A Combinator-based Compiler for a Functional Language

Paul Hudak and David Kranz

Yale University

Department of Computer Science

1. Introduction

Functional languages are known primarily for their elegance, clarity, and expressive power. Unfortunately, this same elegance and power has been a bottleneck to building efficient implementations for such languages, especially those with fully "lazy" evaluation semantics. David Turner in his seminal paper [19] showed how the language SASL, a functional language with lazy evaluation, could be implemented simply and elegantly using *combinators*; yet such an implementation on conventional machines seems to be unbearably slow, as do implementations based on Landin's SECD machine, as described by Peter Henderson [6]. There has seemed to be an inherent inefficiency in implementing such languages, and few practical functional programming systems have been built.

From the outset the goal of our research has been to dispel this belief. We wish to demonstrate that a functional language with the full semantic power of lazy evaluation can be implemented efficiently with an appropriate optimizing compiler. Furthermore, we wish to show that combinators offer a convenient intermediate language with which to perform program transformations and optimizations. Through the use of combinators, traditional optimizations such as constant-folding and *global* common-subexpresssion elimination become trivial, and we are able to eliminate other sources of inefficiency that conventional compilers do not typically deal with. In particular, all procedure boundaries are eliminated from a program, leaving a single combinator expression that can be manipulated at will. One implication of this is that there is almost no penalty to the programmer for building modular, hierarchical programs that agree with "good design practice". Data may be "packaged" into lists for passing between procedures, with no run-time overhead. Once the single combinator expression is created, our compiler creates its own "optimized" procedures that do not necessarily correspond to those defined in the source program.

Further optimizations are possible to overcome the shortcomings of lazy evaluation. Normal-order graph reduction is able to accomplish lazy evaluation quite naturally, at the expense of incurring an abundance of heap manipulations; more conventional implementations require the use of "thunks", "closures", or some equivalent object. Although lazy evaluation increases the expressive power of a language, most applications do not require it. Our compiler takes advantage of this fact: through a relatively simple analysis we are able to determine most cases where an argument to a function is "strict", and may thus be computed prior to invoking the function. When necessary, the use of *self-modifying thunks* ensures us that the overhead of non-strict arguments is kept to a minimum. The use of these special objects also effectively accomplishes code-motion from "loops".

In the next section we provide an overview of ALFL, the source language of our compiler. In Section 3 we discuss Phase I of the compiler, which includes the details of our abstraction and simplification algorithms. Phase II, optimization and code-generation, is discussed in Section 4. The results of our work are presented in Section 5, and in Section 6 we discuss the relationship of our work to others.

This research was supported in part by NSF Grant MCS-8302018 and ONR Grant 87999.

2. ALFL: an Experimental Functional Language

ALFL is a block-structured, lexically-scoped functional language with lazy evaluation semantics; it is similar in style to Keller's FEL [8] and Turner's SASL [18] (with most influence from the former). Although ALFL is not the central issue in this paper, it is described here to provide a framework for our compilation techniques.

The largest syntactic object in ALFL is an *equation group*, which is delimited by curly brackets ("{...}"). Within an equation-group is a collection of *equations* that define local identifiers, and a single *result clause* that expresses the value to which the equation-group will evaluate. A double equal-sign ("==") is used in equations to distinguish it from the infix operator for equality. An equation-group is a special case of an *expression*, and is thus valid wherever an expression is allowed. A conditional expression has the form **"pred → cons, alt"** and is equivalent to the more conventional **"if pred then cons else alt"**. Here is a simple example:

```
{ fac n == n=0 → 1, n*fac(n-1);
  x == 10;
  result fac x }
```

ALFL scoping rules are similar to those for most block-structured languages, in that expressions may reference any identifier defined locally in the current equation-group, or in any surrounding equation-group. However, local references are allowed to be mutually recursive. This implies that equations may appear in any order, consistent with lazy evaluation in which expressions are evaluated "by demand". In the above example, the two equations plus the result clause could appear in any of six different orders.

