# CONTINUATION-BASED PROGRAM TRANSFORMATION STRATEGIES

by

Mitchell Wand
Computer Science Department
Indiana University
Bloomington, Indiana 47401

TECHNICAL REPORT NO. 61

CONTINUATION-BASED PROGRAM TRANSFORMATION STRATEGIES

MITCHELL WAND

MARCH, 1977
REVISED: JANUARY, 1978

to appear:
J. ACM

Research reported herein was supported in part by the National Science Foundation under grant number MCS75-06678A01.

# Continuation-Based Program Transformation Strategies

Mitchell Wand*

## Abstract

Program transformations often involve the generalization of a function to take additional arguments. This paper shows 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 $\alpha$-$\beta$ 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

4.22, 4.12, 5.24

Research reported herein was supported in part by the National Science Foundation under grant number MCS75-06678A1

---

*Author's address: Computer Science Department, Indiana University, Bloomington, Indiana 47401.

## 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 non-identical 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.

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 $\alpha$-$\beta$ 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 will describe our language, a syntactically-sugared dialect of LISP, and we will describe the verification method we will 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 [12, 15, 21]. We will often use sets of simultaneous recursion equations. We will usually use upper-case for the names of functions we are defining (e.g. F above) and lower-case for functions which are assumed elementary (e.g. a, b, c, and p above). Such functions will often be left unspecified. We refer to these functions as <u>trivial</u> and to the ones we define by equations as <u>serious</u> [19]. Occasionally we will 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
tℓ([x ! y]) = y
hd([x y z]) = x
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 non-destructively; 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 <u>closures</u>, 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! which is defined as

$$\text{MAP!}(f, a, x) \Leftarrow \underline{\text{if}}\ x = [\,]\ \underline{\text{then}}\ a$$
$$\underline{\text{else}}\ f(hd(x),\ \text{MAP!}(f, a, t\ell(x)))$$

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 [14, 15]. 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, symbols. For example,

$$F(x) \Leftarrow \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ a(x)\ \underline{\text{else}}\ b(F(\ell(x)), F(r(x)))$$

produces the verification conditions

$$\forall x[p(x) \Rightarrow \psi(x; a(x))]$$

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

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 [10, 22]. In our experience, this method is a powerful and natural method for explaining the correctness of recursive programs. While 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:

$$\text{REV}(x) \Leftarrow \underline{\text{if}}\ x = [\,]\ \underline{\text{then}}\ [\,] \quad \underline{\text{else}}\ \text{append}(\text{REV}(t\ell(x)), [\text{hd}(x)]) \tag{3.1}$$

If REV is called with a non-nil list x, it proceeds to call REV on $t\ell(x)$; given the value of $\text{REV}(t\ell(x))$, say w, the resulting answer is append(w, [hd(x)]). Another way of expressing this idea is that the function $\lambda w.\text{append}(w, [\text{hd}(x)])$ is applied to the argument $\text{REV}(t\ell(x))$. The function $\lambda w.\text{append}(w, [\text{hd}(x)])$ is called the <u>continuation</u> [5, 6, 19, 20, 22, 23, 24, 26]. We can use this idea to rewrite REV in the so-called continuation-passing style:

$$\left.\begin{array}{l} \text{REV2}(x) \Leftarrow \text{REVC}(x, \lambda z.z) \\ \text{REVC}(x,\gamma) \Leftarrow \qquad ;= \gamma(\text{REV}(x))^{*} \\ \qquad \underline{\text{if}}\ x = [\,]\ \underline{\text{then}}\ \gamma([\,]) \\ \qquad\quad \underline{\text{else}}\ \text{REVC}(t\ell(x), \lambda w.\gamma(\text{append}(w, [\text{hd}(x)]))) \end{array}\right\} \tag{3.2}$$

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:

<u>Proposition 3.1</u> $\text{REVC}(x,\gamma) = \gamma(\text{REV}(x))$.

<u>Proof</u>: The specification $\Psi_{\text{REVC}}(x, \gamma; z)$ is $z = \gamma(\text{REV}(x))$.

