) && (\beta\text{-reduction}) \\
&\quad = \lambda w.append(w, append([hd(x)], a)) && (\text{associativity}) \\
&\quad = \lambda w.append(w, [hd(x) \,!\, a]) && (\text{fact about } append).
\end{aligned}$$

Hence, instead of carrying around the function γ, we can carry around the list a which represents it. Instead of writing $\gamma([\,])$, we can write $append([\,], a)$ or just a. This gives us

Program 3.3

$REV3(x) \Leftarrow REVC3(x, [\,])$
$REVC3(x, a) \Leftarrow \quad ,= append(REV(x), a)$
  if $x = [\,]$ **then** a
    else $REVC3(tl(x), [hd(x) \,!\, a])$

which is just the usual "iterative reverse with accumulator." This leads us to our key observation: *An accumulator is often just a data structure representing a continuation function.*[4] Data structures representing functions of some restricted type are widespread. A closure is of course a data structure; an association list is a representation of a function

[2] The operation of β-reduction replaces an expression of the form $(\lambda v\, t)a$ by t, with a substituted for all free occurrences of v

[3] Just as one should include one's invariants as comments.

[4] It would be nice to turn this observation into a theorem by replacing the word "often" by "always" That, unfortunately, would require a formal definition of an "accumulator," which is quite beyond the scope of this paper

```
F(x) ⇐
  if p(x) then a(x)
    else b(F(c(x)), d(x))
```

where b is associative, with right identity 1_b, is replaced by

```
F'(x) ⇐ FC(x, 1_b)
FC(x, γ) ⇐      ,= b(F(x), γ)
  if p(x) then b(a(x), γ)
    else FC(c(x), b(d(x), γ))
```

FIG 1 Replacement of associative continuation builder by accumulator

from keys to values when the keys are atoms;[5] and the run-time stack is a machine-level representation of the top-level continuation [e.g., 19, 26]. Indeed, the identifier environment in which a program is run can be any data structure which can be used to map keys to values; even the form of the keys can be made arbitrary by preprocessing (see [22]).

It is worthwhile to state Proposition 3.1 as a general transformation (Figure 1).

PROPOSITION 3.2. *In Figure* 1, $F'(x) = F(x)$.

PROOF. By subgoal induction on FC. The interesting case is $\sim p(x)$:

$$\begin{aligned} FC(x, \gamma) &= FC(c(x), b(d(x), \gamma)) \\ &= b(F(c(x)), b(d(x), \gamma)) && \text{(induction hypothesis)} \\ &= b(b(F(c(x)), d(x)), \gamma) && \text{(associativity of } b\text{)} \\ &= b(F(x), \gamma). && \text{(definition of } F\text{).} \end{aligned}$$

□

This transformation is well known; what is new in our discussion is the relationship between the accumulator and the continuation.

Similar transformations for the nonassociative case have been considered by Strong [25]. Let us take, for an example, McCarthy's 91-function:

Program 3 4

```
F(x) ⇐
  if x > 100 then x − 10 else F(F(x + 11))
```

In continuation form this becomes

Program 3 5

```
F2(x) ⇐ F2-C(x, λz z)
F2-C(x, γ) ⇐      , = γ(F(x))
  if x > 100 then γ(x − 10)
    else F2-C(x + 11, λw γ(F(w)))
```

Here again, we can obtain a closed form for the continuation: It is always of the form $\lambda w.F(F(F\cdots(w)))$ for some number of F's. Hence we can represent the continuation by a counter. Unfortunately, it is more difficult to simulate $\gamma(x - 10)$ in this representation. To do this, we write a special function *F3-SEND* which simulates functional application for the given representation of the continuation:

Program 3 6

```
F3(x) ⇐ F3-C(x, 0)
F3-C(x, i) ⇐      ,= F^(i)(F(x))
  if x > 100 then F3-SEND(x − 10, i)
    else F3-C(x + 11, i + 1)
F3-SEND(v, i) ⇐      ,= F^(i)(v)
  if i = 0 then v      ;the identity continuation
    else F3-C(v, i − 1)
```

(Compare [12, Problem 3-5]).

[5] If more is known about the function, then a more finely optimized representation may be used, e g , a binary search tree

When less is known about the continuation builders, then the representation of the continuation will perforce be less compact. In previous work we have considered the case where nothing whatsoever is known [27]. Another case, also studied by Strong [25], is that of a single continuation builder with a parameter:

Program 3 7

$F(x) \Leftarrow$
  if $p(x)$ **then** $a(x)$
    else if $q(x)$ **then** $F(b(x))$
    else $c(d(x), F(e(x)))$

Introducing continuations, we get

Program 3 8

$F2(x) \Leftarrow F2\text{-}C(x, \lambda z\ z)$
$F2\text{-}C(x, \gamma) \Leftarrow \quad ,= \gamma(F(x))$
  if $p(x)$ **then** $\gamma(a(x))$
    else if $q(x)$ **then** $F2\text{-}C(b(x), \gamma)$
    else $F2\text{-}C(e(x), \lambda w\ \gamma(c(d(x), w)))$

Here again, we know a closed form for the continuation:

$$\lambda w.c(a_1, c(a_2, \ldots, c(a_n, w))).$$

We can represent this continuation by $[a_n\ a_{n-1} \cdots a_1]$, giving

Program 3 9

$F3(x) \Leftarrow F3\text{-}C(x, [\,])$
$F3\text{-}C(x, \gamma) \Leftarrow$
  if $p(x)$ **then** $F3\text{-}SEND(a(x), \gamma)$
    else if $q(x)$ **then** $F3\text{-}C(b(x), \gamma)$
    else $F3\text{-}C(e(x), [d(x)\ ' \gamma])$
$F3\text{-}SEND(v, \gamma) \Leftarrow$
  if $\gamma = [\,]$ **then** v
    else $F3\text{-}SEND(c(hd(\gamma), v), tl(\gamma))$

This is the well-known technique of replacing a single return address by a data stack [10, 25].[6] We chose to reverse the a_i's in the representation of the continuation so that the transformations of building and decomposing the continuations would be easily implemented in our list processing primitives.