All function applications are "curried" [5, 13]. That is, all functions are assumed to take just one argument, which is no restriction since that function may return another function that takes one argument, etc. If we define function application to associate to the left, curried functions facilitate the use of higher-order functions, as in:

```
{ twice f x == f (f x);
  twofacs == twice fac;
  result twofacs 10 }
```

which is precisely equivalent to:

```
{ twice f == { result g;
               g x == f (f x) }
  twofacs == twice fac;
  result twofacs 10 }
```

ALFL also has a powerful *pattern-matcher* through which complex functions may be defined more easily. For example, factorial could have been defined by:

```
{ fac 0 == 1;
  fac n == n*fac(n-1); ... }
```

The pattern-matcher is even more useful when used with *lists*. Lists are constructed "lazily", and are denoted by "[x,y,z]", which is equivalent, using the infix consing operator "^", to "x^y^z^[]". The selector functions are **hd** ("head") and **tl** ("tail"), corresponding to LISP's CAR and CDR, respectively. The pattern-matcher knows about lists; for example, the function **member** may be defined by:

```
{ member x [] == false;
  member x (x^L) == true;
  member x L == member x (tl L);
  result member 2 [1,2,3] }
```

When using the pattern-matcher note that the order of equations *defining the same function* does matter.

"Infinite lists" are defined in the obvious way. The infinite stream of numbers starting at **n**, for example, can be defined by **"numsfrom n == n^numsfrom(n+1)"**. As a more interesting example, consider this definition of the Fibonacci sequence:

```
{ fib == 1^1^addstreams[fib,tl fib];
  addstreams[x^S1,y^S2] ==
        (x+y)^addstreams[S1,S2]; ... }
```

Of course, elements of an infinite list are not computed until they are selected ("demanded") for evaluation. The infix operator "^^" is used to append lists together, and it too is lazy. Thus one can even append infinite lists together, as in **"fib^^fib"**; the second infinite stream of Fibonacci numbers is simply never reached!

There are other syntactic and semantic features of ALFL, but they are beyond the scope of this paper. Our purpose here is to give a framework on which to base later examples, in addition to pointing out the interesting semantic issues of lazy evaluation, so that the reader may appreciate the implementation problems thus imposed.

3. Phase I: Parsing, Abstraction and Simplification

In this section we describe Phase I of our compiler, which parses the ALFL source program, simultaneously removing all bound variables, and returning in a *single pass* a partially optimized combinator expression. Phase II consists of four passes and is described in the next section.

3.1. The Abstraction Algorithm

Abstraction is the means by which free variables are removed from an expression. Let **"[x]exp"** denote the

abstraction of free variable **x** from expression **exp**. An abstraction process is valid if it obeys the **law of abstraction** which states that **([x]exp) x = exp**. In the lambda calculus **[x]exp** = λ**x.exp**, which is consistent with the law of abstraction through the definition of β-reduction. In the combinatory calculus [5]:

[x] x = **I**
[x] y = **K y**, where **y** is a constant or variable other than **x**
[x] (e1 e2) = **S ([x]e1) ([x]e2)**, where **e1** and **e2** are arbitrary combinations

which is consistent with the law of abstraction through the definitions of **S**, **K**, and **I**, which are primitive functions called *combinators*. Their definitions, or *reduction rules* are:

I x $\Rightarrow$ **x**
K x y $\Rightarrow$ **x**
S f g x $\Rightarrow$ **f x (g x)**

where, as in ALFL, function application associates to the left.

Turner describes in [20] an abstraction algorithm for SASL, a functional language similar to ALFL. This algorithm is deficient for our purposes for two reasons:

1. It does not explicitly handle the lexical scoping required when nesting equation-groups to an arbitrary depth.
2. On mutually recursive equations the method generates a combinator expression that fails to terminate when evaluated.

The latter problem lies in the use of a special combinator **U** to handle mutual recursion.[1] It turns out that no special combinator is needed, and the non-termination problem can be avoided.

