Compiling Lambda Expressions Using Continuations and Factorizations

Mitchell Wand
Daniel P. Friedman
Indiana University
Bloomington, Indiana 47401

TECHNICAL REPORT NO. 55

COMPILING LAMBDA EXPRESSIONS USING CONTINUATIONS AND FACTORIZATIONS

MITCHELL WAND
DANIEL P. FRIEDMAN

REVISED: JULY 1977

To appear Computer Languages (1978).

Research reported herein was supported in part by the National Science Foundation under grants numbered DCR75-06678 and MCS75-08145.

Abstract: We present a source-level transformation for recursion-removal with several interesting characteristics:

(i) the algorithm is simple and provably correct.

(ii) the stack utilization regime is chosen by the compiler rather than being fixed by the run-time environment.

(iii) the stack is available at the source language level so that further optimizations are possible.

(iv) the algorithm arises from ideas in category theory.

In addition to its implications for compilers, the transformation algorithm is useful as an implementation technique for advanced LISP-based systems, and one such application is described.

Key Words: compilation, continuations, factorization, lambda-expressions, LISP, recursion-removal.

1. Introduction

There has been considerable recent interest in compiling (or, more strictly speaking, code generation) via source-to-source transformations [12, 15, 17]. In this paper, we consider a class of transformations which extend this technique to recursive languages such as LISP. Code generation for such languages consists of a recursion removal phase in which the recursive program is transformed into an iterative one which utilizes a stack, followed by conventional optimization and code generation.

We present a source-level transformation for recursion-removal which is interesting for several reasons:

(i) the algorithm is simple and provably correct (although we do not present a correctness proof here).

(ii) depending on the choice made at the key step in the algorithm ("factorization"), a variety of alternative stack utilization regimes may be employed. Conventional compiling techniques seem to view stack usage as part of the run-time system for the language as a whole, rather than as a decision to be made by the compiler.

(iii) the stack is available at the source language level so that other optimizations can take place. Many clever tricks, including some involving alternative stack regimes, are apparently familiar to some assembly-language programmers.(*) Despite this

---

(*) Typically communicated by marginal notes of the form "of course, on the DEC-10, you can replace α with β" where α and β are strings over the alphabet {PUSH,PUSHJ,JRST}. Such notes, of course, appear only rarely in the published literature.

fact, we believe that our account of such techniques is useful, since it liberates them from the realm of undocumented "coding tricks" and allows their use by a somewhat wider class of programmers.

(iv) Algebraic identities are used as the source language.

The algorithm converts recursive LISP source code into a system of identities which can then be transformed into iterative code for a variety of host media. The idea of using the identities as an intermediary arose from a recognition of the similarity between the results of two independent investigations: a study of the relation between direct and continuation semantics in the framework of category theory, and a study of a lambda calculus interpreter based on Sussman and Steele's SCHEME [29] to isolate its non-applicative aspects. We have consciously suppressed the categorical ideas in our presentation, on the hypothesis that a necessary condition for the utility of a theory is that it yield useful applications which do not require intimate knowledge of the underlying theory.(*) Related results in category theory are presented elsewhere [32, 33].

These transformations produce identities which can be translated into iterative code in a straightforward way. If we

---

(*) e.g. one need not understand thermodynamics to enjoy an air conditioner.

were interested only in compiling for a conventional machine, standard code generation techniques [26] would suffice. Our primary interest, however, is in using the transformations as an implementation technique for advanced LISP-based systems. Toward that end, we present a series of "host media," or methods for implementing the identities in LISP. All these hosts use only a bounded portion of the LISP run-time stack, the major part of the stack being maintained by the compiled program in the heap (and therefore being subject to local optimizations [31]). Some register allocation problems are considered in this model.

As an example of the power of these techniques, we consider an implementation problem relating to the Friedman & Wise suspending "CONS " [10].

The remainder of the paper is divided as follows:

Section 2 gives a brief discussion of those features of LISP which we utilize. Section 3 describes the source language and preprocessor. Section 4 describes the main transformation. Section 5 discusses three host media. Section 6 presents an application of these techniques to an implementation problem. Finally, Section 7 discusses some related work by other investigators.

2. Features of LISP

This section briefly indicates those features of LISP which are important for this paper. Section 2.1 gives LISP basics required for an understanding of the algorithm; Section 2.2 discusses some more advanced properties which are used for the host media. This section is not intended to be tutorial; for more detail see [8, 18, 21, 34].

2.1 Basic LISP

The basic data structure in LISP is the list. A list is a sequence of elements, each of which is either a list or an identifier (called in LISP an atom); the null list, called NIL, is also permitted. Lists are delimited by parentheses. Thus ((FOO B) (AR) BAZ) is a list of 3 elements, two of which are lists and one of which is an atom. Lists are decomposed by two functions, CAR and CDR; CAR of a list returns its first element and CDR of a list returns all but the first element. Thus the CAR of the list above is (FOO B) and its CDR is ((AR) BAZ). Lists are built with the function LIST, which, applied to n arguments, returns a list consisting of those arguments. (LIST is in fact synthesized from the more primitive constructor, CONS which allocates a cell with CAR and CDR fields.)

A distinctive feature of LISP is that expressions to be evaluated are represented as lists, just as data are. Expressions to be evaluated are called forms; the CAR of a form is the function to be applied, and the CDR is the list of arguments. Thus

(CAR (LIST (QUOTE (A B)) (QUOTE C))) evaluates to the list (A B). Evaluation may also be invoked by use of the LISP function EVAL. Thus, if the current value of DEBUG is the list (CAR X) and the current value of X is the list (A B), then evaluation of (EVAL DEBUG) returns the atom A.

Functional objects are represented in LISP by lists of the form (LAMBDA formal-param-list body). Data structures of this form are interpreted as functions by the function APPLY. Thus, if the current value of F is the list (LAMBDA (X Y) (LIST Y X (CAR F))) and the current value of Y is the list (A B), then evaluation of (APPLY F (LIST Y (CAR Y))) yields as its value the list (A (A B) LAMBDA).

APPLY causes the body of the lambda-expression to be evaluated in an environment which is the current environment (at the time APPLY is called) extended with the values of the actual parameters assigned to the formal parameters. Thus free variables in the lambda-expression take their values from the call-time environment. This is called dynamic binding, contrasted with static binding, in which free variables retain their values from the time the functional object is created.

Another special kind of list is an association list, which is a list of key-data pairs.

2.2 Property Lists

In LISP, atoms have structure beyond their printed name. All atoms occurring in an input stream are entered in a symbol table called the oblist, and references to atoms are replaced in the machine representation by pointers to the appropriate symbol table entry. This entry is itself a list, called the *property list* of the atom, which functions as an association list. Keys on this association list must be atoms, and are called *indicators*. The print name and the current value of an atom are two properties stored on the property list.(*) The technique of storing the current value of an atom on its property list is called *shallow binding*. (The alternative, that of keeping all atom-value pairs on a single association list, is called *deep binding* and is used in some systems.) The function SETQ may be used to accomplish assignment.

The user may store and retrieve data from the property list using the functions GET and PUTPROP. For our purposes all property lists are static (except for values), so we introduce some notation for creation of indicator-value pairs:

$$F \underset{\text{EXPR}}{\equiv} \text{(LAMBDA (X) (G (CAR X)))}$$

The indicated lambda-expression is to be placed on the property list of F under the EXPR indicator. The EXPR property is used by the LISP system to store functional values for use by EVAL.

(*) This is a somewhat idealized discussion. In most LISP systems, the value property is treated specially to improve performance. Also, the property list may have a format slightly different from that of an association list.

A useful feature found in some LISPs is the REPEAT loop, [35] of which we shall use only the following special case: (REPEAT UNTIL boolean exp) is equivalent to the PASCAL while not boolean do exp.

3. The Source Language

The system takes as input a list of atoms, which are the names of the functions to be compiled. Each atom is associated with a λ-expression (via the EXPR indicator).

Definition: An atom is serious iff it is on the list of input atoms or it is the atom SEND. All other atoms are trivial. A form is serious if it contains an unquoted serious atom anywhere; otherwise it is trivial.

Each of the input expressions must satisfy the following restrictions:

(i) No serious atom may appear unquoted anywhere except in the function position of some form.

(ii) Except for COND, QUOTE, AND, and OR, every trivial function must evaluate all its arguments.

(iii) It may not contain free variables.

The following steps are performed by the preprocessor:

(i) COND, AND, and OR are converted to three-argument IFs: (IF predicate thenpart elsepart).

(ii) A few atoms used by the host medium (in particular SEND, VAL, QUIT, and GAMMA) are renamed throughout.