---

* ;= may be read "is intended to equal"

The verification conditions are:

(i) $(x=[]) \Rightarrow \psi_{REVC}(x, \gamma; \gamma([]))$

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

$\Rightarrow \psi_{REVC}(x, \gamma; z)$

Substituting the definition of $\psi_{REVC}$, we have to show

(i) $(x=[]) \Rightarrow (\gamma([]) = \gamma(REV(x)))$

(ii) $(x\neq[]) \& (z = (\lambda w.\gamma(append(w, [hd(x)])))(REV(t\ell(x))))$

$\Rightarrow (z = \gamma(REV(x)))$

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

$(x\neq[]) \& (z = (\lambda w.\lambda(append(w, [hd(x)])))(REV(t\ell(x)))$

$\Rightarrow (x\neq[]) \& (z = \gamma(append(REV(t\ell(x)), [hd(x)]))$ ($\beta$-reduction)*

$\Rightarrow z = \gamma(REV(x))$ (def. of REV, using $x\neq[]$).

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.** 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($x$).

---

*The operation of $\beta$-reduction replaced an expression of the form $(\lambda v.t)a$ by t, with a substituted for all free occurrences of v.

**Just as one should include one's invariants as comments.

A brief consideration of the usual operational semantics of recursion equations (using, say, call-by-value) reveals that this specification is inconsistent with non-termination. 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 $\gamma$ 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

```
 λw.γ(append(w, [hd(x)]))
 = λw.((λv.append(v, a))(append(w, [hd(x)])))   (def. of γ)
 = λw.append(append(w, [hd(x)]), a)              (β - reduction)
 = λw.append(w,append([hd(x)], a))               (associativity)
 = λw.append(w, [hd(x) ! a])                     (fact about
                                                  append)
```

Hence, instead of carrying around the function $\gamma$, 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:

```
    REV3(x) <= REVC3(x, [])
    REVC3(x, a) <=    ; = append(REV(x), a)
        if x = [] then a                          (3.3)
           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.*(*) 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 from keys to values when the keys are atoms(**); 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 pre-processing (see [22]).

---

(*) 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.

(**) If more is known about the function, then a more finely optimized representation may be used e.g. a binary search tree.

```
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))
```

Figure 3.1 Replacement of associative continuation-builder by accumulator

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

Proposition 3.2 In Figure 3.1, F'(x) = F(x).

Proof: By subgoal induction on FC. The interesting case is ~p(x):

| | | |
|---|---|---|
| FC(x, γ) | = FC(c(x), b(d(x), γ)) | |
| | = b(F(c(x)), b(d(x), γ)) | (IH) |
| | = b(b(F(c(x)), d(x)), γ) | (associativity of b) |
| | = b(F(x), γ) | (definition of F) ∎ |

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

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

```
F(x) <=                                                  (3.4)
      if  x>100 then x-10 else F(F(x+11))
```

In continuation form this becomes:

```
F2(x) <= F2-C(x, λz.z)
F2-C(x, γ) <= ;=γ(F(x))                                  (3.5)
   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 λw.F(F(F....(w))) for some number of F's. Hence we can represent the continuation by a counter. Unfortunately, it is more difficult to simulate γ(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:

```
F3(x) <= F3-C(x, 0)
F3-C(x, i) <=              ;=F^(i)(F(x))                 (3.6)
   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 [13, Problem 3-5]).

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:

```
F(x) <=                                              (3.7)
  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

```
F2(x) <= F2-C(x, λz.z)
F2-C(x, γ) <=              ;=γ(F(x))                 (3.8)
  if p(x) then γ(a(x))
     else if q(x) then F2-C(b(x), γ)
     else F2-C(e(x), λw.γ(c(d(x), w)))
```

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

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

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

```
F3(x) <= F3-C(x, [])
F3-C(x, γ) <=
  if p(x) then F3-SEND(a(x), γ)
     else if q(x) then F3-C(b(x), γ)                 (3.9)
     else F3-C(e(x), [d(x) ! γ])
F3-SEND (v, γ) <=
  if γ=[] then v
     else F3-SEND(c(hd(γ),v), tℓ(γ))
```

This is the well-known technique of replacing a single return address by a data stack [10,25]*. 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 straight forward as that for (3.8). In (3.8) we had the specification F2-C(x, $\gamma$) $\gamma$(F(x)). In (3.9), $\gamma$ is no longer a function but is rather its representation. Hence the corresponding specification is F3-C(x, $\gamma$)=F3-SEND(F(x), $\gamma$).