The correctness proof for (3.9) is not quite so straightforward as that for (3.8). In (3.8) we had the specification $F2\text{-}C(x, \gamma) = \gamma(F(x))$. In (3 9) γ is no longer a function but is rather its representation. Hence the corresponding specification is $F3\text{-}C(x, \gamma) = F3\text{-}SEND(F(x), \gamma)$.

PROPOSITION 3.3. *For all x and γ, $F3\text{-}C(x, \gamma) = F3\text{-}SEND(F(x), \gamma)$.*

PROOF. By subgoal induction on F. If $p(x)$, then

$$\begin{aligned} F3\text{-}C(x, \gamma) &= F3\text{-}SEND(a(x), \gamma) && \text{(definition of } F3\text{-}C\text{)} \\ &= F3\text{-}SEND(F(x), \gamma) && \text{(definition of } F\text{).} \end{aligned}$$

Otherwise, if $q(x)$, then

$$\begin{aligned} F3\text{-}C(x, \gamma) &= F3\text{-}C(b(x), \gamma) && \text{(definition of } F3\text{-}C\text{)} \\ &= F3\text{-}SEND(F(b(x)), \gamma) && \text{(induction hypothesis)} \\ &= F3\text{-}SEND(F(x), \gamma) && \text{(definition of } F\text{).} \end{aligned}$$

[6] Note that in the original definition of F there were two recursive calls on F. The first of these, however, was tail-recursive, and so corresponds to the identity transformation on continuations Similarly, tail-recursive lines could be added to any of our examples without requiring global modifications

Otherwise,

$$\begin{aligned} F3\text{-}C(x, \gamma) &= F3\text{-}C(e(x), [d(x) \,!\, \gamma]) && \text{(definition of } F3\text{-}C\text{)} \\ &= F3\text{-}SEND(F(e(x)), [d(x) \,!\, \gamma]) && \text{(induction hypothesis)} \\ &= F3\text{-}SEND(c(d(x), F(e(x))), \gamma) && \text{(definition of } F3\text{-}SEND\text{)} \\ &= F3\text{-}SEND(F(x), \gamma) && \text{(definition of } F\text{).} \end{aligned}$$ □

PROPOSITION 3.4 $F3(x) = F(x)$.

PROOF. $F3(x) = F3\text{-}C(x, [\,]) = F3\text{-}SEND(F(x), [\,]) = F(x)$. □

By viewing these transformations as data-structure optimizations on continuations, we can consider other cases. We can use interactions between different continuation builders to find closed forms for continuations. The use of associativity was a primitive example of this, and another example appears in Section 5.2. Alternatively, we can use local continuations to represent parts of the global state, relying on the run-time stack to do the rest. Let us consider the following example:

```
G(x) ⇐
  if p(x) then a(x)
    else if q(x) then b(G(c(x)))
    else d(G(l(x)), G(r(x)))
```

where b distributes through d, i.e.,

$$b(d(x, y)) = d(b(x), b(y)).$$

We can then introduce a counter for the b-builder:

```
G2(x) ⇐ G2-C(x, 0)
G2-C(x, i) ⇐      ,= b^(i)(G(x))
  if p(x) then G2-SEND(a(x), i)
    else if q(x) then G2-C(c(x), i + 1)
    else ?
```

The desired value in the place of the question mark is $b^{(i)}(d(G(l(x)), G(r(x))))$. By the distributive law, this is equal to $d(b^{(i)}(G(l(x))), b^{(i)}(G(r(x)))) = d(G2\text{-}C(l(x), i), G2\text{-}C(r(x), i))$. Thus we get

```
G2-C(x, i) ⇐      ,= b^(i)(G(x))
  if p(x) then G2-SEND(a(x), i)
    else if q(x) then G2-C(c(x), i + 1)
    else d(G2-C(l(x), i), G2-C(r(x), i))
G2-SEND(v, i) ⇐      ,= b^(i)(v)
  if i = 0 then v
    else G2-SEND(b(v), i − 1)
```

Although the conditions for this transformation look somewhat restrictive, they arise in both the large examples we will do later. The result of this transformation is not yet in iterative form [15], as the previous examples all were, but it has only a single line with a nontrivial continuation builder, and is therefore in a good form for further transformations. In Section 4 we consider some transformations applicable to binary recursion patterns such as this.