(iii) Embedded lambda-expressions in the function positions are eliminated by associating each such lambda-expression with a new serious function name, adding as arguments all parameters free in the lambda-expression, and then substituting an appropriate call in the form where the λ-expression appeared. For example the function body

```
     (LAMBDA (X) ((LAMBDA (Y) (FOO X Y Y)) (EXPENSIVE-FUNCTION X)))
```

is converted to

```
     (LAMBDA (X) (G X (EXPENSIVE-FUNCTION X)))
```

where G becomes associated with the function body

```
     (LAMBDA (X Y) (FOO X Y Y)).
```

4. The Main Transformation

The basic idea of this algorithm is to remove nontrivial recursions by the use of continuations. That is, to do $g(f(x),y)$, we do $f(x)$ and send the result z to a continuation which applies the function $\lambda z.g(z,y)$ to the value. In the context of this system we add to every serious function f a new argument, the continuation. $f_{new}(x,\gamma)$ should compute $f(x)$ and then apply the continuation γ to the value.

A continuation is therefore a function. Considerable effort has been made in the field of formal semantics attempting to get a good mathematical theory of continuation functions [23, 24, 28, 30]. From the programmer's point of view, however, the important issue is the representation of that functional object as a data structure [29, 31]. For any function represented as a data structure, one must have some application function which causes that data structure to be interpreted as a function.(*) This principle is very general:

(i) if the function is represented as hardware, the application function might be the LOAD-CARDS button.

(ii) if the function is represented as a bitstring, the application function might be an operating system CALL command [36].

(iii) if the function is represented as a lambda-expression, the application function is APPLY.

(*) A similar situation arises in "closed categories" [19, p. 180], where a function space Y^X is characterized by an application function $\epsilon : Y^X \times X \to Y$.

An obvious possibility is to represent continuation functions as lambda-expressions and to use APPLY. Unfortunately, because of dynamic binding, LISP must use a "retention strategy" [7], which uses up too much space. Furthermore, such a representation is needlessly general, since not every lambda-expression is a potential continuation. We shall see that for a given input program, all required continuations can be built from a finite number of continuation-builders. We will use this fact to create a special representation for continuations. In order to interpret this representation of continuations as functions, we must, as part of the transformation, write a function send(val,γ) which applies γ to the argument val.

We do not, however, wish to become immediately embroiled with representation issues. We therefore deal in terms of identities. An identity is a pair of forms. In an identity, atoms in the left-hand side which are not in the function position are pattern variables; all others are pattern literals which must match. Performing rewrites in the obvious way then gives a picture of the substitution semantics for a lambda calculus extended by the pattern matching capability.

Let us consider the simple example shown in Figure 4.1a, the fibonacci function. The first identity in Figure 4.1b shows what to do when presented with (FIB n γ). The goal, of course, is to compute fib(n) and send the result to γ. If $n \le 1$, then fib(n) is 1 and we can send 1 to γ immediately. Otherwise, we start to compute fib(n-2) with continuation (C_1 n γ) which will eventually carry out the rest of the calculation. Here C_1 is a trivial function which builds a data structure for the new continuation. It will be the job of send to decode the data structure built by C_1. Once the decoding (a representation-dependent operation) is completed, the analysis may continue, as follows:

The second identity says what to do if a value v_1 is sent to a continuation of the form $(C_1\ n\ \gamma)$. A continuation of this form is reached only when we have just finished computing fib(n-2) $(=v_1)$ in the computation of fib(n) with continuation γ. So we would like to compute v_1 + fib(n-1) and send the result to γ. To do this, we start to compute fib(n-1) with continuation $(C_2\ v_1\ \gamma)$ which will eventually finish the calculation.

The third identity shows what to do if a value v_2 is sent to a continuation of the form $(C_2\ v_1\ \gamma)$. A continuation of type C_2 is merely waiting for its second addend, and so it sends v_1+v_2 to γ.

This analysis is unsatisfying because it only tells how to build new continuations from old, but not how to get them started. We create the null continuation (represented by (QUIT)), and say that the result of sending any value to the null continuation is just that value. Thus a typical call to the new version of FIB would be (FIB 5 (QUIT)), which would yield the following derivation:

(FIB 5 (QUIT)) ==> (FIB 3 (C_1 5 (QUIT)))
==> (FIB 1 (C_1 3 (C_1 5 (QUIT))))
==> (SEND 1 (C_1 3 (C_1 5 (QUIT))))
==> (FIB 2 (C_2 1 (C_1 5 (QUIT))))

.

.

.

==> (SEND 2 (C_2 1 (C_1 5 (QUIT))))
==> (SEND 3 (C_1 5 (QUIT)))
==> (FIB 4 (C_2 3 (QUIT)))

.

.

.

==> (SEND 5 (C_2 3 (QUIT)))
==> (SEND 8 (QUIT))
==> 8

Note that at every state in the computation, the rewrite took place on the entire form and not on some subform; indeed, no subform ever became eligible for a rewrite. It is this property which makes "continuation form" felicitous for iterative calculations [29, p. 19].

Before we can state the algorithm for generating the identities we need some definitions.

Definition: A substitution is a function σ whose domain is some finite set of atoms (called the variables of σ and denoted vars(σ)), and whose range is the set of lists. If t is a form and σ is a substitution we define the form $t\sigma$ as follows:

$t\sigma = t$ if t is an atom and $t \notin$ vars(σ);

$t\sigma = \sigma(t)$ if $t \in$ vars(σ)

(QUOTE s)σ = (QUOTE s);

(f t_1 ... t_n)σ = (f $t_1\sigma$ ... $t_n\sigma$) f≠QUOTE.

Definition: The set of tail-recursive forms is defined as follows:

(i) If F is a serious function and $u_1,\ldots,u_n$ are trivial forms, then (F $u_1 \ldots u_n$) is tail-recursive.

(ii) If u is a trivial form and t_1 and t_2 are tail-recursive forms, then (IF u t_1 t_2) is tail-recursive.

(iii) Nothing else is tail-recursive.

Notice that all the right-hand sides of the identities of Figure 4.1b are tail-recursive.

The algorithm deals with a set Δ of identities which are successively transformed into identities where every right-hand side is tail-recursive. The transformations, as we will see, preserve the correctness of computations in a moderately straightforward way.

We initialize Δ as follows: for each serious function symbol F, with

$$F \underset{EXPR}{\equiv} (LAMBDA\ (x_1 \ldots x_n)\ rhs)$$

in the preprocessed source code, insert in Δ the identity

$$(F\ x_1\ \ldots\ x_n\ GAMMA) \equiv (SEND\ rhs\ GAMMA) \quad .$$

The initial set of identities may fail to be acceptable as output for two reasons:

(a) Calls on serious function symbols on the right hand side refer to the "old" versions, and therefore do not supply a continuation argument.

(b) A right-hand side may fail to be tail-recursive.

The algorithm proceeds to remedy this fault in a nondeterministic fashion. At each step, we transform a non-tail-recursive form (SEND exp cont) within a right-hand side by driving the SEND inward. When this process is complete, all the identities will be tail-recursive, and all calls on serious functions will supply a continuation argument. We proceed by cases. Notice (by induction) that the second argument to SEND will always be a trivial form.

Case I. (SEND exp cont) where exp is trivial. This is already tail-recursive, so no action required.

Case II. (SEND (F u_1 ... u_n) cont) where each u_i is trivial. Replace by (F u_1 ... u_n cont). This is correct, since the new F is supposed to compute (F u_1 ... u_n) and send the result to the continuation supplied.

Case III. (SEND (IF u t_1 t_2) cont) where u is trivial. Replace by (IF u (SEND t_1 cont) (SEND t_2 cont)).

Case IV. (SEND t cont) where t does not match Cases I, II, or III. In this case t is too complicated to be dealt with immediately, so we need to factor it into simpler subproblems. The general notion of factorization arises from category theory, where it is important in categorical systems theory [1]. The flavor of this notion which is appropriate for our purposes is given as follows:

Definition: A factorization of a form t is a 4-tuple $(t',\sigma,v,<)$ where t' is a form, σ is a substitution, and $v \in \underline{vars}(\sigma)$, such that

(i) $t = t'\sigma$

(ii) if $b \in \underline{vars}(\sigma)$ and $b \neq v$, then $b\sigma$ is trivial.

(iii) $<$ is a total order on $\underline{vars}(\sigma)$.

A typical factorization is shown in Figure 4.2. We think of the form t as a tree, which is factored into a top part t' and a "skirt" specified by σ, whose pieces are $v\sigma, b_1\sigma, \ldots, b_n\sigma$.

It is possible to reconstruct t by attaching the skirt to t' at the variables vars(σ): so $t = t'\sigma$. Furthermore the skirt has the property that at most one of its elements is a serious form. Practitioners of the λ-calculus [6] may recognize this as an inverse β-reduction, that is

$$((\lambda\ (v\ b_1\ ..\ b_n)\ t')\ v\sigma\ b_1\sigma\ ..\ b_n\sigma)$$

β-reduces to t. The ordering $<$ is significant only with respect to side-effects; we will discuss it shortly.