<u>Proposition 3.3</u> For all x and $\gamma$, F3-C(x, $\gamma$)= F3-SEND(F(x), $\gamma$).

<u>Proof</u>: By subgoal induction on F. If p(x), then

F3-C(x, $\gamma$) = F3-SEND(a(x), $\gamma$) (def. of F3-C)

= F3-SEND(F(x), $\gamma$) (def. of F).

Otherwise, if q(x), then

F3-C(x, $\gamma$) = F3-C(b(x), $\gamma$) (def. of F3-C)

= F3-SEND(F(b(x)), $\gamma$) (IH)

= F3-SEND(F(x), $\gamma$) (def. of F).

(*) Note that in the original definition of F there were <u>two</u> 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,

$$\begin{aligned}
\text{F3-C}(x, \gamma) &= \text{F3-C}(e(x), [d(x) \mathbin{!} \gamma]) && \text{(def. of F3-C)} \\
&= \text{F3-SEND}(F(e(x)), [d(x) \mathbin{!} \gamma]) && \text{(IH)} \\
&= \text{F3-SEND}(c(d(x), F(e(x))), \gamma) && \text{(def. of F3-SEND)} \\
&= \text{F3-SEND}(F(x), \gamma) && \text{(def. of F).} \blacksquare
\end{aligned}$$

Proposition 3.4  F3(x) = F(x)

Proof: F3(x) = F3-C(x, []) = F3-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 $b^{(i)}(d(G(\ell(x)), G(r(x)))$. By the distributive law, this is equal to $d(b^{(i)}(G(\ell(x))), b^{(i)}(G(r(x)))) = d(G2\text{-}C(\ell(x), i), G2\text{-}C(r(x), i))$

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 [12], as the previous examples all were, but it has only a single line with a non-trivial continuation-builder, and is therefore in a good form for further transformations. In the next section 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 representation 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 recursion patterns in more detail. In particular, we will examine the recursion equation

$$\begin{array}{l} F(x) \Leftarrow \\ \quad \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ a(x) \\ \qquad \underline{\text{else}}\ b(F(\ell(x)),\ F(r(x))) \end{array} \tag{4.1}$$

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 non-pessimizing 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

$$\lambda\ell.\mathrm{MAP!}(b,\ 1_b,\ \mathrm{MAP}(F,\ \ell))$$

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

We have written the equation (4.1) to suggest the traversal of some binary tree [10], where the trivial functions $\ell$ 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 the last section. A continuation, however, is a summary of the tree above the current node, and therefore an inherited attribute. (See Figure 4.1) is a summary of the tree above the current node, and is therefore an inherited attribute. (See Figure 4.1)

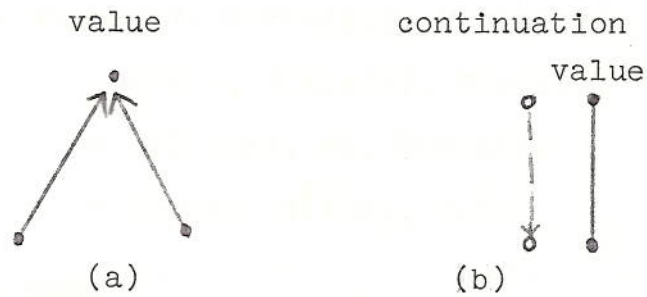


Figure 4.1 Attribute-grammar patterns for recursion equations.
(a) Nonlinear recursions.
(b) Linear recursion in continuation-passing style.