Our abstraction algorithm works for any block-structured, lexically-scoped, functional language. The pattern-matcher first combines all equations defining the same function into a single equation. For example, the definition of **member** given earlier is transformed into:

```
member x1 x2 == x2=[] → false,
                x1=(hd x2) → true,
                member x1 (tl x2)
```

where **x1** and **x2** are new, unique identifiers.

Now consider the following equation group in which equations defining the same function have been combined, and in which there are no nested equation groups:

$\{ f_1\ arg_{11}\ arg_{12} \cdots arg_{1m_1} == exp_1;$
$f_2\ arg_{21}\ arg_{22} \cdots arg_{2m_2} == exp_2;$
$\cdots$
$f_n\ arg_{n1}\ arg_{n2} \cdots arg_{nm_n} == exp_n;$
result E $\}$

There are **n** equations, each equation **i** having m_i arguments (m_i may be zero), and having definition exp_i. The abstractor first *normalizes* each equation **i** by abstracting each argument arg_{ix} (but not f_i) from exp_i. For example, for f_1 this yields:

$$f_1 = [arg_{11}]([arg_{12}](\cdots([arg_{1m_1}]exp_1)))$$

Letting E_i denote the result of normalizing equation **i**, then once all equations have been normalized we have:

$\{ f_1 == E_1$
$f_2 == E_2$
$\cdots$
$f_n == E_n$
result E $\}$

Note that a free variable in an expression E_i can only be one of the variables f_1 through f_n, or any variable that is free in the entire equation-group.

Next, the abstractor eliminates the equations one at a time. That is, each variable f_i is abstracted out of the entire equation-group, replacing it with an equivalent to f_i. Here is the result of eliminating f_1:

$\{ f_2 == ([f_1]E_2)\ E_1'$
$\cdots$
$f_n == ([f_1]E_n)\ E_1'$
result $([f_1]E)\ E_1'$ $\}$

where $E_1' = Y\ ([f_1]E_1)$.[2] **Y**, of course, is the fixpoint combinator defined by **Y f = f (Y f)**. This process is repeated until all equations are eliminated and a single result clause remains.

Now consider a program containing equation-groups nested to an arbitrary depth. If the equation-groups are abstracted from the inner-most group out (i.e., "bottom-up"), the lexical scoping rules are automatically preserved. This is because an identifier is bound to the inner-most surrounding equation-group in which it is

---

[1] In Turner's scheme the special combinator **U** is defined so that **U f (x^y) $\Rightarrow$ f x y**, and abstraction of a list from an expression is defined to be:

[x^y] E = U ([x] ([y] E))

To see why Turner's scheme fails to terminate, consider a program **E where f = e_1, g = e_2**. Turner's algorithm first combines the equations into a list, then abstracts the list as defined above. That is:

E where f^g = e1^e2 $\Rightarrow$
([f^g] E) (Y ([f^g] (e1^e2))) $\Rightarrow$
U ([f] ([g] E)) (Y (U ([f] ([g] (e1^e2)))))

Writing this last expression as **U E' (Y (U e'))** and applying the reduction rule for **U** it is seen that **Y (U e')** must be reduced first. Doing so yields **U e' (Y (U e'))**, but now the reduction rule for **U** tells us that **Y (U e')** must be reduced again! This process clearly does not terminate. If one insists on using this technique it seems necessary to introduce another version of **Y** that evaluates its arguments in a different order to guarantee termination.