Given this factorization, our strategy for evaluating t is as follows: Since $b_1\sigma,..,b_n\sigma$ are all trivial, we compute them immediately, and store them in a continuation. We then start evaluating the (possibly) serious form $v\sigma$. Eventually the computation of $v\sigma$ will finish and send its result to our continuation, which will retrieve the values of $b_1\sigma,..,b_n\sigma$, and continue with the evaluation of t'.

To build this new continuation, we introduce a new continuation-builder (a trivial function) C_i:

Case IV. (SEND t cont) where t does not match Cases I, II, or III. Let $(t',\sigma,v,<)$ be a factorization of t with vars(σ) = $\{v,b_1,...,b_n\}$ and $b_1<...<b_n$. Let C_i be a new trivial function symbol. Replace by (SEND $v\sigma$ (C_i $b_1\sigma$ ... $b_n\sigma$ cont)) and add to Δ the new identity (SEND v (C_i b_1 ... b_n GAMMA)) $\equiv$ (SEND t' GAMMA).

The new identity with left-hand side (SEND v (C_i b_1 ... b_n GAMMA)) ensures that when evaluation of $v\sigma$ terminates and this continuation is resumed, it will go to work on the remaining portion of t -- namely, t'. Notice that (C_i b_1 .. b_n GAMMA) looks very much like a stack in which the topmost stack frame consists of the "return address" C_i and stacked data $b_1,..,b_n$.

This would work fine if it were not for conditionals and side-effects. For example, in

(IF (F X) (CAR X) X)

one may not stack (CAR X) since one does not know that X is nonatomic (i.e. that (CAR X) exists). In general, one may not stack any non-variable in a continuation arising from a conditional unless it is in the predicate part.

Definition: We define the relation $t << t'$ (read "t is performed unconditionally in t' ") to be the smallest transitive relation such that:

(i) if $t' = (\text{IF } t_1\ t_2\ t_3)$, then $t_1 << t'$

(ii) if $t' = (f\ t_1\ \ldots\ t_n)$, where $f \notin \{\text{IF},\text{QUOTE}\}$, then $t_i << t'$ for $1 \le i \le n$.

Definition: A factorization $(t',\sigma,v,<)$ of t admits conditionals iff if $b \in \text{vars}(\sigma)$, $b \neq v$, and $b\sigma$ is nonatomic, then $b\sigma << t$.

Another problem with which we must deal is that of side-effects. We assume that side-effects may be introduced only by trivial functions (e.g. READ, PRINT, etc.). Trivial forms may therefore be partitioned into those with side-effects and those without. We also assume that evaluation of arguments to a function proceeds left-to-right,(*) both in the source code and the object code. We then reach the following definition, which makes the pessimistic assumption that any serious function call may introduce side-effects:

Definition: A factorization $(t',\sigma,v,<)$ admits side-effects iff

(i) v occurs only once in t'

(ii) if $b \in$ vars(σ) and $b\sigma$ has side-effects, then b occurs only once in t'

(iii) if $b \in$ vars(σ) and $b\sigma$ has side-effects, then the unique occurrence of b is to the left of v in t', and

(iv) if $b,b' \in$ vars(σ), $b \neq v$, and $b\sigma$ and $b'\sigma$ both have side-effects, and b occurs to the left of b' in t', then $b<b'$.

(In this definition, "occurs" means "occurs unquoted," of course). When the code generated by Case IV is executed, the $b_i\sigma$ are executed in order, followed by $v\sigma$, followed by t'. These restrictions guarantee that the side-effects of this execution order are the same as that of the execution of t. Note that if we can guarantee, by flow analysis or other means, that $v\sigma$ never performs any side-effects, then condition (i) may be dropped; if, in addition,

---

(*) In a pure λ-calculus system, this assumption may be made unnecessary by using Currying or other techniques to explicitly control argument evaluation order.

$v\sigma$ always terminates, then condition (iii) may be dropped as well.

Last, we need to know that the factorization makes progress:

Definition: A factorization $(t',\sigma,v,<)$ is proper iff

(i) t' is nonatomic, and

(ii) for some $b \in$ vars(σ), $b\sigma$ is nonatomic.

Definition: A factorization is admissible iff it is proper, admits conditionals, and admits side-effects.

Now we obtain a correct algorithm by changing Case IV as follows:

Case IV': Same as Case IV, but require $(t',\sigma,v,<)$ to be admissible.

It is important to notice that there is at least one and usually more than one admissible factorization. In the following examples, let F and G be serious; since there are no side-effects, we omit "<".

(a) We can use factorization to eliminate common subexpressions. The form (G (IF (F (CAR X)) (CDR (CAR X)) (CDR X))) may be factored as:

t' = (G (IF (F B1) (CDR B1) (CDR X)))

σ ={(B1,(CAR X)),(X,X)}

v = X

(choosing v = B1 would also yield an admissible factorization).

(b) The usual factorization is leftmost-innermost, stacking only variables. The same example gives

t' = (G (IF V1 (CDR (CAR X)) (CDR X)))

σ ={(V1,(F (CAR X))),(X,X)}

v = V1

This factorization has the advantage that (SEND $v\sigma$ cont) is guaranteed to fall into Case II.

(c) Another factorization is leftmost-outermost, again stacking only variables. The example gives:

t' = (G V1)

σ ={(V1,(IF (F (CAR X)) (CDR (CAR X)) (CDR (CAR X)) (CDR X)))}

v = V1

This last alternative is the algorithm we chose to implement. It has the interesting property that several stack "frames" [4] may be created at once. An example of this is shown in Figure 4.4 where the first identity creates four frames. This saves the expense of repeatedly pushing and popping Y off the stack.

Either (b) or (c) may be modified to stack expressions instead of variables. Thus the usual trick of stacking the right subtree in a preorder traversal [15] falls within the range of admissible factorizations.

5. Translating for a host medium

The algorithm of Section 4 leaves us with a set of identities of the following forms:

$$(\mathrm{F}\ x_1 \ \ldots \ x_n \ \mathrm{GAMMA}) \equiv \mathrm{rhs}_{\mathrm{F}}$$

$$(\mathrm{SEND}\ v_i \ (\mathrm{C}_i \ b_1 \ \ldots \ b_n \ \mathrm{GAMMA})) \equiv \mathrm{rhs}_i$$

where each rhs is a tail-recursive form. In this section we discuss how these identities may be implemented on a garden-variety shallow-binding LISP system. We refer to these implementations as *host media* to distinguish them from implementations of the algorithm of Section 4.

In order to implement these identities in a host medium, we must first implement the patterns $(\mathrm{C}_i \ b_1 \ \ldots \ b_n \ \mathrm{cont})$ with associated constructors and decomposers. We choose to represent the pattern $(\mathrm{C}_i \ b_1 \ \ldots \ b_n \ \mathrm{cont})$ as a list of n+2 elements, the first of which is the atom Ci. Such a pattern may be decomposed by appropriate selectors. For example, if the form Q evaluates to an instance of the pattern $(\mathrm{C}_i \ b_1 \ \ldots \ b_n \ \mathrm{GAMMA})$, we could obtain the components using the substitution $\sigma^i_Q = \{(b_1, Q\nu_2), \ldots, (b_n, Q\nu_{n+1}), (\mathrm{GAMMA}, Q\nu_{n+2})\}$ where ν_i is the transformation on forms defined by

$$t\nu_1 = (\mathrm{CAR}\ t)$$

$$t\nu_{j+1} = (\mathrm{CDR}\ t)\nu_j$$

Then $b_j\sigma^i_Q = Q\nu_{j+1}$ is a form which selects the (j+1)-st element (that is, the value of b_j) from Q. We use suffixes consistently to denote transformations on forms.

Patterns may be built by associating a trivial function with each continuation-builder C_i:

$C_i \underset{\text{EXPR}}{\equiv}$ (LAMBDA (b_1 .. b_n GAMMA) (LIST (QUOTE C_i) b_1 .. b_n GAMMA))

and then β-reducing each call (C_i t_1 .. t_n cont).