From our programming knowledge, we can observe that we would like to implement equation 4.1 with an accumulator. The attribute-grammar pattern for this implementation is shown in Figure 4.2.

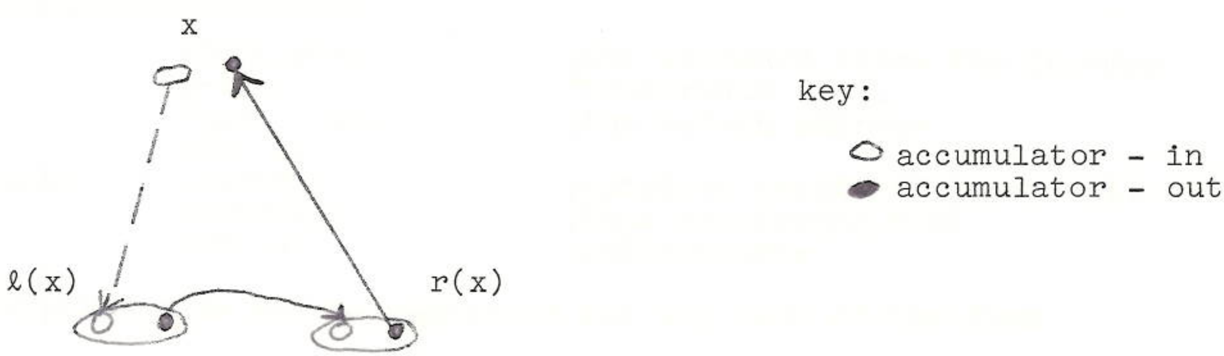


Figure 4.2 Attribute-grammar pattern for nonlinear recursion.

How can we write this information flow pattern as a term in a recursion equation? If we set $G(x, v) = b(v, F(x))$, compiled code above. We refer to this phenomenon as evlis-tail-recursion.(*)

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:

$$\left.\begin{array}{l}
F3(x) \Leftarrow G3(x,\ 1_b,\ [\,]) \\
G3(x,\ v,\ \gamma) \Leftarrow \\
\quad \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ SEND3(b(v,\ a(x)),\ \gamma) \\
\qquad \underline{\text{else}}\ G3(\ell(x),\ v,\ [r(x)\ !\ \gamma]) \\
SEND3(v,\ \gamma) \Leftarrow \\
\qquad \underline{\text{if}}\ \gamma = [\,]\ \underline{\text{then}}\ v \\
\qquad\quad \underline{\text{else}}\ G3(hd(\gamma),\ v,\ t\ell(\gamma))
\end{array}\right\} \qquad (4.3)$$

The transformation from (4.2) to (4.3) differs from the transformation from (3.7) to (3.9) only in regard to the else line (setting q(x)=false in (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

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

---

(*) 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.

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

$$\begin{aligned} G3(x, v, \gamma) &= G3(\ell(x), v, [r(x) ! \gamma]) && \text{(def. of G3)} \\ &= SEND3(G2(\ell(x), v), [r(x) ! \gamma]) && \text{(IH)} \\ &= G3(r(x), G2(\ell(x), v), \gamma) && \text{(def. of SEND3)} \\ &= SEND3(G2(r(x), G2(\ell(x), v)), \gamma) && \text{(IH)} \\ &= SEND3(G2(x, v), \gamma) && \text{(def. of G2)} \blacksquare \end{aligned}$$

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

Proof: $F3(x)=G3(x, 1_b, [])=SEND3(G2(x, 1_b), []) = G2(x, 1_b)=F2(x)=F(x)$. $\blacksquare$

Equations (4.3) are in iterative form [12] and so do not require use of the run-time stack. Note that G3 and SEND3 are mutually tail-recursive; this corresponds to the use go-to'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 go-to is permissible [10]. While this code uses the go-to, the correctness proof bears little, if any, resemblance to a correctness proof for (4.3) using the inductive assertion method. The structure of the proof followed not the "object code" (4.3) but rather the "source code" (4.2), and is hence much easier.

