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 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

4.22, 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.

\[
F(x,y) \leftarrow \text{if } p(x) \text{ then } y \quad \text{;this is a comment} \\
\text{else } a(b(x), F(c(x), y))
\]

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 trivial and to the ones we define by equations as serious [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 \(\text{hd}\) and \(\text{tl}\) to select the first element of a list and its remainder; thus

\[
\text{hd}([x \mid y]) = x \\
\text{tl}([x \mid y]) = y \\
\text{hd}([x \ y \ z]) = x \\
\text{tl}([x \ y \ z]) = [y \ z]
\]

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 \(\text{append}(x, y)\) concatenates two lists non-destructively; \(\text{append}\) is associative, so we will sometimes write \(\text{append}(x, y, z)\).

We will also occasionally use temporary functional objects passed as arguments. Such objects, called \textit{closures}, are created using \(\lambda\)-notation. For example, the definition

\[
\text{F}(x, y) := \text{MAPHD}(\lambda z. [x \mid z], y) \\
\text{MAPHD}(f, x) := \begin{cases} 
\text{if } x = [] & \text{then } [] \\
\text{else } [f(\text{hd}(x)) \mid \text{MAPHD}(f, \text{tl}(x))] 
\end{cases}
\]

passes a functional object as a parameter to \(\text{MAPHD}\). 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 \(\text{F}(3, [[1] [2] [3]])\) evaluates to \([[3 \ 1] [3 \ 2] [3 \ 3]]\). \(\text{MAPHD}\) is a generally useful function which we shall take as primitive. Another useful function is...
MAP! which is defined as

\[
\text{MAP!}(f, a, x) \iff \begin{cases} 
\text{if } x = [] \text{ then } a \\
\text{else } f(\text{hd}(x), \text{MAP!}(f, a, t\&x)) 
\end{cases}
\]

where \( f \) is a function of two arguments; \( \text{MAP!}(\text{sum}, 0, x) \) 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,
\[
F(x) = \text{ if } p(x) \text{ then } a(x) \\
\text{ else } b(F(\ell(x)), F(r(x)))
\]
produces the verification conditions
\[
\forall x [p(x) \Rightarrow \psi(x; a(x))]
\forall x \forall z_1 \forall z_2 [\neg p(x) \& \psi(\ell(x); z_1) \& \psi(r(x); z_2) \&
[\ell(x) = r(x) \Rightarrow z_2] \Rightarrow \psi(x; b(z_1, z_2))]
\]
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:

\[ \text{REV}(x) := \begin{cases} \text{if } x = [] \text{ then } [] \\ \text{else } \text{append} \left( \text{REV}(\text{tl}(x)) \right), \text{[hd}(x)] \end{cases} \]  

(3.1)

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

\[
\text{REV}^2(x) := \text{REVC}(x, \lambda z.z) \\
\text{REVC}(x, \gamma) := \begin{cases} \gamma(\text{REV}(x)) & \text{if } x = [] \\ \text{else } \text{REVC}(\text{tl}(x), \lambda w.\gamma(\text{append}(w, \text{[hd}(x)]))) \end{cases}
\]  

(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:

**Proposition 3.1** \( \text{REVC}(x, \gamma) = \gamma(\text{REV}(x)) \).

**Proof:** The specification \( \psi_{\text{REVC}}(x, \gamma; z) \) is \( z = \gamma(\text{REV}(x)) \).

\[ * ;= \text{may be read "is intended to equal"} \]
The verification conditions are:

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

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

\[ \Rightarrow \psi_{\text{REVC}}(x, \gamma; z)\]

Substituting the definition of \(\psi_{\text{REVC}}\), we have to show

(i) \((x=[]) \Rightarrow (\gamma([]) = \gamma(\text{REV}(x)))\)

(ii) \((x=[]) \& (z = (\lambda w. \gamma(\text{append}(w, [\text{hd}(x)])))(\text{REV}(t\!\&\!(x))))\)

\[ \Rightarrow (z = \gamma(\text{REV}(x)))\]