5.1 Monadic Medium

An obvious implementation policy is to represent the entire state of the computation as a single list, as in the first example of Section 4.(*) We will then have a driver with an inner loop that looks like:

```
(REPEAT  UNTIL (EQ (CAR STATE) (QUOTE QUIT))
        (SETQ STATE (NEXT-STATE STATE)) )
```

At termination of this loop, the answer will be in (CAR(CDR STATE)). NEXT-STATE will be a function which looks at the state-list, determines which identity to apply, and constructs the new state, extracting components from the old state as necessary. For each serious function symbol F, with associated identity

(F x_1..x_n GAMMA) ≡ rhs_F

let $\xi_F = \{(x_1, \text{STATE}\nu_2), \ldots, (x_n, \text{STATE}\nu_{n+1}), (\text{GAMMA}, \text{STATE}\nu_{n+2})\}$;

ξ_F is the extractor substitution. For each continuation-builder C_i, with associated identity

---

(*) We call this <u>monadic</u> because it uses a single data object to represent the state.

(SEND v_i (C_i b_1 .. b_n GAMMA)) $\equiv rhs_i$

let $\delta_i = \{(v_i, \mathrm{STATE}\nu_2)\} \cup \sigma^{i}_{\mathrm{STATE}\nu_3}$; δ_i is the extractor substitution. We may now write a skeleton for NEXT-STATE as follows:

```
NEXT-STATE $\underset{\mathrm{EXPR}}{\equiv}$

(LAMBDA (STATE) (COND
   ((EQ (CAR STATE) (QUOTE $F_1$))$rhs_{F_1}\xi_{F_1}$§)
      .
      .
      .
   ((EQ (CAR STATE) (QUOTE $F_m$))$rhs_{F_m}\xi_{F_m}$§)
   ((EQ (CAR STATE) (QUOTE SEND))
    (COND
      ((EQ STATE$\nu_3\nu_1$) (QUOTE $C_1$)) $rhs_1\delta_1$§)
         .
         .
         .
      ((EQ STATE$\nu_3\nu_1$) (QUOTE $C_p$)) $rhs_p\delta_p$§)
      (T (ERROR UNKNOWN-CONTINUATION))))
   (T (ERROR UNKNOWN-FUNCTION-SYM))))
```

Here $F_1,..,F_m$ are the serious function symbols, and $C_1,..,C_p$ are the continuation-builders. To each rhs the appropriate extractor substitution is applied, and then the resulting form (still tail-recursive) is transformed by the state-building transformation §, which is defined (on tail-recursive forms only) as follows:

(i) (F u_1 .. u_n)§ = (LIST (QUOTE F) u_1 .. u_n) where F is serious (including SEND)

(ii) (IF t u_1 u_2)§ = (IF t u_1§ u_2§)

The code produced for FIB is shown in Figure 5.1. The continuation variable is just another component of the state and requires no special treatment.

the extractor substitution is

$$\rho^i = \sigma^i_{\text{THE-MESSAGE}\nu_2} \cup \{(v_i, \text{THE-MESSAGE}\nu_1)\}$$

The state-building transformation ¢ looks like:

$(G\ u_1 \ldots u_{n+1})$¢ produces code which performs the parallel assignment

THE-TARGET ← (QUOTE G)

THE-MESSAGE ← (LIST $u_1 \ldots u_{n+1}$)

and (IF u t_1 t_2)¢= (IF u t_1¢ t_2¢).

We next observe that the sequential search performed by SEND (the embedded COND in the monadic host) can be replaced by a similar use of the property list. This gives us the following code:

$F \underset{\text{SCRIPT}}{\equiv} rhs_F \eta_F$¢ for each serious function symbol except SEND

$\text{SEND} \underset{\text{SCRIPT}}{\equiv}$ (EVAL (GET THE-MESSAGE$\nu_2\nu_1$ (QUOTE SCRIPT)))

$C_i \underset{\text{SCRIPT}}{\equiv} rhs_i \rho_i$¢

The code for FIB is shown in Figure 5.2.

We have chosen the names for this host to accentuate the similarities with Hewitt's actors [13]. Here each serious function symbol is an actor which receives a message; upon receiving the message it follows its script and sends another message by leaving notes in the mail boxes (THE-TARGET and THE-MESSAGE); the driver then delivers the message by telling the new target it can find a message addressed to him in THE-MESSAGE. The same metaphor works for the continuation symbols, except that SEND sends them messages directly rather than through the driver loop.

5.3 Register Machine Host

In this host, we spread the state out still further. We store the serious function symbol as the value of the atom #PC# , and each parameter in the most convenient place -- namely as the value of the lambda-variables of the serious function symbol (VAL and GAMMA for SEND). This architecture is due to Sussman and Steele [29]. We store the code for determining the next state on the property list of each function symbol, under the INSTR property. Thus the inner loop of the driver will look like

```
     (REPEAT UNTIL (EQ #PC# (QUOTE QUIT))
            (EVAL (GET #PC# (QUOTE INSTR))) )
```

with the variable VAL containing the result upon completion of the loop.

Since the variables for the serious functions are already in their lambda-variables, no extractor substitution is necessary. For a continuation identity of the form

$$\text{(SEND } v_i \text{ (} C_i\ b_1 \text{ .. } b_n \text{ GAMMA))} \equiv \text{rhs}_i$$

the extractor substitution τ_i is $\sigma^i_{\text{GAMMA}} \cup \{(v_i, \text{VAL})\}$.

The state-building transformation \$ is defined as follows:

(i) $(G\ u_1 \ldots u_{n+1})\$$ is any code that performs the parallel assignment:

$$\begin{array}{l} \#PC\# \leftarrow \text{(QUOTE G)} \\ y_1 \leftarrow u_1 \\ \vdots \\ y_{n+1} \leftarrow u_{n+1} \end{array}$$

where the y_i's are G's λ-variables, and the u_i are evaluated left to right to ensure that side-effects are handled properly.

(ii) (IF u t_1 t_2)\$ = (IF u t_1\$ t_2\$) .

We then store the following code under the INSTR property:

SEND $\underset{\text{INSTR}}{\equiv}$ (EVAL (GET (CAR GAMMA) (QUOTE INSTR)))

F $\underset{\text{INSTR}}{\equiv}$ $\text{rhs}_F\$$

Ci $\underset{\text{INSTR}}{\equiv}$ $\text{rhs}_i\tau_i\$$

Here, SEND effectively does an assigned go-to, indexed by the continuation symbol, instead of using a sequential search.(*)

There is considerable freedom in choosing an implementation of (G u_1 .. u_{n+1})\$. The simplest solution is to make G an EXPR which performs the appropriate assignments (thus using the λ-bindings as temporaries), and setting

(G u_1 .. u_{n+1})\$ = (G u_1 .. u_{n+1}) .

Our solution is to generate a sequential assignment, using a topological sort to determine the order of assignments and to introduce temporaries (e.g. GCD in Figure 5.4). We use a function BLOCK to delimit such sequences of assignments.

If a function calls itself with one or more of its parameters unchanged, this algorithm generates code of the form (SETQ var var), which is promptly eliminated. In particular, the extra variables introduced by the preprocessor to eliminate free variables in an embedded lambda-expression never cause any assignments to be generated.

(*) A more precise analogue is the machine instruction "execute indirect."

Similarly, variable assignments of the form (SETQ #PC# (QUOTE G)) are eliminated when the value of #PC# is already G. This greatly reduces the number of assignments in tightly-looping functions. For example, in Figure 5.5, the first recursive call on ALLREMBER is implemented with a single assignment.

A similar phenomenon occurs when two or more functions share a lambda-variable. For example, all functions share the variable GAMMA; thus mutually tail-recursive function calls(*) and function returns via SEND need not generate any assignments for the continuation variable (eg, the terminal condition of ALLREMBER). Similar assignment elimination can occur on any variable.

If the LISP system in which the register machine is implemented resists compiling things other than EXPRs, it may be preferable to create a new symbol g for each INSTR-binding

$$\text{atom} \underset{\text{INSTR}}{\equiv} \text{code}$$

and instead bind as follows:

$$\text{atom} \underset{\text{INSTR}}{\equiv} (g)$$

$$g \underset{\text{EXPR}}{\equiv} (\lambda()\ \text{code}) \quad .$$

This makes compiling, tracing, etc possible. This idea is seen in Sussman and Steele [29].(†)

In order to simplify the user interface, the top-level code

(*) Like EVAL and APPLY in the LISP interpreter [18, Chap. 1].

(†) The interface can be made even smoother if something like MACLISP's FUNCALL is available.

reads a list of the form $(F\ a_1\ ..\ a_n)$ from the console, loads F into #PC#, loads (QUIT) into GAMMA, and loads the a_i into F's λ-variables. It then starts the inner loop.

The result of all this is a LISP system which runs iteratively on any reasonable LISP system. On our DEC-10 with a KL processor, without any compiled code, running time averages about 2.5 msec per step.

5.4 Optimizing Assignments in the Register Machine

The register machine offers considerable flexibility in the implementation of $, which is unavailable in either of the other hosts. In this section we will discuss a method for discovering opportunities for register-sharing at the preprocessor phase.