Some other observations about (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 will use this later). Furthermore, the values of v are all of the form $b(b(b(1_b,a_1),..,a_n)$. 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 SEND3 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 "go-to" 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 SEND3. We unwind the call to G3 in the definition of SEND3 to get

$SEND4(v,\ \gamma)$ <= ;=$SEND3(v,\ \gamma)$

   <u>if</u> $\gamma$ = [] <u>then</u> v

      <u>else</u> <u>if</u> $p(hd(\gamma))$ <u>then</u> $SEND4(b(v,\ a(hd(\gamma))),\ t\ell(\gamma))$

      <u>else</u> $G3(\ell(hd(\gamma)),\ v,\ [r(hd(\gamma))\ !\ t\ell(\gamma)])$

Using the previous identity we get our next version:

$$\left.\begin{array}{l} F4(x) \Leftarrow SEND4(1_b,\ [x]) \\ SEND4(v,\ \gamma) \Leftarrow\ ;=SEND3(v,\ \gamma) \\ \quad \underline{if}\ \gamma = [\,]\ \underline{then}\ v \\ \qquad \underline{else}\ \underline{if}\ p(hd(\gamma))\ \underline{then}\ SEND4(b(v,\ a(hd(\gamma))),\ t\ell(\gamma)) \\ \qquad \underline{else}\ SEND4(v,\ [\ell(hd(\gamma))\ r(hd(\gamma))\ !\ t\ell(\gamma)]) \end{array}\right\}(4.4)$$

Note that SEND4 is just a while-loop and may therefore be claimed to be "more structured" than (4.3), which requires unrestricted gotos.

Proposition 4.3 $\quad SEND4(v,\gamma) = SEND3(v,\gamma)$

Proof: By subgoal induction on SEND4. ■

Proposition 4.4 $\quad F4(x) = F(x)$

Proof: $F4(x) = SEND4(1_b,\ [x]) = SEND3(1_b,[x]) = G3(x,\ 1_b,[\,]) = F3(x) = F(x)$. ■

We may further transform (4.4) by observing that SEND4 treats its first argument as an accumulator and is therefore the target of a transformation like that of Figure 3.1. We invert the transformation to get

$$\left.\begin{array}{l} F5(x) \Leftarrow SEND5([x]) \\ SEND5(\gamma) \Leftarrow \\ \quad \underline{if}\ \gamma=[\,]\ \underline{then}\ 1_b \\ \qquad \underline{else}\ \underline{if}\ p(hd(\gamma))\ \underline{then}\ b(a(hd(\gamma)),\ SEND5(t\ell(\gamma))) \\ \qquad \underline{else}\ SEND5([\ell(hd(\gamma))\ r(hd(\gamma))\ !\ t\ell(\gamma)]) \end{array}\right\}(4.5)$$

Proposition 4.5 $\quad SEND4(v,\gamma)=b(v,\ SEND5(\gamma))$

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 non-obvious. 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, $\gamma$ 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 (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 needs 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: $\alpha$-$\beta$ tree searching and the conversion of formulas of propositional logic to conjunctive normal form.

### 5.1 $\alpha$-$\beta$ 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 non-null list of sons. We have two functions, $F^+$ and $F^-$, which seek to maximize and minimize the values associated with a node:

$$\left.\begin{array}{l} F^{+}(x) \Leftarrow \\ \quad \underline{\text{if}}\ \text{leaf?}(x)\ \underline{\text{then}}\ \text{value}(x) \\ \qquad \underline{\text{else}}\ \max(\text{MAPHD}(F^{-}, \text{sons}(x))) \\ F^{-}(x) \Leftarrow \\ \quad \underline{\text{if}}\ \text{lead?}(x)\ \underline{\text{then}}\ \text{value}(x) \\ \qquad \underline{\text{else}}\ \min(\text{MAPHD}(F^{+}, \text{sons}(x))) \end{array}\right\} \quad (5.1.1)$$