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

\((x \neq []) \& (z = (\lambda w. \lambda (\text{append}(w, [\text{hd}(x)])))(\text{REV}(t\!\&\!(x))))\)

\[ \Rightarrow (x \neq []) \& (z = \gamma(\text{REV}(t\!\&\!(x)), [\text{hd}(x)]))(\beta\text{-reduction})^*\]

\[ \Rightarrow z = \gamma(\text{REV}(x))\]  \hspace{1cm} \text{(def. of \(\text{REV}\), using \(x \neq []\)).}

The use of subgoal induction lets us prove the correctness of \(\text{REVC}\) essentially "line by line", referring to the definition of \(\text{REV}\) 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 \(\text{REV}\) terminates (which it always does), then so does \(\text{REVC}\). This may be proved by considering the performance specification

\[ \psi_{\text{REVC}}(x, \gamma; z) = \text{if } y \text{ occurs as a first argument to } \text{REVC} \]

during the computation of \(\text{REVC}(x, \gamma)\),

then \(y\) occurs as a first argument to \(\text{REV}\) during the computation of \(\text{REV}(x)\).

---

*The operation of \(\beta\text{-reduction}\) 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 γ supplied to REVC is of the form λv.append(v, a) for some a. To prove this, we observe that λz.z is of this form (with a = []), and if γ = λv.append(v, a), then

$$\lambda w. (\lambda v. append(w, [hd(x)]))$$

$$= \lambda w. ((\lambda v. append(v, a)) (append(w, [hd(x)]))) \quad \text{(def. of γ)}$$

$$= \lambda w. append(append(w, [hd(x)]), a) \quad \text{(β - reduction)}$$

$$= \lambda w. append(w, append([hd(x)], a)) \quad \text{(associativity)}$$

$$= \lambda w. append(w, [hd(x) \cdot a]) \quad \text{(fact about append)}$$

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

This gives us:

$$\begin{align*}
\text{REV3}(x) & \equiv \text{REV3}(x, []) \\
\text{REV3}(x, a) & \equiv \begin{cases} 
\text{append}(\text{REV}(x), a) \\
\text{if } x = [] \text{ then } a \\
\text{else } \text{REV3}(\text{tl}(x), [hd(x) \cdot a])
\end{cases}
\end{align*}$$

(3.3)

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.
\[
F(x) \leftarrow \\
\text{if } p(x) \text{ then } a(x) \\
\text{else } b(F(c(x)), d(x))
\]
where \( b \) is associative, with right identity \( l_b \), is replaced by

\[
F'(x) \leftarrow FC(x, l_b) \\
FC(x, \gamma) \leftarrow \ ; = b(F(x), \gamma) \\
\text{if } p(x) \text{ then } b(a(x), \gamma) \\
\text{else } FC(c(x), b(d(x), \gamma))
\]

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 \( \neg p(x) \):

\[
FC(x, \gamma) = FC(c(x), b(d(x), \gamma)) \\
= b(F(c(x)), b(d(x), \gamma)) \quad \text{(IH)} \\
= b(b(F(c(x)), d(x)), \gamma) \quad \text{(associativity of } b) \\
= b(F(x), \gamma) \quad \text{(definition of } F) 
\]

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:

\[
F(x) \Leftarrow \begin{cases} 
  \text{if } x > 100 \text{ then } x - 10 & \text{else } F(F(x+11)) 
\end{cases}
\]  

(3.4)

In continuation form this becomes:

\[
\begin{align*}
F_2(x) & \Leftarrow F_2-C(x, \lambda z. z) \\
F_2-C(x, \gamma) & \Leftarrow \gamma(F(x)) \\
& \begin{cases} 
  \text{if } x > 100 \text{ then } \gamma(x - 10) \\
  \text{else } F_2-C(x+11, \lambda w. \gamma(F(w))) 
\end{cases}
\end{align*}
\]  

(3.5)

Here again, we can obtain a closed form for the continuation: it is always of the form \(\lambda w.F(F(F(\ldots(w))))\) 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 \(F_3-\text{SEND}\) which simulates functional application for the given representation of the continuation:

\[
\begin{align*}
F_3(x) & \Leftarrow F_3-C(x, 0) \\
F_3-C(x, i) & \Leftarrow \begin{cases} 
  \text{if } x > 100 \text{ then } F_3-\text{SEND}(x - 10, i) \\
  \text{else } F_3-C(x+11, i+1) 
\end{cases} \\
F_3-\text{SEND}(v, i) & \Leftarrow \begin{cases} 
  \text{if } i = 0 \text{ then } v \\
  \text{else } F_3-C(v, i-1) 
\end{cases}
\end{align*}
\]  