We construct an equivalence relation on the set of registers and write [v] for the equivalence class of v. With each equivalence class Q we associate a set

$$\underline{\text{users}}(Q) = \{F \mid F \text{ is a serious function symbol and } (\exists v \in Q)(v \text{ is a } \lambda\text{-variable of } F)\} .$$

We start by renaming all lambda-variables to eliminate duplications, and then set [v] = {v} for each lambda-variable v. For each form (F .. w ..) in the source code (where w is a variable) there is an opportunity to merge w and v, where v is the corresponding λ-variable of F. (This corresponds to the generation of (SETQ v w) in the register machine.) If the users([v]) are disjoint from the users([w]), then we merge [v] and [w]. At the end of the process, all equivalent registers are replaced by a single register. Thus each (SETQ v w) for which a merge took place would be eliminated. The order in which pairs are merged evidently makes a difference, and we do not claim that this gives a minimal number of SETQs (indeed, optimal code generation is NP-hard in most cases [3]).

If it is desired to minimize the total number of registers used, equivalence classes whose user sets have empty intersections may now be merged arbitrarily.

This algorithm assumes that all functions are global; more sophisticated sharing may be obtained by consideration of program localities and lifetimes of temporaries [27, 37].

6. An Application

In this section we discuss an implementation problem which was solved using the techniques of this paper. We consider the "suspending CONS" of [10]. Normally, CONS evaluates its arguments and allocates a cell in which the CAR and CDR fields point to the values of the two arguments. Instead of placing these final values in the node, a suspending CONS places in each field the unevaluated argument and the current environment in a distinguishable data structure called a suspension. CONS immediately returns with (a pointer to) the new cell. When either CAR or CDR is applied to that node, the argument is evaluated in the preserved environment. The resulting value takes the place of the suspension in the node and is returned.

The semantics of the suspending CONS is implemented by making a few changes to the LISP interpreter of [18, Chap. 1]. The CONS line is removed from APPLY and added in EVAL as follows:(*)

```
EVAL  ≡   (LAMBDA (E A) (COND         ;E is the form, A the environment
     EXPR
            ((ATOM (CAR E))
             (COND
              ((EQ (CAR E) (QUOTE CONS))
               (CONS (SUSPEND Eν_2 A)
                     (SUSPEND Eν_3 A)))
                  .
                  .
                  .
```

(*) We adopt the notation used in the rest of this paper.

Here SUSPEND is a trivial function which builds the "distinguishable data structure" and CONS is the old CONS of the defining language. We change the CAR line of APPLY as follows:

```
APPLY  ≡   (LAMBDA (FN X A) (COND                       ;FN is the function
      EXPR                                              ;X is the list of
          ((ATOM FN)                                    ;evaluated arguments
                                                        ;A is the call-time
           (COND                                        ;environment
                                                        ;Xν1 is the cell we are
            ((EQ FN (QUOTE CAR))                        trying to CAR on.

             (COND

              ((SUSPENDED? (CAR Xν1))                   ;is its CAR suspended?

               (RPLACA Xν1 (EVAL (FORM (CAR Xν1))       ;Yes-evaluate the sus-
                                                         pension
                                 (ENV (CAR Xν1)))))     ;and RPLACA it

              (T (CAR Xν1))))                           ;No-just grab it.
```

Here SUSPENDED? determines if a structure is suspended, and FORM and ENV extract the components from a suspension. The CDR line in APPLY is modified similarly, by changing each "CAR" to "CDR" (The ν_i's stay unchanged). Under the interpreter just described, a call to CAR runs until it finishes-- possibly forever. Let us try to put this evaluation under the programmer's control by supplying a primitive called COAXA. (COAXA L) is a predicate which returns true if the CAR of L is a suspension, with the side-effect that the evaluation of the suspension is forced to proceed some small amount. Given such a primitive we could write code like:

```
CHOOSE        ≡    (LAMBDA (L) (HELP-CHOOSE L L))
             EXPR
HELP-CHOOSE  ≡    (LAMBDA (PART WHOLE) (COND
            EXPR     ((NULL PART) (CHOOSE WHOLE))
                     ((COAXA PART) (HELP-CHOOSE (CDR PART) WHOLE))
                     (T (CAR PART))))
```

If L is a form which evaluates to a finite list, some of whose members may diverge, then (CHOOSE L) returns a convergent element of L, if there is one, by "kicking" each element of L with COAXA in a round-robin.

Now it turns out that COAXA is not a desirable addition; it quickly leads to pathological programs. It is, however, the first and simplest of a sequence of extensions which eventually led to a flexible facility for multisets and related structures [11]. We therefore consider the problem of implementing COAXA (and, of course, COAXD).

It seems reasonably clear that COAXA cannot be implemented by modifying this version of the interpreter, since one has no choice but to wait for EVAL to return a value. The techniques of this paper, however, give a good notion of "a small amount" - namely one step of the driver.

Let us, therefore, apply the main transformation to the modified interpreter and consider the monadic host, where NEXT-STATE is a well-defined notion. For notational convenience, let α and ϵ be the extractor substitutions for APPLY and EVAL (called ξ_{APPLY} and ξ_{EVAL} in Section 5.1).

The activation of a suspension in the CAR line of APPLY compiles into

```
(LIST (QUOTE EVAL)
      (FORM (CAR Xν_1))α
      (ENV (CAR Xν_1))α
      cont)
```

where cont is some continuation which takes care of the RPLACA, etc. If we intend that COAXA advance a suspension "one step", then the suspended computation might no longer be in an EVAL state; the suspension itself will have to indicate where it wants to be restarted. Furthermore, stepping through the computation might generate a local continuation, which must be maintained and interfaced as necessary to the continuation of the calling computation.

Thus we conclude that a suspension must contain a state; we therefore change SUSPEND to take a state-list as its argument and introduce STATE-COMP to extract state-list from a suspension. We change the code for CONS in the EVAL portion of NEXT-STATE. The original compilation yields:

```
(LIST (QUOTE SEND)
      (CONS
       (SUSPEND Eν_2ε Aε)
       (SUSPEND Eν_3ε Aε))
      GAMMA)
```

We change this to

```
(LIST (QUOTE SEND)
      (CONS
       (SUSPEND
        (LIST (QUOTE EVAL) Eν_2ε Aε (QUOTE (LOCALEND))))
       (SUSPEND
        (LIST (QUOTE EVAL) Eν_3ε Aε (QUOTE (LOCALEND)))))
      GAMMA)
```

where each suspension is equipped with the dummy continuation LOCALEND. We put code for COAXA in the APPLY portion of NEXT-STATE with the following COND-pair:

```
((EQ FNα (QUOTE COAXA))
 (COND
  ((SUSPENDED? (CAR Xν_1α))
   (PROG2(*)
         (HELP-COAXA Xν_1α
          (NEXT-STATE (STATE-COMP (CAR Xν_1α))))
         (LIST (QUOTE SEND) T GAMMAα)))
  (T (LIST (QUOTE SEND) NIL GAMMAα))))
```

with

```
HELP-COAXA  ≡     (LAMBDA (CELL STATE) (COND
           EXPR       ((LOCALEND?  STATE) (RPLACA CELL STATEν_2))
                      (T (RPLAC-STATE (CAR CELL) STATE)) ))
```

If the CAR is suspended, the state of the suspension is extracted and advanced one step by NEXT-STATE. If the resulting state is a terminal state of the local computation, of the form (SEND val (LOCALEND)), then the value is RPLACA'd into the cell. Otherwise, the state field of the suspension is updated (using RPLAC-STATE). Note that

---

(*) (PROG2 exp1 exp2) evaluates exp1 and exp2 in order and returns the value of exp2.

NEXT-STATE is now recursive! Luckily, it recurses down the structure of the state, which is never circular, so termination is assured. Last, we must fix the code for CAR and CDR. To do a CAR on a suspended structure, we restart the suspended calculation and let it run to completion; we then resume the global calculation. To do this we merely splice the continuation of the calling process (which takes care of the RPLACA) onto the end of the local continuation. The code for reactivating a suspension in the CAR line of the APPLY portion of NEXT-STATE, which was originally

```
(LIST (QUOTE EVAL)
      (FORM (CAR Xν1))α
      (ENV (CAR Xν1))α
      cont)
```

is changed to:

```
(APPEND (ALL-BUT-LAST† (STATE-COMP (CAR Xν1α)))
        (SPLICE
          (LAST† (STATE-COMP (CAR Xν1α)))
          GAMMAα))
```

with

```
SPLICE  ≡   (LAMBDA (LCONT GCONT) (COND
       EXPR  ((EQUAL LCONT (QUOTE (LOCALEND))) GCONT)
             ((NULL (CDR LCONT))
              (LIST (SPLICE (CAR LCONT) GCONT)))
             (T  (CONS (CAR LCONT)
                       (SPLICE (CDR LCONT) GCONT))) ))
```

A similar change is made in CDR.