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

$$
\left.
\begin{array}{l}
F2^{+}(x) \Leftarrow \\
\quad \underline{\text{if}}\ \text{leaf?}(x)\ \underline{\text{then}}\ \text{value}(x) \\
\qquad \underline{\text{else}}\ G2^{+}(\text{sons}(x)) \\
G2^{+}(\ell) \Leftarrow \\
\quad \underline{\text{if}}\ t\ell(\ell) = [\,]\ \underline{\text{then}}\ F2^{-}(\text{hd}(\ell)) \\
\qquad \underline{\text{else}}\ \max(F^{-}(\text{hd}(\ell)),\ G2^{+}(t\ell(\ell))) \\
F2^{-}(x) \Leftarrow \\
\quad \underline{\text{if}}\ \text{leaf?}(x)\ \underline{\text{then}}\ \text{value}(x) \\
\qquad \underline{\text{else}}\ G2^{-}(\text{sons}(x)) \\
G2^{-}(\ell) \Leftarrow \\
\quad \underline{\text{if}}\ t\ell(\ell) = [\,]\ \underline{\text{then}}\ F2^{+}(\text{hd}(\ell)) \\
\qquad \underline{\text{else}}\ \min(F2^{+}(\text{hd}(\ell)),\ G2^{-}(t\ell(\ell)))
\end{array}
\right\} \qquad (5.1.2)
$$

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 $\alpha \le \beta$, then $\max(\alpha,\min(\beta,v)) = \min(\beta, \max(\alpha,v))$

(ii) If $\alpha \le \beta$, then $\alpha \le \max(\alpha, \min(\beta,v)) \le \beta$

(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+(α,β,x) <=   ;=max(α,min(β,F2+(x))) if α≤β
    if leaf?(x) then max(α, min(β, value(x)))
        else G3+(α,β,sons(x))
G3+(α,β,ℓ) <=   ;=max(α,min(β,G2+(ℓ))) if α≤β
    if tℓ(ℓ) = [] then F3-(α,β,hd(ℓ))
        else max(α,min(β,max(F2-(hd(ℓ)), G2+(tℓ(ℓ))))))
```

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

```
max(α,min(β,
          max(F2-(hd(ℓ)), G2+(tℓ(ℓ))))))
= max(max(α,min(β,F2-(hd(ℓ))))
       max(α,min(β,G2+(tℓ(ℓ)))))
= max(F3-(α,β,hd(ℓ)),
        max(α,min(β,G2+(tℓ(ℓ)))))
```

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

```
G4+(α,β,ℓ) <=    ;= G3+(α,β,ℓ)
   if tℓ(ℓ)=[] then F3-(α,β,hd(ℓ))
       else H4+(α,β,tℓ(ℓ), F3-(α,β,hd(ℓ))))
H4+(α,β,ℓ,v) <= max(v, max(α,min(β,G2+(ℓ))))
```

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

```
max(v, max(α,min(β,G2+(ℓ))) =
    if v ≥ β then v else max(v, min(β,G2+(ℓ)))
```

Substituting this for H4, we reach

$F5^+(\alpha,\beta,x)$ <=

  *if* leaf? (x) *then* $\max(\alpha,\min(\beta,\mathrm{value}(x)))$

    *else* $G5^+(\alpha,\beta,\mathrm{sons}(x))$

$G5^+(\alpha,\beta,\ell)$ <=

  *if* $t\ell(\ell) = []$ *then* $F5^-(\alpha,\beta,hd(\ell)))$

    *else* $H5^+(\alpha,\beta,t\ell(\ell),\ F5^-(\alpha;\beta,hd(\ell)))$

$H5^+(\alpha,\beta,\ell,v)$ <=

  *if* $v \geq \beta$ *then* v *else* $G5^+(v,\beta,\ell)$

and the simultaneous corresponding minimizing functions. Here H5 performs cut-off.

This example is interesting because $\alpha$-$\beta$ 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 (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 accessable 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

$$C_1 \& C_2 \& .. \& C_n$$

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

$$\ell_1 \vee .. \vee \ell_{m_i}$$

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

```
  A => B -> ~A v B
~(A&B)   -> ~A v ~B
~(AvB)   ->  A & ~B
~~A      -> A
```

Then the distributive law

```
(A&B)vC  -> (AvC)&(BvC)
```

and its variants are aplied until the task is completed.

Removing implications and driving negations inwards 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*.