4. *Nonlinear Recursions*

The previous example showed how the user can retain control over the presentation of portions of the continuation while allowing the run-time stack to handle the "messier" portions, such as return addresses. In this section we will consider the case of nonlinear

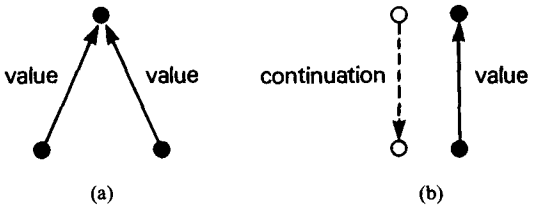

FIG 2 Attribute-grammar patterns for recursion equations (a) nonlinear recursions, (b) linear recursion in continuation-passing style

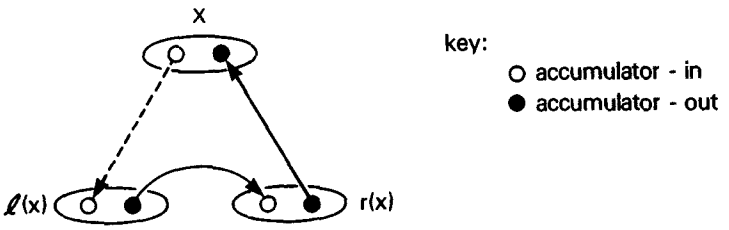

FIG 3 Attribute-grammar pattern for nonlinear recursion

recursion patterns in more detail. In particular we will examine the following program:

Program 4 1

$F(x) \Leftarrow$
  if $p(x)$ **then** $a(x)$
    else $b(F(l(x)), F(r(x)))$

where b is an associative operation with an identity. We will see how one may use accumulators here without falling back on assignment, and how a nonpessimizing interpreter will in fact mimic rather clever hand-compiled code. Last, we will see why in this situation one is led to consider the generalization "take a list of arguments," or

$$\lambda l.MAP!(b, 1_b, MAP(F, l))$$

(a quite mysterious generalization!), and why this is usually wrong.

We have written program 4.1 to suggest the traversal of some binary tree [10], where the trivial functions l and r select the left and right subtrees. In terms of attribute grammars [8] F computes a synthesized attribute, as do the original examples of Section 3. A continuation, however, is a summary of the tree above the current node, and is therefore an inherited attribute. (See Figure 2.)

From our programming knowledge, we can observe that we would like to implement program 4.1 with an accumulator. The attribute-grammar pattern for this implementation is shown in Figure 3.

How can we write this information flow pattern as a term in a recursion equation? If we set $G(x, v) = b(v, F(x))$, then

$$\begin{aligned} G(x, v) &= b(v, b(F(l(x)), F(r(x)))) \\ &= b(b(v, F(l(x))), F(r(x))) && \text{(associativity of } b\text{)} \\ &= b(G(l(x), v), F(r(x))) && \text{(definition of } G\text{)} \\ &= G(r(x), G(l(x), v)) && \text{(definition of } G\text{).} \end{aligned}$$

Thus we have

Program 4 2

$F2(x) \Leftarrow G2(x, 1_b)$,= $F(x)$
$G2(x, v) \Leftarrow$,= $b(v, F(x))$
  if $p(x)$ **then** $b(v, a(x))$
    else $G2(r(x), G2(l(x), v))$

where 1_b is a left identity for b. The code generated for the last line of program 4.2 should go roughly as follows [22]:

```
        STACK r(x)        ;the standard trick for inorder
        x ← l(x)            traversals (See Ref. [10].)
        STACK [TAG1]      ,the return address
        GOTO G2
TAG1:   v ← ANSWER        ;retrieve result of inner call
        x ← UNSTACK       ,thus retrieving r(x)
        GOTO G2           ,tail recurse
```

Similar code may be generated for any term of the form $F(t_1, \ldots, t_n, u)$, where u is the first nontrivial subterm.

Similar behavior may be obtained by interpreted code (in a static binding environment) if EVLIS is modified so that it does not store the calling environment across the evaluation of the last argument. Then all that is stacked across evaluation of u in $F(t_1, \ldots, t_n, u)$ is the list of values of $t_1, \ldots, t_n$; thus the behavior of the compiled code above is mimicked. We refer to this phenomenon as evlis tail recursion.[7]

We can still do better by hand, however, since we can take advantage (as an interpreter cannot) of the fact that there is only a single function symbol which is stacked, namely, $G2$. In other words, we have a single continuation builder with a parameter, and so we can apply the transformation of the preceding section to get our third version:

Program 4.3

$F3(x) \Leftarrow G3(x, 1_b, [\,])$
$G3(x, v, \gamma) \Leftarrow$
  if $p(x)$ **then** $SEND3(b(v, a(x)), \gamma)$
    else $G3(l(x), v, [r(x) \,!\, \gamma])$
$SEND3(v, \gamma) \Leftarrow$
  if $\gamma = [\,]$ **then** v
    else $G3(hd(\gamma), v, tl(\gamma))$

The transformation from program 4.2 to program 4.3 differs from the transformation from program 3.7 to program 3.9 only in regard to the **else** line (setting $q(x) =$ false in program 3.7). Let us check this verification condition.

PROPOSITION 4.1. $G3(x, v, \gamma) = SEND3(G2(x, v), \gamma)$.