(3.6)

(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:

\[
F(x) = \begin{cases} 
  a(x) & \text{if } p(x) \text{ then} \\
  F(b(x)) & \text{else if } q(x) \text{ then} \\
  c(d(x), F(e(x))) & \text{else}
\end{cases}
\]  

Introducing continuations, we get

\[
F2(x) = F2-C(x, \lambda z.z) \\
F2-C(x, \gamma) = \begin{cases} 
  \gamma(a(x)) & \text{if } p(x) \text{ then} \\
  \gamma(F(b(x))) & \text{else if } q(x) \text{ then} \\
  \gamma(c(d(x), F(e(x)))) & \text{else}
\end{cases}
\]  

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 \([a_n \ldots a_1]\), giving:

\[
F3(x) = F3-C(x, []) \\
F3-C(x, \gamma) = \begin{cases} 
  F3-SEND(a(x), \gamma) & \text{if } p(x) \text{ then} \\
  F3-C(b(x), \gamma) & \text{else if } q(x) \text{ then} \\
  F3-C(e(x), [d(x) ! \gamma]) & \text{else}
\end{cases}
\]

\[
F3-SEND(v, \gamma) = \begin{cases} 
  v & \text{if } \gamma=[] \text{ then} \\
  F3-SEND(c(hd(\gamma), v), \text{tl}(\gamma)) & \text{else}
\end{cases}
\]
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 straightforward as that for (3.8). In (3.8) we had the specification \(F_2-C(x, \gamma) \gamma(F(x))\). In (3.9), \(\gamma\) is no longer a function but is rather its representation. Hence the corresponding specification is \(F_3-C(x, \gamma) = F_3-SEND(F(x), \gamma)\).

**Proposition 3.3** For all \(x\) and \(\gamma\), \(F_3-C(x, \gamma) = F_3-SEND(F(x), \gamma)\).

**Proof:** By subgoal induction on \(F\). If \(p(x)\), then

\[
F_3-C(x, \gamma) = F_3-SEND(a(x), \gamma) = F_3-SEND(F(x), \gamma)
\]

(def. of \(F_3-C\))

Otherwise, if \(q(x)\), then

\[
F_3-C(x, \gamma) = F_3-C(b(x), \gamma)
\]

(def. of \(F_3-C\))

\[
= F_3-SEND(F(b(x)), \gamma) = F_3-SEND(F(x), \gamma)
\]

(IH)

(def. of \(F\)).

\(^*\)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,
\[
F_3-C(x, \gamma) = F_3-C(e(x), [d(x) ! \gamma]) \quad \text{(def. of } F_3-C) \\
= F_3-SEND(F(e(x)), [d(x) ! \gamma]) \quad \text{(IH)} \\
= F_3-SEND(c(d(x), F(e(x))), \gamma) \quad \text{(def. of } F_3-SEND) \\
= F_3-SEND(F(x), \gamma) \quad \text{(def. of } F). \]

**Proposition 3.4** \( F_3(x) = F(x) \)

**Proof:** \( F_3(x) = F_3-C(x, []) = F_3-SEND(F(x), []) = F(x). \)

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:

\[
G(x) :\ = \\
\begin{array}{ll}
\text{if } p(x) \text{ then } a(x) \\
\text{else if } q(x) \text{ then } b(G(c(x))) \\
\text{else } d(G(\ell(x)), G(r(x)))
\end{array}
\]

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:

\[
G_2(x) :\ = G_2-C(x, 0) \\
G_2-C(x, 1) :\ = b^{(1)}(G(x)) \\
\begin{array}{ll}
\text{if } p(x) \text{ then } G_2-SEND(a(x), 1) \\
\text{else if } q(x) \text{ then } G_2-C(c(x), i+1) \\
\text{else } ?
\end{array}
\]
The desired value in the place of the question mark is 
\( b^{(1)}(d(G(l(x)), G(r(x)))) \). By the distributive law, this
is equal to 
\( d(b^{(1)}(G(l(x))), b^{(1)}(G(r(x)))) = d(G_2-C(l(x), 1), G_2-C(r(x), 1)) \)