[2] In our implementation all of the occurrences of E_1' are shared.

defined. The reader may verify that the overall algorithm obeys the law of abstraction by reversing each step to be sure that the original expression results. Note that:

1. Mutual recursion is handled naturally as the equations are sequentially eliminated.
2. If $\mathbf{f_1}$ is not recursive (i.e., $\mathbf{f_1}$ is not free in $\mathbf{E_1}$), then $\mathbf{E_1' = Y\ (K\ E_1) = E_1}$. Similarly, if $\mathbf{f_1}$ is not free in $\mathbf{E_i}$, $\mathbf{i \neq 1}$, then $\mathbf{f_i = ([f_1]E_i)\ E_1' = K\ E_i\ E_1' = E_i}$.
3. The above steps are always taken, even for equations with no arguments, since the equation might still be recursive. For example, the equation "**x == 1^x**" is recursive, and well-defined (i.e., the infinite list of ones).

In our implementation an additional technique is used to improve the performance of the abstractor: a *free-variable list* is kept for each subexpression. Thus given a very large expression **E**, if **x** is not contained in the free-variable list for **E**, then **[x]E** immediately returns **K E**. Without the free-variable list, **x** would need to be abstracted from *every* subexpression of **E**, and many simplifications would be required before the final result **K E** is returned.

3.2. Simplifications and Optimizations

Given a combinator (or lambda) expression, several reducible subexpressions (called *redexes* [17]) may appear at once. There are two standard ways of choosing which to reduce first. In *applicative-order* evaluation the *inner-most* redexes are reduced first. This corresponds roughly to the call-by-value function-calling mechanisms used by most Algol-like languages, where the arguments to a function are computed prior to invoking the function. In *normal-order* evaluation the *left-most* redex is reduced first, meaning that outer redexes are reduced before inner ones. It can be shown that some programs will terminate under normal-order evaluation, but not under applicative-order,[3] but any terminating applicative-order evaluation will also terminate under normal-order. It is thus often said that the *expressive power* of normal-order evaluation is greater than that of applicative-order. Lazy evaluation corresponds roughly to normal-order evaluation, but generally carries the additional connotation that no subexpression is reduced more than once -- this has also been called "fully lazy evaluation" [7].[4]

Turner describes a method for evaluating a combinator expression by normal-order *graph-reduction* [19], which results in a fully lazy implementation. The technique has certain "self-optimizing properties", such as constant-folding and code-motion from "loops". Our goal is to accomplish as many of these same optimizations as possible *prior* to program execution -- the machine code that we eventually generate will then benefit accordingly. As the source text is parsed in recursive-descent form, combinator expressions are created from the inner-most expression out. Rather than wait until run-time, these expressions are simplified *as they are constructed*, using the following rules:

1. **S K** ⇒ **K I**
2. **S (K I)** ⇒ **I**
3. **S (K (K x))** ⇒ **K (K x)**
4. **S (K x) I** ⇒ **x**
5. **S (K x) (K y)** ⇒ **K (x y)**
6. **S f g x** ⇒ **f x (g x)**
7. **K x y** ⇒ **x**
8. **I x** ⇒ **x**
9. **Y (K x)** ⇒ **x**
10. **binop c1 c2** ⇒ **z**, where **c1** and **c2** are constants, **z** = **c1 binop c2**, and **binop** is one of +, *, -, /, etc.
11. **typepred ob** ⇒ **bool**, where **ob** is an object whose type can be inferred, and **bool** is the result of applying **typepred** (one of **pair?**, **integer?**, **string?**, etc.) to **ob**.
12. **EQ x y** ⇒ **true**, if **x** is the same object as **y**, or if they have the same value.
13. **EQ x y** ⇒ **false**, if **x** and **y** are of different types, or if they have the same type but different values.
14. **NOT true** ⇒ **false, NOT false** ⇒ **true**
15. **AND true** ⇒ **I, AND false** ⇒ **K false**
16. **AND x true** ⇒ **x, AND x false** ⇒ **false**
17. **OR true** ⇒ **K true, OR false** ⇒ **I**
18. **OR x true** ⇒ **true, OR x false** ⇒ **x**
19. **IF true** ⇒ **K, IF false** ⇒ **K I**
20. **IF (NOT x)** ⇒ **S(K(S(IF x)))K**
21. **(IF p c a) x** ⇒ **IF p (c x) (a x)**
22. **f (IF p c a)** ⇒ **IF p (f c) (f a)**, provided **f** is strict in its first argument
23. **HD (P x y)** ⇒ **x**, **TL (P x y)** ⇒ **y**