---

†if L = $(a_1..a_n)$, (LAST L) = a_n and (ALL-BUT-LAST L) = $(a_1..a_{n-1})$.

By compiling the interpreter, we got to a level where "a computational step" was meaningful. (We could similarly introduce escape expressions [22] at this point.) The result is a recursive version of NEXT-STATE. We could let the LISP run-time stack handle this recursion, or we could compile the patched NEXT-STATE again to remove the recursion, using the tail-recursive driver

```
DRIVE   ≡   (LAMBDA (STATE) (COND
       EXPR    ((EQ (CAR STATE) (QUOTE QUIT)) (CAR (CDR STATE)))
               (T (DRIVE (NEXT-STATE STATE))) ))
```

The second compilation may be made with any host; if the register host is used, the code for DRIVE will look very much like the driver of Section 5.1.

7. Conclusions and relation to other work

The experimental programming system we have described is related to work on several current topics in programming. We classify these into four groups: recursion removal, compiling, semantics, and modular programming.

7.1 Recursion Removal

Our system performs automatic recursion removal on a wide class of LISP programs, using source-to-source transformations in the style of Knuth [15]. The techniques used are quite general and have been demonstrated on examples, such as a LISP interpreter, which are considerably larger than Knuth's "sturdy toddler." LISP extensions such as functional combination and starred functions [9] are also compilable.

Other work in this vein has been done by Auslander and Strong [2]. Much of their effort is devoted to overcoming the handicaps imposed by PL/1 as a source language -- imperative control structures, side-effects, declarations, scope of variables, etc. Consequently, their system is only semi-automatic. Our choice of LISP, with its flexible data structures, simple control structures and variable bindings, bypasses these roadblocks and allows us to obtain good performance with an automatic system. Our work is therefore complementary to theirs.

It is also worth noting that a naive approach to manual recursion removal in LISP would typically replace instances of CONS by APPEND or NCONC, which causes degradation of algorithm performance. Our system commonly improves the asymptotic space performance of the input program.

7.2 Compilers

The major innovation of this work is the use of factorizations and continuations as a sufficient criterion for the correctness of stack-manipulation regimes. In conventional language processors, one starts with a particular scheme for using the stack. By using the ideas of factorization, one may consider alternative schemes, e.g.

(1) stacking different sets of variables for different resumption points of a procedure.

(2) stacking expressions rather than variables.

(3) creating multiple frames (e.g. Figure 4.4).

The last alternative may be attractive in some cases since it trades stack space against the time for retrieving and restacking variables across function calls which do not involve them (as in Figure 4.4c).

This kind of analysis is useful even if local variables are maintained on the run-time stack. Both the registers for the register machine host and the temporary locations for trivial calculations may be allocated at the top of the stack. In this implementation, the order in which variables are allocated may be varied in order to decrease the amount of rearrangement necessary at continuation-building time.

The system itself is useful as an implementation tool for LISP systems. Because it produces code which will run iteratively even on LISP systems which do not properly handle tail-recursion or its variants, it is possible to run LISP programs which were formerly impractical to run because of their egregious stack usage -- recursive LISP interpreters, for example. Tail-recursion, which was heretofore discouraged in real LISP programs, may now be encouraged -- as it should be, since the compiled code runs in a tight, iterative loop.

7.3 Semantics

The system is an outgrowth of research into the relation between direct and continuation semantics. It emphasizes the notion of a continuation as a data-structure. Although this idea is implicit in [22] and mentioned explicitly in [23] (see also [25]), it has been obscured in the literature by alternative notions. Fischer [7] used continuations and a retention strategy to suppress procedure returns, thus eliminating the need for returning closures. Sussman and Steele [29] correctly point out that a continuation is merely a closure to be applied at some later point. Similarly, Tennent [30] speaks of "prophetic interpretations." Unfortunately, this picture of a continuation as a function requires lattice theory for a proper mathematical treatment [23,24, 28,30].

The contribution of category theory is the recognition that for a given program, all continuations needed for that program may be built as first-order objects, using only a finite number of continuation-builders. This avoids getting embroiled in lattice theory or anything else more complicated than list processing. The algorithm clearly works for any set of "trivial" functions which include list builders and decomposers. Although we have chosen not to emphasize it, this property gives the system a flavor of program schemata.

7.4 Modular Programming Systems

It has become almost conventional to describe computations in terms of a "frame" model [3], that is, a model in which the major object is a local state of the computation, possibly with pointers to other local state spaces, each describing a state of the computation, and a transition function on these packets.(*) Typically one specifies a set of packets sufficient to describe the computations of any program in a particular programming language [9, 11]. Such a model constitutes an interpreter for that language. Our system, given a LISP program, in effect produces a frame model tailor-made for that program.

Each frame (INSTR) is an independent entity to which control is passed unconditionally. This property is the origin of the

---

(*) Similar packet models, associated with assumptions about control structures in human information processing, have become prominent in artificial intelligence and natural language processing [20].

similarity between our object code and the concepts of "distributed computing" and "unidirectional message passing" advocated by Hewitt et. al [13]. Our system may therefore be viewed as a system for go-to introduction in the style of Knuth [15], except that our host machine's program counter is not sequential, so one may not "fall into" a label.(*)

(*) In other words, we have an n+1-address machine -- the program counter is always set and never incremented.

References

1. M. A. Arbib and E.G. Manes, *The Categorical Imperative*, Academic Press, 1974.
2. M. A. Auslander and H. R. Strong, Systematic Recursion Removal, IBM, Yorktown Heights, NY, Report RC5841, 1976.
3. D. G. Bobrow & B. Raphael, New Programming Languages for Artificial Intelligence Research , *Computing Surveys* *6*, 155-174 (1974).
4. D. Bobrow & B. Wegbreit, A Model and Stack Implementation of Multiple Environments, *Comm. ACM* *16* , 591-602 (1973).
5. J. Bruno and R. Sethi, Code Generation for a One-Register Machine, *J. ACM 23*, 502-510 (1976).
6. A. Church, *The Calculi of Lambda-Conversion*, Annals of Mathematics Studies, no. 6, Princeton University Press, Princeton, NJ (1941).
7. M. J. Fischer, Lambda-Calculus Schemata, Proc. ACM Conf. on Proving Assertions About Programs (Las Cruces, 1972), *SIGPLAN Notices* *7*,1 104-109, (January, 1972).
8. D. P. Friedman, *The Little LISPer*, Science Research Associates, Palo Alto, CA (1974).
9. D. P. Friedman and D. S. Wise, Functional Combination, *Computer Languages* *3*, 1 (1978).
10. D. P. Friedman and D. S. Wise, Cons should not evaluate its arguments, in S. Michaelson and R. Milner (eds.) *Automata, Languages,and Programming*, Edinburgh University Press, Edinburgh pp. 257-284 (1976).

References (Con't)

11. D. P. Friedman and D. S. Wise, Applicative Multiprogramming, Indiana Univ. Computer Science Department, Technical Report #72 (January, 1978).
12. W. Harrison, A new Strategy for Code Generation-- The General Purpose Optimizing Compiler, Conf. Rec. 4th ACM Symp. on Principles of Programming Langs. 29-37 (1977).
13. C. Hewitt, Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence 8, 323-364 (1977).
14. J. B. Johnston, The Contour Model of Block Structured Processes, Proc. ACM Symp. on Data Structures in Programming Languages, SIGPLAN Notices 6, 2, 55-81 (February, 1971).
15. D. E. Knuth, Structured Programming with Goto Statements, Computing Surveys 6, 261-301 (1974).
16. P. J. Landin, The Mechanical Evaluation of Expressions, Computer J. 6, 308-320 (1964).
17. D. B. Loveman, Program Improvement by Source-to-Source Transformation J. ACM 24, 121-145 (1977).
18. J. McCarthy et. al., LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Mass. (1965).
19. S. MacLane, Categories for the Working Mathematician, Springer-Verlag, New York (1971).
20. M. Minsky, A Framework for Representing Knowledge, in The Psychology of Computer Vision (Winston, ed.), McGraw-Hill, New York, pp. 211-277 (1975).
21. J. Moses, The Function of FUNCTION in LISP, SIGSAM Bulletin 15, 13-27 (July, 1970).
22. J. C. Reynolds, Definitional Interpreters for Higher-Order Programming Languages, Proc. ACM Nat'l Conf., 717-740 (1972).

References (Con't)

23. J. C. Reynolds, On the Relation between Direct and Continuation Semantics, Proc. 2nd Colloq. on Automata, Languages, and Programming, Saarbrucken (1974).