Thus we get:

\[
G_2-C(x, 1) <= b^{(1)}(G(x)) \\
\text{if } p(x) \text{ then } G_2-\text{SEND}(a(x), 1) \\
\text{else if } q(x) \text{ then } G_2-C(c(x), 1+1) \\
\text{else } d(G_2-C(l(x), 1), G_2-C(r(x), 1)) \\
G_2-\text{SEND}(v, 1) <= b^{(1)}(v) \\
\text{if } i=0 \text{ then } v \\
\text{else } G_2-\text{SEND}(b(v), 1-1)
\]

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

\[ F(x) = \begin{cases} 
\text{if } p(x) \text{ then } a(x) \\
\text{else } b(F(l(x)), F(r(x)))
\end{cases} \quad (4.1) \]

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

\( \lambda l. \text{MAP!}(b, l_b, \text{MAP}(F, l)) \)

(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 \( l \) 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 the last section. A continuation, however,
is a summary of the tree above the current node, and is therefore an inherited attribute. (See Figure 4.1)

(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.

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:

\[
\begin{align*}
F_3(x) & \Leftarrow G_3(x, l_b, []) \\
G_3(x, v, \gamma) & \Leftarrow \\
& \quad \begin{cases} 
\text{if } p(x) \text{ then } SEND_3(b(v, a(x)), \gamma) \\
\text{else } G_3(\lambda(x), v, [r(x) \uplus \gamma]) 
\end{cases} \\
SEND_3(v, \gamma) & \Leftarrow \\
& \quad \begin{cases} 
\text{if } \gamma = [] \text{ then } v \\
\text{else } G_3(\mathrm{hd}(\gamma), v, \mathrm{tl}(\gamma)) 
\end{cases}
\end{align*}
\]

(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** \(G_3(x, v, \gamma) = SEND_3(G_2(x, v), \gamma)\)

**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

\[\neg p(x) \& \psi_{G_2}(\lambda(x), v; z_1) \& \psi_{G_2}(r(x), z_1; z_2) \Rightarrow \psi_{G_2}(x, v; z_2)\]

(*) 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:

\[
G_3(x, v, \gamma) = G_3(\& (x), v, [r(x) \uparrow \gamma]) \quad \text{(def. of } G_3) \\
= \text{SEND}_3(G_2(\& (x), v), [r(x) \uparrow \gamma]) \quad \text{(IH)} \\
= G_3(r(x), G_2(\& (x), v), \gamma) \quad \text{(def. of } \text{SEND}_3) \\
= \text{SEND}_3(G_2(r(x), G_2(\& (x), v)), \gamma) \quad \text{(IH)} \\
= \text{SEND}_3(G_2(x, v), \gamma) \quad \text{(def. of } G_2) \quad \blacksquare
\]

**Proposition 4.2** \( F_3(x) = F(x) \).

**Proof:** \( F_3(x) = G_3(x, l_b, []) = \text{SEND}_3(G_2(x, l_b), []) = G_2(x, l_b) = F_2(x) = F(x) \). \( \blacksquare \)

Equations (4.3) are in iterative form [12] and so do not require use of the run-time stack. Note that \( G_3 \) and \( \text{SEND}_3 \) 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
\[ b(b(b(l_b, a_1), \ldots, a_n), \ldots). \]
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 \( F_3 \)
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 \( u \),
\[ \text{SEND3}(v, [t \ u \ ! \ y]) = G3(t, v, [u \ ! \ y]) \]
by unwinding the definition of SEND3. We unwind the call
to \( G3 \) in the definition of SEND3 to get
\[ \text{SEND4}(v, \ y) := \text{SEND3}(v, \ y) \]
if \( y = [] \) then \( v \)
else if \( p(hd(y)) \) then \( \text{SEND4}(b(v, a(hd(y))), t\&(y)) \)
else \( G3(t\&(hd(y)), v, [r(hd(y)) \ ! t\&(y)]) \)
Using the previous identity we get our next version:

\[
F_4(x) \leq \text{SEND}_4(1_b, [x])
\]

\[
\text{SEND}_4(v, \gamma) \leq \quad \equiv \text{SEND}_3(v, \gamma)
\]

\[
\begin{align*}
\text{if } \gamma = [] & \text{ then } v \\
\text{else if } p(\text{hd}(\gamma)) & \text{ then } \text{SEND}_4(b(v, a(\text{hd}(\gamma))), \text{tl}(\gamma)) \\
\text{else } & \text{SEND}_4(v, [l(\text{hd}(\gamma)) \ r(\text{hd}(\gamma)) \ ! \text{tl}(\gamma)]])
\end{align*}
\]