PROOF. We proceed by subgoal induction on $G2$. The only verification condition which differs significantly from those in Proposition 3.3 is the last, the case where $p(x)$ is false. The verification condition is

$$\sim p(x) \;\&\; \psi_{G2}(l(x), v; z_1) \;\&\; \psi_{G2}(r(x), z_1; z_2) \Rightarrow \psi_{G2}(x, v; z_2).$$

(Here we have ignored the superfluous functionality condition.) Assuming the hypotheses, we calculate:

$$\begin{aligned}
G3(x, v, \gamma) &= G3(l(x), v, [r(x) \,!\, \gamma]) && \text{(definition of } G3\text{)} \\
&= SEND3(G2(l(x), v), [r(x) \,!\, \gamma]) && \text{(induction hypothesis)} \\
&= G3(r(x), G2(l(x), v), \gamma) && \text{(definition of } SEND3\text{)} \\
&= SEND3(G2(r(x), G2(l(x), v)), \gamma) && \text{(induction hypothesis)} \\
&= SEND3(G2(x, v), \gamma) && \text{(definition of } G2\text{).}
\end{aligned}$$

□

PROPOSITION 4.2. $F3(x) = F(x)$.

PROOF. $F3(x) = G3(x, 1_b, [\,]) = SEND3(G2(x, 1_b), [\,]) = G2(x, 1_b) = F2(x) = F(x)$. □

[7] The idea of evlis tail recursion was discovered jointly with D.P Friedman and D S Wise Like tail recursion, evlis tail recursion can dramatically improve the space performance of many programs

Program 4.3 is in iterative form [15] and so does not require use of the run-time stack. Note that *G*3 and *SEND*3 are mutually tail recursive; this corresponds to the use of goto's in the iterative code. Since we have derived this program from a more structured one by provably correct transformations, we conclude that here the use of the goto is permissible [10]. While this code uses the goto, the correctness proof bears little, if any, resemblance to a correctness proof for program 4.3 using the inductive assertion method. The structure of the proof followed not the "object code" (program 4.3) but rather the "source code" (program 4.2), and is hence much easier.

Some other observations about program 4.3 concern the sole appearance of b. Its second argument is $a(x)$, where $p(x)$ is true. If b is a user function which is complicated but known to be associative, it may be optimized to take advantage of this fact. (We use this later.) Furthermore, the values of v are all of the form $b(b(b(1_b, a_1), a_2), \ldots, a_n)$. Hence this version is most suitable for b's which prefer to associate to the left. For b's which are cheaper to associate to the right (like *append*), one can get a similar program based on $G(x, v) ;= b(F(x), v)$. Furthermore, we can take advantage of the way the values of v are built. If b is, say, conjunction, and some $a(x)$ comes out false, then we can exit immediately, rather than calling *SEND*3 again. Note that this may cause $F3$ to converge when the original F diverged!

We next consider some transformations which we might use to restore some "structure" to the "goto" program (program 4.3). We first observe that for any t and u,

$$SEND3(v, [t\ u\ !\ \gamma]) = G3(t, v, [u\ !\ \gamma])$$

by unwinding the definition of *SEND*3. We unwind the call to *G*3 in the definition of *SEND*3 to get

```
SEND4(v, γ) ⇐      ,= SEND3(v, γ)
  if γ = [ ] then v
     else if p(hd(γ)) then SEND4(b(v, a(hd(γ))), tl(γ))
     else G3(l(hd(γ)), v, [r(hd(γ)) ! tl(γ)])
```

Using the previous identity we get our next version

Program 4 4

```
F4(x) ⇐ SEND4(1_b, [x])
SEND4(v, γ) ⇐      ,= SEND3(v, γ)
  if γ = [ ] then v
     else if p(hd(γ)) then SEND4(b(v, a(hd(γ))), tl(γ))
     else SEND4(v, [l(hd(γ)) r(hd(γ)) ! tl(γ)])
```

Note that *SEND*4 is just a while-loop and may therefore be claimed to be "more structured" than program 4.3, which requires unrestricted gotos.

PROPOSITION 4.3. $SEND4(v, \gamma) = SEND3(v, \gamma)$.

PROOF. By subgoal induction on *SEND*4. □

PROPOSITION 4.4. $F4(x) = F(x)$.

PROOF. $F4(x) = SEND4(1_b, [x]) = SEND3(1_b, [x]) = G3(x, 1_b, [\ ]) = F3(x) = F(x)$. □

We may further transform program 4.4 by observing that *SEND*4 treats its first argument as an accumulator and is therefore the target of a transformation like that of Figure 1. We invert the transformation to get

Program 4 5

```
F5(x) ⇐ SEND5([x])
SEND5(γ) ⇐
  if γ = [ ] then 1_b
     else if p(hd(γ)) then b(a(hd(γ)), SEND5(tl(γ)))
     else SEND5([l(hd(γ)) r(hd(γ)) ! tl(γ)])
```

PROPOSITION 4.5. $SEND4(v, \gamma) = b(v, SEND5(\gamma))$.

PROOF. By induction on *SEND5*. □

PROPOSITION 4.6. $F5(x) = F(x)$.

PROOF. $F5(x) = SEND5([x]) = b(1_b, SEND5([x])) = SEND4(1_b, [x]) = F4(x) = F(x)$. □

SEND5 is the generalization of F to take a *list* of arguments instead of a single argument—a generalization which seems a priori nonobvious. If, however, one set about to optimize *SEND5*, one would first introduce an accumulator for the associative builder on the $p(hd(\gamma))$ line. One might then observe that after the final call to *SEND5*, γ is guaranteed to be unequal to [ ], and therefore introduce an inner loop to avoid the $\gamma =$ [ ] test. One might then spread $hd(\gamma)$ in a separate register. The result of these changes would be nothing other than program 4.3.