```
CNF1(x) <=

   if literal?(x) then [[x]]                                       (5.2.1)
    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 function (yet to be written) which performs approximately as follows:

$$\mathrm{DISTR}\ (C_1\ \&..\&C_n, D_1\ \&..\&D_m) = (C_1 \vee D_1)\&(C_1 \vee D_2)\&\ldots\&(C_n \vee D_m)$$

taking two formulas in c.n.f. and returning the n×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

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

of the last section, getting

```
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])        (5.2.2)
S2(v,γ) <=
  if γ=[] then v
    else G2(hd(γ),v,tℓ(γ))
```

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)) ≡ cnf(X∨(Y&Z))* ≡ cnf((X∨Y)&(X∨Z)) ≡ append(DISTR(X,Y), DISTR(X,Z)), where ≡ denotes logical equivalence among representations of cnf formulas. So the conjunction branch may be simplified to

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

* here cnf(X) means "some c.n.f. of X" rather than the c.n.f. produced by some particular program.

We can achieve a fold if we can prove:

<u>Proposition 5.1</u> S2(append(x,y),γ) = append(S2(x,γ), S2(y,γ))

<u>Proof</u>: We consider

S2'(v,γ) <= ;= S2(v,γ)

<u>if</u> γ = [] <u>then</u> v

<u>else</u> S2'(DISTR(v,CNF1(hd(γ))), tℓ(ℓ))

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

S2'(append(x,y),γ)

= S2'(DISTR(append(x,y), CNF1(hd(γ))),tℓ(γ))

= S2'(append(DISTR(x,CNF1(hd(γ)), DISTR(y,CNF1(hd(γ)))),
 tℓ(γ)) (by the argument above)

= append(S2'(DISTR(x,CNF1(hd(γ))), tℓ(γ))
 S2'(DISTR(y,CNF1(hd(γ))), tℓ(γ))) (by IH)

= append(S2'(x,γ), S2'(y,γ)). ∎

By Proposition 5.1, we can simplify the conjunction branch to
append(S2(DISTR(v,CNF1(op1(x))),γ),
 S2(DISTR(v,CNF1(op2(x))),γ))

which we fold with G2 to obtain:

```
CNF3(x) <= G3(x,[[]],[])
G3(x,v,γ) <= ;=S3(DISTR(v,CNF1(x)),γ)
   if literal?(x) then append(G3(op1(x),v,γ),            (5.2.3)
      else if conj?(x) then S3(DISTR(v,[[x]]),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 (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:

$$\mathrm{DISTR}(C_1\&..\&C_n,[[x]]) = (C_1 \vee x)\&..\&(C_n \vee x)$$

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 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 ! $C_1$]. We therefore rename v to lits, since it accumulates literals a clause. We also rename $\gamma$ to rest, since it stores the rest of x which will be processed later. These changes yield