Note that \text{SEND}_4 is just a \textit{while}-loop and may therefore be claimed to be "more structured" than (4.3), which requires unrestricted gotos.

**Proposition 4.3** \text{SEND}_4(v,\gamma) = \text{SEND}_3(v,\gamma)

**Proof:** By subgoal induction on \text{SEND}_4. \qed

**Proposition 4.4** \(F_4(x) = F(x)\)

**Proof:** \(F_4(x) = \text{SEND}_4(1_b, [x]) = \text{SEND}_3(1_b, [x]) = \text{G}_3(x, 1_b, []) = \text{F}_3(x) = F(x)\). \qed

We may further transform (4.4) by observing that \text{SEND}_4 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

\[
F_5(x) \leq \text{SEND}_5([x])
\]

\[
\text{SEND}_5(\gamma) \leq
\]

\[
\begin{align*}
\text{if } \gamma = [] & \text{ then } 1_b \\
\text{else if } p(\text{hd}(\gamma)) & \text{ then } b(a(\text{hd}(\gamma)), \text{SEND}_5(\text{tl}(\gamma))) \\
\text{else } & \text{SEND}_5([l(\text{hd}(\gamma)) \ r(\text{hd}(\gamma)) \ ! \text{tl}(\gamma)])
\end{align*}
\]

**Proposition 4.5** \(\text{SEND}_4(v,\gamma)=b(v, \text{SEND}_5(\gamma))\)
Proof: By induction on SEND5. □

**Proposition 4.6** \( F_5(x) = F(x) \)

**Proof:** \( F_5(x) = SEND5([x]) = b(l_b, SEND5([x])) = SEND4(l_b, [x]) = F_4(x) = F(x) \). □

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 "\( F_5(x) \) 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: 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 non-null list of sons. We have two functions, \( F^+ \) and \( F^- \), which seek to maximize and minimize the values associated with a node:

\[
\begin{align*}
F^+(x) & = \begin{cases} 
\text{if } \text{leaf?(}x\text{) then } \text{value}(x) \\
\text{else } \max(\text{MAPHD}(F^-, \text{sons}(x)))
\end{cases} \\
F^-(x) & = \begin{cases} 
\text{if } \text{lead?(}x\text{) then } \text{value}(x) \\
\text{else } \min(\text{MAPHD}(F^+, \text{sons}(x)))
\end{cases}
\end{align*}
\]

(5.1.1)

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

\[
F_2^+(x) \leq \\
\begin{cases} 
\text{if leaf?(x) then value(x)} \\
\text{else } G_2^+(\text{sons}(x))
\end{cases}
\]

\[
G_2^+(\lambda) \leq \\
\begin{cases} 
\text{if } tl(\lambda) = [] \text{ then } F_2^-(\text{hd}(\lambda)) \\
\text{else } \max(F_2^-(\text{hd}(\lambda)), G_2^+(\text{tl}(\lambda)))
\end{cases}
\]

\[
F_2^-(x) \leq \\
\begin{cases} 
\text{if leaf?(x) then value(x)} \\
\text{else } G_2^-(\text{sons}(x))
\end{cases}
\]

\[
G_2^-(\lambda) \leq \\
\begin{cases} 
\text{if } tl(\lambda) = [] \text{ then } F_2^+(\text{hd}(\lambda)) \\
\text{else } \min(F_2^+(\text{hd}(\lambda)), G_2^-(\text{tl}(\lambda)))
\end{cases}
\]