One last note is in order. If one has an equivalence relation R on the data objects, and one is willing to weaken Proposition 4.6 to "$F5(x)$ and $F(x)$ are equivalent modulo R," then b need not be associative: One need only that $b(x, b(y, z))$ and $b(b(x, y), z)$ are equivalent modulo R.

5 *Examples*

In this section we shall do two fair-sized examples: α-β tree searching and the conversion of formulas of propositional logic to conjunctive normal form.

5.1 α-β TREE SEARCHING. We wish to do minimax searching of a game tree. We assume that every node is either a leaf node with a numeric value or else has associated with it a nonnull list of sons. We have two functions, F^+ and F^-, which seek to maximize and minimize the values associated with a node:

Program 5.1.1

```
F⁺(x) ⇐
  if leaf?(x) then value(x)
    else max(MAPHD(F⁻, sons(x)))
F⁻(x) ⇐
  if leaf?(x) then value(x)
    else min(MAPHD(F⁺, sons(x)))
```

Our first step is to eliminate the instances of *MAPHD* by standard transformations [2]:

Program 5.1.2

```
F2⁺(x) ⇐
  if leaf?(x) then value(x)
    else G2⁺(sons(x))
G2⁺(l) ⇐
  if tl(l) = [ ] then F2⁻(hd(l))
    else max(F⁻(hd(l)), G2⁺(tl(l)))
F2⁻(x) ⇐
  if leaf?(x) then value(x)
    else G2⁻(sons(x))
G2⁻(l) ⇐
  if tl(l) = [ ] then F2⁺(hd(l))
    else min(F2⁺(hd(l)), G2⁻(tl(l)))
```

We notice that we have two associative continuation builders, max and min. Furthermore, under reasonable conditions they commute with each other:

PROPOSITION 5.1.1

(*i*) *If* $\alpha \leq \beta$, *then* $max(\alpha, min(\beta, v)) = min(\beta, max(\alpha, v))$.

(*ii*) *If* $\alpha \leq \beta$, *then* $\alpha \leq max(\alpha, min(\beta, v)) \leq \beta$.

(*iii*) $max(v, min(x, y)) = min(max(v, x), max(v, y))$.

(*iv*) $min(v, max(x, y)) = max(min(v, x), min(v, y))$.

We therefore consider the *conditional* generalization

$F3^+(\alpha, \beta, x) \Leftarrow$,= $max(\alpha, min(\beta, F2^+(x)))$ **if** $\alpha \leq \beta$
  if $leaf?(x)$ **then** $max(\alpha, min(\beta, value(x)))$
    else $G3^+(\alpha, \beta, sons(x))$
$G3^+(\alpha, \beta, l) \Leftarrow$,= $max(\alpha, min(\beta, G2^+(l)))$ **if** $\alpha \leq \beta$
  if $tl(l) = [\ ]$ **then** $F3^-(\alpha, \beta, hd(l))$
    else $max(\alpha, min(\beta, max(F2^-(hd(l)), G2^+(tl(l)))))$

and the simultaneous symmetric generalization for the pair of minimizing functions. Using the associative and distributive laws (Proposition 5.1.1), we obtain

$$\begin{aligned} &\max(\alpha, \min(\beta, \max(F2^-(hd(l)), G2^+(tl(l))))) \\ &\quad = \max(\max(\alpha, \min(\beta, F2^-(hd(l)))), \max(\alpha, \min(\beta, G2^+(tl(l))))) \\ &\quad = \max(F3^-(\alpha, \beta, hd(l)), \max(\alpha, \min(\beta, G2^+(tl(l))))). \end{aligned}$$

Introducing a help function for this continuation of $F3^-$, we obtain

$G4^+(\alpha, \beta, l) \Leftarrow$,= $G3^+(\alpha, \beta, l)$
  if $tl(l) = [\ ]$ **then** $F3^-(\alpha, \beta, hd(l))$
    else $H4^+(\alpha, \beta, tl(l), F3^-(\alpha, \beta, hd(l)))$
$H4^+(\alpha, \beta, l, v) \Leftarrow max(v, max(\alpha, min(\beta, G2^+(l))))$

But $\alpha \leq v \leq \beta$, so

$$\begin{aligned} &\max(v, \max(\alpha, \min(\beta, G2^+(l)))) \\ &\quad = \textbf{if } v \geq \beta \textbf{ then } v \textbf{ else } \max(v, \min(\beta, G2^+(l))). \end{aligned}$$

Substituting this for $H4$, we reach

$F5^+(\alpha, \beta, x) \Leftarrow$
  if $leaf?(x)$ **then** $max(\alpha, min(\beta, value(x)))$
    else $G5^+(\alpha, \beta, sons(x))$
$G5^+(\alpha, \beta, l) \Leftarrow$
  if $tl(l) = [\ ]$ **then** $F5^-(\alpha, \beta, hd(l))$
    else $H5^+(\alpha, \beta, tl(l), F5^-(\alpha, \beta, hd(l)))$
$H5^+(\alpha, \beta, l, v) \Leftarrow$
  if $v \geq \beta$ **then** v **else** $G5^+(v, \beta, l)$

and the simultaneous corresponding minimizing functions. Here $H5$ performs cutoff.