Rules 1-5 are optimal, in that they were obtained from an exhaustive case-by-case analysis of abstraction using **S**, **K**, and **I**. Rules 6-8 are simply the reduction rules for **S**, **K**, and **I**, and rule 9 eliminates the **Y** operator when no recursion is present. Rule 10 accomplishes arithmetic

---

[3] "Infinite lists" are one example, but a more straightforward one is the ALFL program **{f x y == y=0 → 0, f (f x y) (y-1); result f 0 1}**.

[4] Other terms have included *procrastinating, lenient, demand-driven, and call-by-need.* The *call-by-name* strategy used in ALGOL is also closely related.

constant-folding, and rule 11 folds type-testing when possible. Rule 12 or 13 is applicable only if it can be determined either that **x** and **y** are equal, or that they are not equal, respectively. Rules 14-20 are boolean simplifications (rule 20 may not seem to "simplify" much, but **NOT** has been eliminated, and the **S**'s and **K**'s will dissappear as a result of other simplifications). The purpose of rules 21 and 22 is to cause as much function application as possible to allow further simplifications. Note that to retain normal-order reduction semantics, rule 22 cannot be applied unless **f** is a strict function; more will be said about this later. Finally, rule 23 forces selection of elements from lists. As a further simplification, the extra combinators **B**, **C**, **S'**, **B'**, and **C'** [20] are used in our implementation to reduce the "intermediate code" size.

As the combinator expressions are created, common-subexpression elimination is also performed. Using standard techniques, a hash-value is computed for each expression and saved for comparison to subsequent ones. If a match is found, the common subexpressions are collapsed into one. Since there are no side-effects, this allows *global* common subexpression elimination.

3.3. Implications

3.3.1. Advantages

Phase I of the compiler essentially performs a *partial evaluation* of the entire source program, collapsing it into a single expression.[5] The abstraction algorithm automatically preserves the block-structured scoping rules, and mutually recursive references are eliminated as easily as any other. Constant-folding optimizations occur automatically, and the same mechanisms that fold arithmetic expressions such as **+ x y** also fold binding expressions such as **S f g x**. Higher-order functions such as **twofacs** defined earlier collapse into precisely the same combinator expression that would have resulted if the function **twice** were *not* used. Indeed, the abstraction algorithm actually "collapses" all procedure boundaries except recursive ones. Thus there is no penalty for employing "good design rules" that suggest, for example, naming subexpressions and using procedural abstractions (especially unshared ones) to modularize and clarify code.

Another implication of our algorithm is that there is no penalty for "packaging" data into lists that are passed between procedures, such as the "destructuring" capability found in some LISPs (which generally does *not* come without penalty). This advantage is accrued not only with non-recursive procedures, but also with recursive ones. For example, for this contrived version of **member**:

```
{ member [[x],[[L]]] ==
                L=[] → false,
                x=hd L → true,
                member [[x],[[tl L]]];
  result member [[2],[[[1,2,3]]]] }
```

our compiler generates *precisely* the same code as the version defined earlier! All conses are eliminated completely except for those in **[1,2,3]**.

3.3.2. Disadvantages

The chief disadvantage of our simplification strategy is that all objects defined by the user via an equation get embedded, or *integrated*, into the expressions that reference them. In the case of common sub-expressions, this is no problem, since the object is still shared -- all references to the object point to the same thing. The same is true for a recursive function, since it remains intact because no simplifications are made across the **Y** combinator. However, this is not true with *non-recursive* functions, in that the body of the function is essentially duplicated wherever it is applied. This does not result in a detectable common-subexpression, but rather creates a common *interior* portion of the graph that represents the function body. Of course our reason for doing all this in the first place is to induce as much constant-folding as possible, and to eliminate the overhead of a procedure call.