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) \( \min(v, \max(x, y)) = \max(\min(v, x), \min(v, y)) \) \( \blacksquare \)
We therefore consider the conditional generalization:
\[ F^+(\alpha, \beta, x) \leq \max(\alpha, \min(\beta, F^+(x))) \quad \text{if} \quad \alpha \leq \beta \]

\[
\text{if leaf?(x) then } \max(\alpha, \min(\beta, \text{value}(x))) \\
\text{else } G_3^+(\alpha, \beta, \text{sons}(x))
\]

\[ G_3^+(\alpha, \beta, l) \leq \max(\alpha, \min(\beta, G^+(l))) \quad \text{if} \quad \alpha \leq \beta \]

\[
\text{if } tl(l) = [] \text{ then } F^-^+(\alpha, \beta, \text{hd}(l)) \\
\text{else } \max(\alpha, \min(\beta, \max(F^-(hd(l)), G^+(tl(l))))))
\]

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

\[
\max(\alpha, \min(\beta, \\
\max(F^-(\text{hd}(l)), G^+(\text{tl}(l))))))
\]

\[
= \max(\max(\alpha, \min(\beta, F^-(\text{hd}(l))))), \\
\max(\alpha, \min(\beta, G^+(\text{tl}(l))))))
\]

\[
= \max(F^-^+(\alpha, \beta, \text{hd}(l)), \\
\max(\alpha, \min(\beta, G^+(\text{tl}(l))))))
\]

Introducing a help function for this continuation of \( F^-^+ \), we obtain:

\[ G_4^+(\alpha, \beta, l) \leq G_3^+(\alpha, \beta, l) \]

\[
\text{if } tl(l) = [] \text{ then } F^-^+(\alpha, \beta, \text{hd}(l)) \\
\text{else } H_4^+(\alpha, \beta, tl(l), F^-^+(\alpha, \beta, \text{hd}(l))))
\]

\[ H_4^+(\alpha, \beta, l, v) \leq \max(v, \max(\alpha, \min(\beta, G^+(l))))
\]

But \( \alpha \leq v \leq \beta \), so

\[
\max(v, \max(\alpha, \min(\beta, G^+(l)))) = \\
\text{if } v \geq \beta \text{ then } v \text{ else } \max(v, \min(\beta, G^+(l)))
\]
Substituting this for $H_4$, we reach

$$F_5^+(\alpha, \beta, x) \leq$$

if leaf? (x) then max(\alpha, min(\beta, value(x)))
else $G_5^+(\alpha, \beta, sons(x))$

$$G_5^+(\alpha, \beta, \lambda) \leq$$

if t&l(\lambda) = [] then $F_5^-(\alpha, \beta, hd(\lambda))$
else $H_5^+(\alpha, \beta, t&l(\lambda), F_5^-(\alpha, \beta, hd(\lambda))$

$$H_5^+(\alpha, \beta, \lambda, v) \leq$$

if v > \beta then v else $G_5^+(v, \beta, \lambda)$

and the simultaneous corresponding minimizing functions.

Here $H_5$ 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 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

\[ C_1 \land C_2 \land \ldots \land C_n \]

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

\[ \ell_1 \lor \ldots \lor \ell_{m_i} \]

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:

\[ A \rightarrow B \rightarrow \neg A \lor B \]

\[ \neg (A \land B) \rightarrow \neg A \lor \neg B \]

\[ \neg (A \lor B) \rightarrow A \land \neg B \]

\[ \neg \neg A \rightarrow A \]

Then the distributive law

\[ (A \land B) \lor C \rightarrow (A \lor C) \land (B \lor C) \]

and its variants are applied 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*.

$$\text{CNFl}(x) \leftarrow$$

if literal?(x) then \([[[x]]]\) \hspace{1cm} (5.2.1)

else if conj?(x) then append(CNFl(op1(x)), CNFl(op2(x)))

else if disj?(x) then DISTR(CNFl(op1(x)), CNFl(op2(x)))

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

$$\text{DISTR} \left( C_1 \land \ldots \land C_n, D_1 \land \ldots \land D_m \right) = \left( C_1 \lor D_1 \right) \land \left( C_1 \lor D_2 \right) \land \ldots \land \left( C_n \lor D_m \right)$$