This example is interesting because α-β cutoff is usually justified by referring to a picture of the global tree, rather than by a program transformation argument [9]. Indeed, a possible criticism of recursive procedures is that they induce a premature contraction of the state space; in this example, the choice of state space inherent in the original version (program 5.1.1) would seem to preclude the introduction of insights derived from the global state. Here, however, the continuation variable supplies precisely what is needed: a window allowing the global state to be included in the local state. (In retrospect this might have been expected, since the true state space of a set of recursive procedures includes the run-time stack as a ghost variable. It is surprising nonetheless that this state is accessible in comprehensible form at the source level [18].)

5.2 CONJUNCTIVE NORMAL FORM. Given a formula of the propositional calculus involving implication, negation, binary conjunction, and binary disjunction, we are asked to produce a formula logically equivalent to the original and which is in conjunctive

normal form (c.n.f.) [3], that is, of the form

$$C_1 \,\&\, C_2 \,\&\, \cdots\, C_n,$$

where each C_i (called a *clause*) is of the form

$$l_1 \vee \cdots \vee l_{m_i},$$

where each l_j (called a *literal*) is a propositional variable or its negation. Traditionally, such a formula is produced by first removing implications and then driving negations inward, via the rules:

$$\begin{aligned} A \Rightarrow B &\to \sim A \vee B, \\ \sim(A \,\&\, B) &\to \sim A \vee \sim B, \\ \sim(A \vee B) &\to A \,\&\, \sim B, \\ \sim\sim A &\to A. \end{aligned}$$

Then the distributive law

$$(A \,\&\, B) \vee C \to (A \vee C) \,\&\, (B \vee C)$$

and its variants are applied until the task is completed.

Removing implications and driving negations inward are straightforward programming tasks; distribution seems somewhat harder, since it is not immediately clear how one can do better than say, "Search the entire formula for a possible rewrite." Furthermore, one must switch at some point from binary operations to n-ary operations; this conversion is not discussed in the standard sketch of the algorithm. We will therefore concentrate on the distribution task.

We will assume the input formulas are given in some abstract form, but we will use a concrete representation for the output formulas: A clause is a list of literals, and a formula is a list of clauses.

We make the following try at a program:[8]

Program 5 2 1

```
CNF1(x) ⇐
  if literal?(x) then [[x]]
  else if conj?(x) then append(CNF1(op1(x)), CNF1(op2(x)))
  else if disj?(x) then DISTR(CNF1(op1(x)), CNF1(op2(x)))
```

where *DISTR* is a function (yet to be written) which performs approximately as follows:

$$DISTR(C_1 \,\&\, \cdots \,\&\, C_n,\ D_1 \,\&\, \cdots \,\&\, D_m) = (C_1 \vee D_1) \,\&\, (C_1 \vee D_2) \,\&\, \cdots \,\&\, (C_n \vee D_m),$$

taking two formulas in c.n.f. and returning the $n \times m$ clauses in the c.n.f. of their disjunction. Such a function looks like it would take some care to code correctly; furthermore, it would seem to involve considerable recopying of lists with the attendant inefficiency. We observe, however, that the specifications for *DISTR* require that it be associative (at least up to logical equivalence), with [[ ]], the false formula, as an identity, so we can use the transformation of the last section, getting

Program 5 2 2

```
CNF2(x) ⇐ G2(x, [[ ]], [ ])
G2(x, v, γ) ⇐      ,= S2(DISTR(v, CNF1(x)), γ)
  if literal?(x) then S2(DISTR(v, [[x]]), γ)
     else if conj?(x) then S2(DISTR(v, append(CNF1(op1(x)), CNF1(op2(x)))), γ)
     else if disj?(x) then G2(op1(x), v, [op2(x) ! γ])
S2(v, γ) ⇐
  if γ = [ ] then v
     else G2(hd(γ), v, tl(γ))
```

[8] Here we use "**else if** *disj*?(*x*) **then** . ." rather than "**else**. . ." to remind the reader of the conditions under which the final clause is executed

Here we regard $CNF1$ as a "trivial" function—though we must, of course, eliminate it before we are done!

We now note that $DISTR(X, append(Y, Z)) \equiv \text{cnf}(X \vee (Y \,\&\, Z)) \equiv \text{cnf}((X \vee Y) \,\&\, (X \vee Z)) \equiv append(DISTR(X, Y), DISTR(X, Z))$, where $\equiv$ denotes logical equivalence among representations of c.n.f. formulas.[9] So the conjunction branch may be simplified to

```
S2(append(DISTR(v, CNF1(op1(x))),
          DISTR(v, CNF1(op2(x)))),
   γ)
```

We can achieve a fold if we can prove:

PROPOSITION 5.2.1. $S2(append(x, y), \gamma) = append(S2(x, \gamma), S2(y, \gamma))$.