24. J. C. Reynolds, Semantics of the Domain of Flow Diagrams, J. ACM 24, 484-503 (1977).

25. B. Russell, On an Equivalence between Continuation and Stack Semantics Acta Informatica 8, 113-124 (1977).

26. R. Sethi and J. D. Ullman, The Generation of Optimal Code for Arithmetic Expressions, J.ACM, 17, 715-728 (1970).

27. G. L. Steele, LAMBDA: The Ultimate Declarative, Mass. Inst. of Tech., AI Memo 379 (October, 1976).

28. C. Strachey and C. P. Wadsworth, Continuations: A Mathematical Semantics for Handling Full Jumps, Oxford University Computing Laboratory, Technical Monograph PRG-11 (January, 1974).

29. G. J. Sussman and G. L. Steele, Jr., SCHEME: An Interpreter for Extended Lambda Calculus, Mass. Inst. of Tech., AI Memo 349 (December, 1975).

30. R. D. Tennent, Denotational Semantics of Programming Languages, Comm. ACM 19, 437-453 (1976).

31. M. Wand, Continuation-Based Program Transformation Strategies, J.ACM, to appear.

32. M. Wand, Final Algebra Semantics and Data Type Extensions, Indiana University Computer Science Department, Technical Report #65 (July, 1977).

References (Con't)

33. M. Wand, Algebraic Theories and Tree Rewriting Systems, Indiana Univ. Computer Science Department, Technical Report #66 (July, 1977).

34. C. Weissman, LISP 1.5 Primer, Dickenson Publishing Co., Encino, CA (1966).

35. D. S. Wise, D. P. Friedman, S. C. Shapiro, and M. Wand, Boolean-Valued Loops, BIT 15, 431-451 (1975).

36. W. Wulf et.al., HYDRA: The Kernel of a Multiprocessor Operating System, Comm. ACM 17, 337-345 (1974).

37. W. Wulf, R. K. Johnson, C. B. Weinstock, S. O. Hobbs, and C. M. Geshke, The Design of an Optimizing Compiler, American Elsevier, New York (1975).

```
(DEFPROP FIB
 (LAMBDA(N)
  (IF (LESSP N 1)
      1
      (IF (EQUAL N 1)
          1
          (PLUS (FIB (DIFFERENCE N 2)) (FIB (DIFFERENCE N 1))))))
EXPR)
```

```
(DEFPROP IDENTITIES
 (((FIB N GAMMA)
   (IF (LESSP N 1)
       (SEND 1 GAMMA)
       (IF (EQUAL N 1)
           (SEND 1 GAMMA)
           (FIB (DIFFERENCE N 2) (C1 N GAMMA)))))
  ((SEND V1 (C1 N GAMMA)) (FIB (DIFFERENCE N 1) (C2 V1 GAMMA)))
  ((SEND V2 (C2 V1 GAMMA)) (SEND (PLUS V1 V2) GAMMA)))
IDENT)
```

Figure 4.1   Preprocessed source code and identities for FIB

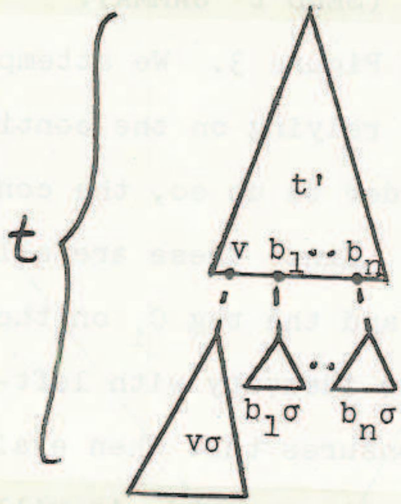

t
t'
v b_1..b_n
$v\sigma$
$b_1\sigma$
$b_n\sigma$

Figure 4.2 A factorization of t. Here $\mathrm{vars}(\sigma)=\{v,b_1,..,b_n\}$

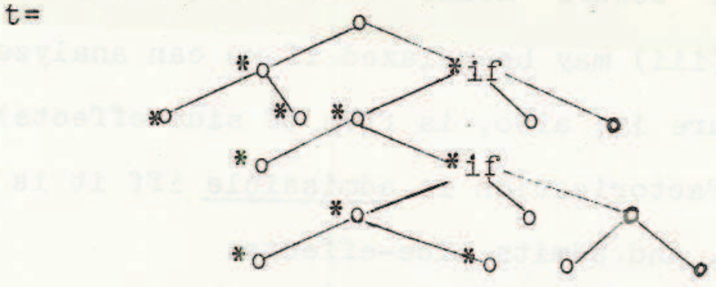


Figure 4.3 The starred nodes delimit the subtrees evaluated unconditionally in t.

```
(DEFPROP F
 (LAMBDA (X Y) (G (H (H (H (H X)))) Y))
EXPR)

(DEFPROP G
 (LAMBDA (X Y) (QUOTE GSTUB))
EXPR)

(DEFPROP H
 (LAMBDA (X) (QUOTE HSTUB))
EXPR)
```

(a) Source code

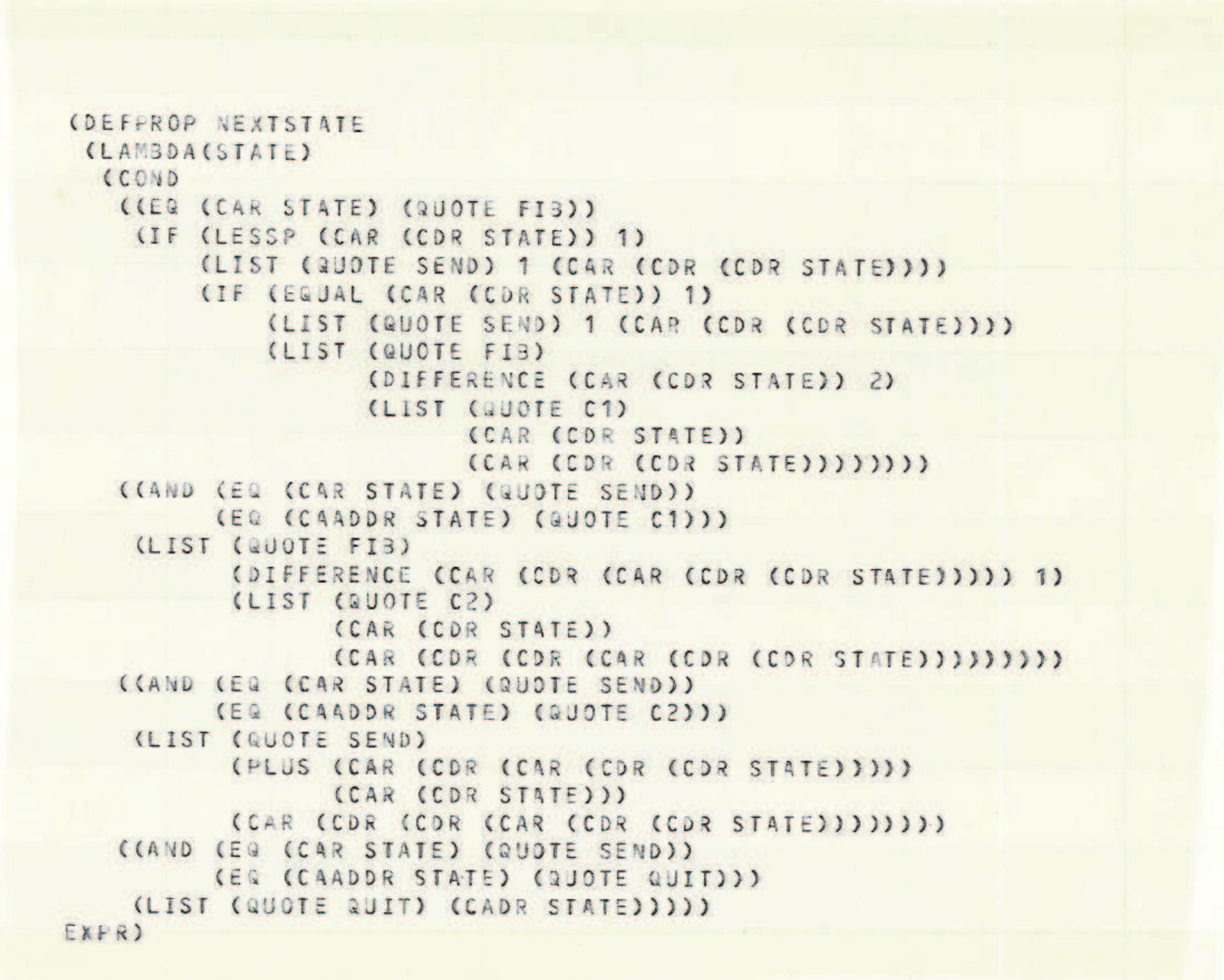
```
(DEFPROP IDENTITIES
 (((F X Y GAMMA) (H X (C1 (C2 (C3 (C4 Y GAMMA))))))
  ((G X Y GAMMA) (SEND (QUOTE GSTUB) GAMMA))
  ((H X GAMMA) (SEND (QUOTE HSTUB) GAMMA))
  ((SEND V1 (C1 GAMMA)) (H V1 GAMMA))
  ((SEND V2 (C2 GAMMA)) (H V2 GAMMA))
  ((SEND V3 (C3 GAMMA)) (H V3 GAMMA))
  ((SEND V4 (C4 Y GAMMA)) (G V4 Y GAMMA)))
IDENT)
```