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

(*): 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.
of the last section, getting

\[ \text{CNF2}(x) \equiv \text{G2}(x, [[]], []) \]

\[ \text{G2}(x,v,\gamma) \equiv \text{G2}(\text{DISTR}(v, \text{CNF1}(x)), \gamma) \]

if \( \text{literal?(x)} \) then \( \text{G2}(\text{DISTR}(v, [[]]), \gamma) \)
else if \( \text{conj?}(x) \) then \( \text{G2}(\text{DISTR}(v, \text{append}(\text{CNF1}(\text{op1}(x)), \text{CNF1}(\text{op2}(x)))), \gamma) \)
else if \( \text{disj?}(x) \) then \( \text{G2}(\text{op1}(x), v, [\text{op2}(x) \land \gamma]) \)

\[ \text{S2}(v, \gamma) \equiv \]

if \( \gamma = [] \) then \( v \)
else \( \text{G2}(\text{hd}(\gamma), v, \text{tl}(\gamma)) \)

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

We now note that \( \text{DISTR}(X, \text{append}(Y, Z)) \equiv \text{cnf}(Xv(Y\land Z)) \)
\( \equiv \text{cnf}((XvY)\land(XvZ)) \equiv \text{append}(\text{DISTR}(X, Y), \text{DISTR}(X, Z)) \), where \( \equiv \) denotes logical equivalence among representations of \text{cnf} formulas. So the conjunction branch may be simplified to

\[ \text{S2}(\text{append}(\text{DISTR}(v, \text{CNF1}(\text{op1}(x))), \text{DISTR}(v, \text{CNF1}(\text{op2}(x)))), \gamma) \]

\[ \text{(5.2.2)} \]

*Here \text{cnf}(X) means "some \text{c.n.f. of } X" rather than the \text{c.n.f.} produced by some particular program.*
We can achieve a fold if we can prove:

**Proposition 5.1** \( S_2(\text{append}(x,y), \gamma) = \text{append}(S_2(x,\gamma), S_2(y,\gamma)) \)

**Proof:** We consider

\[
S_2'(v, \gamma) <= \begin{cases} 
= S_2(v, \gamma) \\
\text{if } \gamma = [] \text{ then } v \\
\text{else } S_2'(\text{DISTR}(v, \text{CNFL}(\text{hd}(\gamma))), \text{tl}(\gamma))
\end{cases}
\]

\( S_2'(v, \gamma) = S_2(v, \gamma) \) follows by subgoal induction on \( S_2' \) using the fact that \( G_2(x, v, \gamma) = S_2(\text{DISTR}(v, \text{CNFL}(x)), \gamma) \).

We then prove the proposition \( (\forall x, y)[S_2'(\text{append}(x, y), \gamma) = \text{append}(S_2'(x, \gamma), S_2'(y, \gamma))] \) by induction on \( \gamma \). If \( \gamma = [] \), then both sides equal \( \text{append}(x, y) \). If \( \gamma \neq [] \), then

\[
S_2'(\text{append}(x, y), \gamma) \\
= S_2'(\text{DISTR}(\text{append}(x, y), \text{CNFL}(\text{hd}(\gamma))), \text{tl}(\gamma)) \\
= S_2'(\text{append}(\text{DISTR}(x, \text{CNFL}(\text{hd}(\gamma))), \text{DISTR}(y, \text{CNFL}(\text{hd}(\gamma))))), \text{tl}(\gamma)) \text{ (by the argument above)} \\
= \text{append}(S_2'(\text{DISTR}(x, \text{CNFL}(\text{hd}(\gamma))), \text{tl}(\gamma))), S_2'(\text{DISTR}(y, \text{CNFL}(\text{hd}(\gamma))), \text{tl}(\gamma))) \text{ (by IH)} \\
= \text{append}(S_2'(x, \gamma), S_2'(y, \gamma)).
\]

By Proposition 5.1, we can simplify the conjunction branch to

\[
\text{append}(S_2(\text{DISTR}(v, \text{CNFL}(\text{op1}(x))), \gamma), S_2(\text{DISTR}(v, \text{CNFL}(\text{op2}(x))), \gamma))
\]
which we fold with G2 to obtain:

\[
CNF3(x) \triangleq G3(x, [[[]]], [])
\]

\[
G3(x, v, \gamma) \triangleq \begin{cases} 
\text{S3(DISTR}(v, CNF1(x)), \gamma) \\
\text{if literal?}(x) \text{ then append}(G3(op1(x), v, \gamma),) \\
\text{else if conj?}(x) \text{ then S3(DISTR}(v, [[x]]), \gamma) \\
\text{else if disj?}(x) \text{ then G3(op1(x), v, [op2(x) \mid \gamma])} \\
\end{cases}
\]

\[
S3(v, \gamma) \triangleq 
\begin{cases} 
\text{if } \gamma = [] \text{ then } v \\
\text{else } G3(hd(\gamma), v, t\ell(\gamma)) \\
\end{cases}
\]