The two chief complaints about large amounts of code are that it places a strain on the address space and that it might induce excessive paging. Note, however, that when executing a program by normal-order graph reduction (for example as described in [4] or [19]), the same code explosion occurs, except that it only happens if the function is actually applied, and the garbage collector *may* collect the space at some point after application. In essence we are trading static space for dynamic space, and trading utilization of the address space for load on the garbage collector. The paging performance would be roughly the same in both cases.

Even though the code explosion only happens with non-recursive functions, we view it as a problem, and would like to avoid it. One obvious solution is to be judicious as to when we integrate a function. A much better solution, and the one we are pursuing, is to integrate all procedures as we have described, and then abstract out *common interior subtrees*. Generally not all common interior subtrees would be abstracted. For example, implementation details may indicate that the extra code caused by a function call offsets the gain made by abstracting a particular function. Also, user supplied performance requirements may dictate that the time overhead of a particular function call cannot be tolerated.

[5] The only difference between our strategy and applicative-order reduction is that we do not perform **Y** reductions, since that would amount to program execution.

We are currently investigating ways to detect and abstract common interior subtrees. Efficient algorithms have been developed and are being integrated into the compiler. The results of this work to date, however, are too preliminary to report here.

Another disadvantage of our approach is that the compilation strategy is not guaranteed to terminate! This seems surprising, since we don't perform any **Y** reductions, which correspond to the recursions that are normally the cause of non-terminating programs. The problem can arise, however, if one performs certain forms of *self-application*. For example, the simple ALFL program:

```
{ f x == x x;
  result f f }
```

corresponds to the combinator expression **S I I (S I I)** (and the lambda expression **(λx.xx)(λx.xx)**), and does not terminate, since our applicative-order partial evaluator applies **f** to **f**, reducing to an expression trying to do the same thing. We do not view this is a serious problem, for two reasons: First, self-application is a rare programming event, and not all self-applications will fail to terminate (we have never encountered a non-terminating compilation). Second, a program that does not terminate during compilation will *probably* not terminate when executed.

4. Phase II: Combinator Translation and Code Generation

Once the optimized combinator expression is constructed as described in the last section, how is it executed? One could use a normal-order reducer such as described in [19], but this approach has certain annoying inefficiencies. The primary problem is the way in which actual parameters are bound in function calls. Consider a variable **v** that appears at depth n in the body of a function (by depth we do not mean *lexical* depth -- rather, we mean the depth in the expression tree). When the corresponding combinator expression is applied to a value for **v**, at least n reductions are required before the value arrives at its proper place (and more if the variable appears in more than one place).[7] In a standard procedure call on a conventional machine, the value of **v** would be loaded into a register or stack location from which it is directly accessible (perhaps through a small number of indirections). Clearly the latter is much more efficient.[8]

Furthermore, most implementors of normal-order combinator reduction (or, for that matter, lambda reduction) make two implicit assumptions that make the job easier: First, it is assumed that *all* expressions must be computed lazily, so that the normal-ordering of the redexes can be followed "blindly". The overhead paid for this high, although can be subtly hidden in the reduction strategy. Second, all expressions are assumed to be *shared*, which means that all intermediate results are "stored away" by overwriting the appropriate vertices in the graph, regardless of whether they will be needed or not.

The converses of the above arguments, of course, point out the *advantages* of graph reduction. The variable binding mechanisms obviate the need for an environment structure, and in those cases where expressions *must* be treated lazily or shared, the graph reduction strategy works quite well. The approach we have taken is to combine the best features of both strategies, using a conventional machine as our target architecture.[9] Specifically, our first goal is to utilize the efficient variable referencing mechanisms of standard procedure-calling strategies. Furthermore, although we believe in making normal-order evaluation the default *conceptually* in a functional language, we also recognize that most applications do not require it.[10] Since applicative-order reduction can be implemented more efficiently on conventional machines, a great savings would result by detecting those cases where lazy evaluation is not needed. Finally, empirical studies show that only a small percentage of all objects are actually shared in most programs [3], so another significant savings would result by retaining the values of only those objects that are shared.