PROOF. We consider

```
S2'(v, γ) ⇐      ,= S2(v, γ)
  if γ = [ ] then v
    else S2'(DISTR(v, CNF1(hd(γ))), tl(γ))
```

$S2'(v, \gamma) = S2(v, \gamma)$ follows by subgoal induction on $S2'$ using the fact that $G2(x, v, \gamma) = S2(DISTR(v, CNF1(x)), \gamma)$. We then prove the proposition $(\forall x, y)[S2'(append(x, y), \gamma) = append(S2'(x, \gamma), S2'(y, \gamma))]$ by induction on γ. If $\gamma = [\ ]$, then both sides equal $append(x, y)$. If $\gamma \neq [\ ]$, then

$$
\begin{aligned}
&S2'(append(x, y), \gamma)\\
&\quad = S2'(DISTR(append(x, y), CNF1(hd(\gamma))), tl(\gamma))\\
&\quad = S2'(append(DISTR(x, CNF1(hd(\gamma))), DISTR(y, CNF1(hd(\gamma)))), tl(\gamma))\\
&\qquad\qquad \text{(by the argument above)}\\
&\quad = append(S2'(DISTR(x, CNF1(hd(\gamma))), tl(\gamma)),\\
&\qquad\quad S2'(DISTR(y, CNF1(hd(\gamma))), tl(\gamma))) \quad \text{(induction hypothesis)}\\
&\quad = append(S2'(x, \gamma), S2'(y, \gamma)).
\end{aligned}
$$

□

By Proposition 5.2 1, we can simplify the conjunction branch to

$$
\begin{aligned}
&append(S2(DISTR(v, CNF1(op1(x))), \gamma),\\
&\qquad S2(DISTR(v, CNF1(op2(x))), \gamma))
\end{aligned}
$$

which we fold with $G2$ to obtain

Program 5 2 3

```
CNF3(x) ⇐ G3(x, [[ ]], [ ])
G3(x, v, γ) ⇐      ,= S3(DISTR(v, CNF1(x)), γ)
  if literal?(x) then S3(DISTR(v, [[x]]), γ)
    else if conj?(x) then append(G3(op1(x), v, γ), G3(op2(x), v, γ))
    else if disj?(x) then G3(op1(x), v, [op2(x) ' γ])
S3(v, γ) ⇐
  if γ = [ ] then v
    else G3(hd(γ), v, tl(γ))
```

Before proceeding further it is worth examining program 5.2.3 to see if we can make some intuitive sense out of its components. We first notice that in the only call to $DISTR$, the second argument is a formula consisting of a single literal:

$$DISTR(C_1 \,\&\, \cdots \,\&\, C_n, [[x]]) = (C_1 \vee x) \,\&\, \cdots \,\&\, (C_n \vee x).$$

This simplification would surely make $DISTR$ much easier to write, but we can observe something stronger. Every value of v is a formula consisting of a single clause. This is true because v starts out as a formula of one clause ([[ ]]), and the only operation which

[9] Here cnf(X) means "some c n f of X" rather than the c n f produced by some particular program

changes v is $DISTR(v, [[x]])$, which preserves this property. Therefore we can represent v as a clause instead of a formula; then $DISTR(C_1, [[x]])$ is logically equivalent to $[x \mid C_1]$. We therefore rename v to *lits*, since it accumulates literals into a clause. We also rename γ to *rest*, since it stores the rest of x which will be processed later. These changes yield

Program 5.2.4

```
CNF4(x) ⇐ G4(x, [ ], [ ])
G4(x, lits, rest) ⇐      ,= S4(DISTR([lits], CNF1(x)), rest)
  if literal?(x) then S4([x ' lits], rest)
    else if conj?(x) then append(G4(op1(x), lits, rest),
                                 G4(op2(x), lits, rest))
    else if disj?(x) then G4(op1(x), lits, [op2(x) ' rest])
S4(lits, rest) ⇐
  if rest = [ ] then [lits]
    else G4(hd(rest), lits, tl(rest))
```

With program 5.2.4 we once again have a single associative continuation builder, *append*, so we can eliminate the recursion entirely by applying the transformation of Section 4 once again. Here we must be a little careful, since we have two mutually recursive functions instead of one. We name the accumulator *clauses*. The result is

Program 5.2.5

```
CNF5(x) = G5(x, [ ], [ ], [ ], [ ])
G5(x, lits, rest, clauses, γ) ⇐
    ,= SEND5(append(clauses, G4(x, lits, rest)), γ)
  if literal?(x) then S5([x ! lits], rest, clauses, γ)
    else if conj?(x) then G5(op1(x), lits, rest, clauses, [op2(x), lits, rest ' γ])
    else if disj?(x) then G5(op1(x), lits, [op2(x) ' rest], clauses, γ)
S5(lits, rest, clauses, γ) ⇐
  if rest = [ ] then SEND5([lits ' clauses], γ)
    else G5(hd(rest), lits, tl(rest), clauses, γ)
SEND5(clauses, γ) ⇐
  if γ = [ ] then clauses
    else G5(hd(γ), hd(tl(γ)), hd(tl(tl(γ))), clauses, tl(tl(tl(γ))))
```

This is not quite the last word, however. We can prevent the inclusion of tautologous clauses by replacing the literal branch of $G5$ in program 5.2.5 by

```
if literal?(x) then
  if tautology?(x, lits) then SEND5(clauses, γ)
    else S5([x ' lits], rest, clauses, γ)
```

and we can similarly eliminate subsumed clauses [3] by changing the first branch of $S5$ to be

```
if rest = [ ] then
  if subsumed?(lits, clauses) then SEND5(clauses, γ)
                              else SEND5([lits ' clauses], γ)
```

Either of these changes would have required an *escape* or similar major surgery on any of the previous versions, since the necessary variables were hidden from the local state space.

6. *Conclusions*

We have presented a technique for producing generalizations of functions as a step in the program transformation process. In this technique one generalizes by adding a variable corresponding to the continuation or global context in which the local computation takes place. One can often express the continuation or portions of it in closed form; one can then use standard local simplification techniques. This allows recovery from the premature

contraction of the state space that sometimes accompanies recursive programming styles. If a closed form is known, then data structure optimizations may be used to obtain an efficient representation of the continuation.