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:

\[
\text{DISTR}(C_1 \& \ldots \& C_n, [[x]]) = (C_1 v x) \& \ldots \&(C_n v x)
\]

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 \([x \mid C_1]\). 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

$$\text{CNF}^4(x) \Leftarrow \text{G}^4(x, [], [])$$

$$\text{G}^4(x, \text{lits}, \text{rest}) \Leftarrow ; = \text{S}^4(\text{DIST}([\text{lits}], \text{CNF}^1(x)), \text{rest})$$

1. **if** literal?($x$) **then** S4([x ! lits], rest)
2. **else if** conj?($x$) **then** append($\text{G}^4(\text{op}1(x), \text{lits}, \text{rest})$)
   $$\text{G}^4(\text{op}2(x), \text{lits}, \text{rest})$$
3. **else if** disj?($x$) **then** $\text{G}^4(\text{op}1(x), \text{lits}, [\text{op}2(x) ! \text{rest}])$

$$\text{S}^4(\text{lits}, \text{rest}) \Leftarrow$$

1. **if** rest = [] **then** [lits]
2. **else** $\text{G}^4(\text{hd}(\text{rest}), \text{lits}, \text{tl}(\text{rest}))$

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
We name the accumulator clauses. The result is:

\[
\text{CNF5}(x) = \text{G5}(x, [], [], [], [])
\]

\[
\text{G5}(x, \text{lits}, \text{rest}, \text{clauses}, \gamma) \Leftarrow \text{SEND5} (\text{append} (\text{clauses},
\text{G4}(x, \text{lits}, \text{rest}),
\gamma))
\]

\[
\text{if literal?(x) then } \text{S5}([x \oplus \text{lits}], \text{rest}, \text{clauses}, \gamma)
\]

\[
\text{else if conj?(x) then } \text{G5} (\text{opl}(x), \text{lits}, \text{rest}, \text{clauses},
[\text{op2}(x), \text{lits}, \text{rest} \ominus \gamma])
\]

\[
\text{else if disj?(x) then } \text{G5} (\text{opl}(x), \text{lits}, \text{[op2}(x) \ominus \text{rest}],
\text{clauses}, \gamma)
\]

\[
\text{S5} (\text{lits}, \text{rest}, \text{clauses}, \gamma) \Leftarrow
\]

\[
\text{if rest = [] then } \text{SEND5} ([\text{lits} \ominus \text{clauses}], \gamma) \quad (5.2.5)
\]

\[
\text{else } \text{G5} (\text{hd}(\text{rest}), \text{lits}, \text{tl}(\text{rest}), \text{clauses}, \gamma)
\]

\[
\text{SEND5} (\text{clauses}, \gamma) \Leftarrow
\]

\[
\text{if } \gamma = [] \text{ then clauses}
\]

\[
\text{else } \text{G5} (\text{hd}(\gamma), \text{hd}(\text{tl}(\gamma)), \text{hd}(\text{tl}(\text{tl}(\gamma))), \text{clauses},
\text{tl}(\text{tl}(\text{tl}(\gamma))))
\]

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:

\[
\text{if literal?(x) then}
\]

\[
\text{if tautology?(x, lits) then } \text{SEND5} (\text{clauses}, \gamma)
\]

\[
\text{else } \text{S5} ([x \ominus \text{lits}], \text{rest}, \text{clauses}, \gamma)
\]

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

\[
\text{if rest = [] then}
\]

\[
\text{if subsumed?(lits, clauses) then } \text{SEND5} (\text{clauses}, \gamma)
\]

\[
\text{else } \text{SEND5} ([\text{lits} \ominus \text{clauses}], \gamma)
\]
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 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 benefited from discussions with S. C. Shapiro.
References


References (Cont.)


References (Cont.)