```
CNF4(x) <=  G4(x, [], [])
G4(x, lits, rest) <=  ; = S4(DISTR([lits], CNF1(x)), rest)
   if literal?(x) then S4([x ! lits], rest)                        (5.2.4
      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 version (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:

```
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)        (5.2.5)
      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 (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 <u>escape</u> 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 towards 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, Chap. 6].

## 7. Acknowledgements

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 benefitted from discussions with S. C. Shapiro.

References


1. Aubin, Raymond. Some generalization heuristics in proofs by induction. Proc. IRIA Symp. Proving and Improving Programs, Arc-et-Senans, France (1975), 197-208.

2. Burstall, R.M., and Darlington, John. A transformation system for developing recursive programs. J. ACM 24 (1977), 44-67.

3. Chang, C.C., and Lee, R.C.T. Symbolic Logic and Mechanical Theorem-Proving. Academic Press, New York (1973).

4. Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R. Structured Programming. Academic Press, London (1972).

5. Fischer, Michael J. Lambda-calculus schemata. Proc. ACM Conf. on Proving Assertions about Programs (Los Cruces, 1972) SIGPLAN Notices 7, 1 (January, 1972), 104-109.

6. Hewitt, Carl, Bishop, Peter, and Steiger, Richard. A universal modular ACTOR formalism for artificial intelligence. Proc. Int'l Joint Conf. on Artificial Intelligence 3 (1973), 235-245.

7. Hewitt, Carl E., & Smith, Brian. Towards a programming apprentice. IEEE Trans. Software Eng. SE-1 (1975), 26-45.

8. Knuth, D.E. Semantics of context-free languages. Math. Sys. Th. 2 (1968), 127-145; correction, 5 (1971), 95-96.

9. Knuth, Donald E. An analysis of alpha-beta pruning. Stanford University Computer Science Dept., Technical Report STAN-CS-74-441 (1974).

10. Knuth, D. E. Structured programming with go to statements. Computing Surveys 6 (1974), 261-301.

11. Luckham, David C., and Suzuki, Norihisa. Automatic program verification IV: Proof of termination within a weak logic of programs. Stanford University Artificial Intelligence Laboratory Memo AIM-269 (October, 1975).

12. McCarthy, John. "Towards a mathematical science of computation. Information Processing 62 (Popplewell, ed.) North-Holland, Amsterdam, (1963), 21-28.

13. Manna, Zohar. Mathematical Theory of Computation. McGraw-Hill, New York, (1974).

References (Cont.)

14. Manna, Z., & Pnueli, A. Formalization of properties of functional programs. J.ACM 17 (1970), 555-569.

15. Manna, Zohar, and Vuillemin, John. Fixpoint approach to the theory of computation. Comm. ACM 15 (1972), 528-536.

16. Morris, James H. & Wegbreit, Ben. Subgoal induction. Comm. ACM 20 (1977), 209-222.

17. Moses, Joel. The function of FUNCTION in LISP. SIGSAM Bulletin 15 (July, 1970), 13-27.

18. Partsch, Helmut & Pepper, Peter. A family of rules for recursion removal. Info. Proc. Letters 5 (1976), 174-177.

19. Reynolds, J.C. Definitional interpreters for higher-order programming languages. Proc. ACM Nat'l Conf. (1972), 717-740.

20. Reynolds, J.C. On the relation between direct and continuation semantics. Proc. 2nd Colloq. on Automata, Languages, and Programming, Saarbrucken (1974).

21. Scott, D., and Strachey, C. Toward a mathematical semantics for computer languages. Computers and Automata (J. Fox, ed.), Wiley, New York (1972), 19-46.

22. Steele, Guy Lewis. LAMBDA: The ultimate declarative. Mass. Inst. of Tech. AI Memo 379 (October, 1976).

23. Steele, Guy Lewis, Jr. and Sussman, Gerald Jay. LAMBDA: The ultimate imperative. Mass. Inst. of Tech. AI Memo 353 (March, 1976).

24. Strachey, Christopher & Wadsworth, Christopher P. Continuations: A mathematical semantics for handling full jumps. Oxford University Computing Laboratory. Technical Monograph PRG-11 (January, 1974).

25. Strong, H.R. Translating recursion equations into flow charts. J. Comp. & Sys. Sci 5 (1971), 254-285.

26. Sussman, Gerald Jay, and Steele, Guy Lewis, Jr. SCHEME: An interpreter for extended lambda calculus. Mass. Inst. of Tech, AI Memo 349 (December, 1975).

References (Cont.)

27. Wand, M., and Friedman, D.P. Compiling lambda expressions using continuations. Indiana University Computer Science Department, Technical Report No. 55 (1976), to appear.

28. Wegbreit, Ben. Goal-directed program transformation IEEE Transac. on Software Engineering, SE-2 (1976), 69-79.

29. Wirth, W. Program development by stepwise refinement. Comm. ACM 14 (1971), 221-227.