This generalization technique is complementary to the ones suggested in [2], which seem to be aimed toward finding uses for their abstraction rule to eliminate redundant function calls. Our local manipulations are, of course, similar to theirs; our contribution is an account of the origin of the additional variables in the generalization.

We have given two moderate-sized examples of the technique, involving repeated applications of these transformations. Other examples where these ideas have been applied include: the equal-tips problem, backtrack programs (e.g., [4, pp. 63–66]), finding all factors of a clause [3, p. 80], and implementing semantic resolution [3, Ch. 6].

ACKNOWLEDGMENTS. D.P. Friedman planted the seed for the ideas expressed here when he challenged us to come up with a meaning for the term "data structure continuation." We also benefited from discussions with S.C. Shapiro.

REFERENCES

1 AUBIN, R Some generalization heuristics in proofs by induction Proc IRIA Symp on Proving and Improving Programs, Arc-et-Senans, France, 1975, pp 197–208
2 BURSTALL, R M, AND DARLINGTON, J A transformation system for developing recursive programs *J ACM 24*, 1 (Jan 1977), 44–67
3 CHANG, C C, AND LEE, R C T *Symbolic Logic and Mechanical Theorem-Proving* Academic Press, New York and London, 1973
4 DAHL, O J, DIJKSTRA, E W, AND HOARE, C A R *Structured Programming* Academic Press, London and New York, 1972
5 FISCHER, M J Lambda-calculus schemata Proc ACM Conf on Proving Assertions about Programs SIGPLAN Notices (ACM) 7, 1 (Jan 1972), 104–109
6 HEWITT, C, BISHOP, P, AND STEIGER, R A universal modular ACTOR formalism for artificial intelligence Proc Third International Joint Conf on Artificial Intelligence, 1973, pp 235–245
7 HEWITT, C E, AND SMITH, B Towards a programming apprentice *IEEE Trans Software Eng SE-1* (1975), 26–45
8 KNUTH, D E Semantics of context-free languages *Math Systems Theory 2* (1968), 127–145, correction, *5* (1971), 95–96
9 KNUTH, D E An analysis of alpha-beta pruning Tech Rep STAN-CS-74-441, Computer Sci Dept, Stanford U, Stanford, Calif, 1974
10 KNUTH, D E Structured programming with **go to** statements *Computing Surveys 6*, 4 (Dec 1974), 261–301
11 LUCKHAM, D C, AND SUZUKI, N Proof of termination within a weak logic of programs. *Acta Informatica 8* (1977), 21–36.
12 MANNA, Z. *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974
13 MANNA, Z, AND PNUELI, A Formalization of properties of functional programs *J ACM 17*, 3 (July 1970), 555–569
14 MANNA, Z, AND VUILLEMIN, J Fixpoint approach to the theory of computation. *Comm ACM 15*, 7 (July 1972), 528–536
15 MCCARTHY, J Towards a mathematical science of computation In *Information Processing 62*, C M Popplewell, Ed, North-Holland, Amsterdam, 1963, pp 21–28
16 MORRIS, J H, AND WEGBREIT, B Subgoal induction *Comm ACM 20*, 4 (April 1977), 209–222
17 MOSES, J The function of FUNCTION in LISP *SIGSAM Bulletin (ACM) 15* (July 1970), 13–27.
18 PARTSCH, H, AND PEPPER, P A family of rules for recursion removal *Inform Proc Letters 5* (1976), 174–177
19 REYNOLDS, J C Definitional interpreters for higher-order programming languages Proc ACM Annual Conf, Boston, Mass, 1972, pp 717–740
20 REYNOLDS, J C On the relation between direct and continuation semantics Proc 2nd Colloq on Automata, Languages, and Programming, J Loeckx, Ed, *Lecture Notes in Computer Science, Vol 14*, Springer-Verlag, New York, 1974, pp 141–156
21 SCOTT, D, AND STRACHEY, C Toward a mathematical semantics for computer languages In *Computers and Automata*, J Fox, Ed, Wiley, New York, 1972, pp 19–46
22 STEELE, G L JR LAMBDA The ultimate declarative AI Memo 379, M I T, Cambridge, Mass, 1976
23 STEELE, G L JR, AND SUSSMAN, G J LAMBDA The ultimate imperative AI Memo 353, M I T, Cambridge, Mass, 1976
24 STRACHEY, C, AND WADSWORTH, C P Continuations A mathematical semantics for handling full jumps Tech Monog PRG-11, Oxford U Computing Laboratory, Oxford U, Oxford, England, 1974

25. STRONG, H.R Translating recursion equations into flow charts *J Comptr Syst Sci. 5* (1971), 254–285
26 SUSSMAN, G.J , AND STEELE, G.L , JR SCHEME An interpreter for extended lambda calculus AI Memo 349, M.I.T., Cambridge, Mass., 1975
27 WAND, M., AND FRIEDMAN, D P Compiling lambda expressions using continuations and factorizations *J Comptr Languages 3* (1978), 241–263.
28. WEGBREIT, B Goal-directed program transformation. *IEEE Trans Software Eng. SE-2* (1976), 69–79.
29 WIRTH, N Program development by stepwise refinement *Comm. ACM 14*, 4 (April 1971), 221–227

RECEIVED JUNE 1977, REVISED FEBRUARY 1979