(b) Nested continuation version

```
(DEFPROP IDENTITIES
 (((F X Y GAMMA) (H X (C1 Y GAMMA)))
  ((G X Y GAMMA) (SEND (QUOTE GSTUB) GAMMA))
  ((H X GAMMA) (SEND (QUOTE HSTUB) GAMMA))
  ((SEND V1 (C1 Y GAMMA)) (H V1 (C2 Y GAMMA)))
  ((SEND V2 (C2 Y GAMMA)) (H V2 (C3 Y GAMMA)))
  ((SEND V3 (C3 Y GAMMA)) (H V3 (C4 Y GAMMA)))
  ((SEND V4 (C4 Y GAMMA)) (G V4 Y GAMMA)))
IDENT)
```

(c) Iterated Continuation Version

Figure 4.4 Creation of multiple frames



```
(DEFPROP NEXTSTATE
 (LAMBDA(STATE)
  (COND
   ((EQ (CAR STATE) (QUOTE FIB))
    (IF (LESSP (CAR (CDR STATE)) 1)
        (LIST (QUOTE SEND) 1 (CAR (CDR (CDR STATE))))
        (IF (EQUAL (CAR (CDR STATE)) 1)
            (LIST (QUOTE SEND) 1 (CAR (CDR (CDR STATE))))
            (LIST (QUOTE FIB)
                  (DIFFERENCE (CAR (CDR STATE)) 2)
                  (LIST (QUOTE C1)
                        (CAR (CDR STATE))
                        (CAR (CDR (CDR STATE)))))))))
   ((AND (EQ (CAR STATE) (QUOTE SEND))
         (EQ (CAADDR STATE) (QUOTE C1)))
    (LIST (QUOTE FIB)
          (DIFFERENCE (CAR (CDR (CAR (CDR (CDR STATE))))) 1)
          (LIST (QUOTE C2)
                (CAR (CDR STATE))
                (CAR (CDR (CDR (CAR (CDR (CDR STATE)))))))))
   ((AND (EQ (CAR STATE) (QUOTE SEND))
         (EQ (CAADDR STATE) (QUOTE C2)))
    (LIST (QUOTE SEND)
          (PLUS (CAR (CDR (CAR (CDR (CDR STATE)))))
                (CAR (CDR STATE)))
          (CAR (CDR (CDR (CAR (CDR (CDR STATE))))))))
   ((AND (EQ (CAR STATE) (QUOTE SEND))
         (EQ (CAADDR STATE) (QUOTE QUIT)))
    (LIST (QUOTE QUIT) (CADR STATE)))))
EXPR)
```

Figure 5.1 Monadic host code for FIB

```
(DEFPROP FIB
 (IF (LESSP (CAR MESSAGE) 1)
     (PROG2 (SETQ TARGET (QUOTE SEND))
            (SETQ MESSAGE (LIST 1 (CAR (CDR MESSAGE)))))
     (IF (EQUAL (CAR MESSAGE) 1)
         (PROG2 (SETQ TARGET (QUOTE SEND))
                (SETQ MESSAGE (LIST 1 (CAR (CDR MESSAGE)))))
         (PROG2 (SETQ TARGET (QUOTE FIB))
                (SETQ MESSAGE
                      (LIST (DIFFERENCE (CAR MESSAGE) 2)
                            (LIST (QUOTE C1)
                                  (CAR MESSAGE)
                                  (CAR (CDR MESSAGE))))))))
SCRIPT)



(DEFPROP SEND
 (EVAL (GET (CAADR MESSAGE) (QUOTE SCRIPT)))
SCRIPT)



(DEFPROP C1
 (PROG2 (SETQ TARGET (QUOTE FIB))
        (SETQ MESSAGE
              (LIST
               (DIFFERENCE (CAR (CDR (CAR (CDR MESSAGE)))) 1)
               (LIST (QUOTE C2)
                     (CAR MESSAGE)
                     (CAR (CDR (CDR (CAR (CDR MESSAGE))))))))))
SCRIPT)



(DEFPROP C2
 (PROG2 (SETQ TARGET (QUOTE SEND))
        (SETQ MESSAGE
              (LIST
               (PLUS (CAR (CDR (CAR (CDR MESSAGE)))) (CAR MESSAGE))
               (CAR (CDR (CDR (CAR (CDR MESSAGE)))))))))
SCRIPT)
```

Figure 5.2 Actor host code for FIB

```
(DEFPROP FIB
 (IF (LESSP N 1)
     (BLOCK (SETQ #PC# (QUOTE SEND)) (SETQ VAL 1))
     (IF (EQUAL N 1)
         (BLOCK (SETQ #PC# (QUOTE SEND)) (SETQ VAL 1))
         (BLOCK (SETQ TEMP1 (DIFFERENCE N 2))
                (SETQ GAMMA (LIST (QUOTE C1) N GAMMA))
                (SETQ N TEMP1))))
INSTR)



(DEFPROP SEND
 (EVAL (GET (CAR GAMMA) (QUOTE INSTR)))
INSTR)



(DEFPROP C1
 (BLOCK (SETQ #PC# (QUOTE FIB))
        (SETQ N (DIFFERENCE (CAR (CDR GAMMA)) 1))
        (SETQ GAMMA (LIST (QUOTE C2) VAL (CAR (CDR (CDR GAMMA))))))
INSTR)



(DEFPROP C2
 (BLOCK (SETQ VAL (PLUS (CAR (CDR GAMMA)) VAL))
        (SETQ GAMMA (CAR (CDR (CDR GAMMA)))))
INSTR)
```

Figure 5.3   Register host code for FIB

```
(DEFPROP GCD
 (LAMBDA(X Y)
  (IF (EQUAL X Y)
      X
      (IF (GREATERP X Y) (GCD (DIFFERENCE X Y) Y) (GCD Y X))))
EXPR)
```

```
(DEFPROP IDENTITIES
 (((GCD X Y GAMMA)
   (IF (EQUAL X Y)
       (SEND X GAMMA)
       (IF (GREATERP X Y)
           (GCD (DIFFERENCE X Y) Y GAMMA)
           (GCD Y X GAMMA)))))
IDENT)
```

```
(DEFPROP GCD
 (IF (EQUAL X Y)
     (BLOCK (SETQ #PC# (QUOTE SEND)) (SETQ VAL X))
     (IF (GREATERP X Y)
         (BLOCK (SETQ X (DIFFERENCE X Y)))
         (BLOCK (SETQ TEMP1 Y) (SETQ Y X) (SETQ X TEMP1))))
INSTR)
```

Figure 5.4   Register host code for GCD

```
(DEFPROP ALLREMBER
 (LAMBDA(A L)
  (IF (NULL L)
      NIL
      (IF (EQ (CAR L) A)
          (ALLREMBER A (CDR L))
          (CONS (CAR L) (ALLREMBER A (CDR L))))))
 EXPR)
```

```
(DEFPROP IDENTITIES
 (((ALLREMBER A L GAMMA)
   (IF (NULL L)
       (SEND NIL GAMMA)
       (IF (EQ (CAR L) A)
           (ALLREMBER A (CDR L) GAMMA)
           (ALLREMBER A (CDR L) (C1 L GAMMA)))))
  ((SEND V1 (C1 L GAMMA)) (SEND (CONS (CAR L) V1) GAMMA)))
IDENT)
```

```
(DEFPROP ALLREMBER
 (IF (NULL L)
     (BLOCK (SETQ #PC# (QUOTE SEND)) (SETQ VAL NIL))
     (IF (EQ (CAR L) A)
         (BLOCK (SETQ L (CDR L)))
         (BLOCK (SETQ TEMP1 (CDR L))
                (SETQ GAMMA (LIST (QUOTE C1) L GAMMA))
                (SETQ L TEMP1))))
INSTR)
```

```
(DEFPROP C1
 (BLOCK (SETQ VAL (CONS (CAR (CAR (CDR GAMMA))) VAL))
        (SETQ GAMMA (CAR (CDR (CDR GAMMA)))))
INSTR)
```

Figure 5.5 Code for ALLREMBER