Phase II of our compiler accomplishes the above optimizations by making four passes over the program, starting with the combinator expression from Phase I and ending in machine code:

1. Phase IIa: An environment structure is embedded into the optimized combinator expression by converting it into a nested graph of lambda expressions that we call **macro-combinators**. (See Section 4.1.)

[7]Despite this inefficiency it is conceivable that if the reduction mechanisms were made an integral part of a machine's architecture, combinator reduction could be a feasible alternative to conventional machines [4].

[8]Although the study in [10] seems to indicate that combinator reduction does about as well (if not better) than lambda reduction, care must be taken in drawing any conclusions from this, for implementation-dependent reasons. For example, the lambda reducer used in that study employed an association-list lookup strategy to resolve variable references, which is clearly non-optimal, since one can determine at compile-time the precise lexical environment and location in that environment of each variable.

[9]Another way at looking at our approach is that we are trying to combine the best features of a pure "fixed-program machine" with those of a pure "substitution", or "reduction machine".

[10]The success of conventional languages is sufficient testimony to this fact!

2. Phase IIb: Two simple analyses are then done, one to locate all *shares* (those expressions whose value might be shared, which is more than just the set of common subexpressions), and the other to determine in which variables and shares each macro-combinator is *strict*. (See Section 4.2.)
3. Phase IIc: In order to observe a stack discipline for function calls whenever possible, an analysis is made to determine when the extent of a variable or share might exceed the lifetime of its activation record. (See Section 4.3.)
4. Phase IId: Code generation: "Lazy" shares and arguments are implemented as *self-modifying thunks* to preserve normal-order semantics, heap-allocated environments are created when necessary, and a few other standard optimizations are done to improve code efficiency. (See Section 4.4.)

Each of the above passes is described in detail in the following sections.

4.1. Uncurrying the Combinator Graph

In order to compile the optimized combinator graph into code for a conventional machine, we first need to *uncurry*[11] all functions in order to gain the efficiency of passing more than one argument in a function call. In anticipation of this, as the combinator expression is constructed in Phase I, we maintain a **redex-arity** for each subexpression, defined to be the number of arguments needed by that expression to form a redex at the outermost level. For example, the redex-arity of **S** is 3, of **S (K f)** is 2, and of **+ x** is 1. Note that the redex-arity of an expression may actually *increase* after it is applied and reduced, as with **K S**.

Now consider a combinator expression **expl** having a redex-arity of at least 1. If **expl** is applied to a series of "dummy variables" until the redex-arity of the resultant expression is zero, we will have embedded variable names into the expression. If $\mathbf{x_1}$ through $\mathbf{x_n}$ are the dummy variables supplied, and **expl'** is the resulting expression, then the combinator expression **expl** corresponds to the lambda expression $\lambda \mathbf{x_1 ... x_n}$. **expl'**.

Next consider the combinator expression **Y exp2**, whose redex-arity is zero. The redex-arity of **exp2** must be at least one. To obtain a lambda expression from this we first apply **exp2** to a "dummy function" **f**, and then supply dummy arguments as above. From this we eventually obtain a *recursive equation* defining the function **f**; viz.:

$$\mathbf{f} = \lambda \mathbf{x_1 ... x_n} \,.\, \mathbf{exp2'}$$

We refer to functions so generated as **macro-combinators**.

Phase IIa of the compiler essentially converts all expressions in the combinator graph to lambda expressions, uncovering macro-combinators in preparation for code generation. The above algorithm is used to do this, except that the sharing of higher-order functions must be maintained. For example, the combinator graph shown in Figure 4-1 contains a subexpression **Y fun expl** that is shared by two nodes (in this figure each node is an *apply* node, the left branch being the function, the right the argument). If $\mathbf{f} = \lambda \mathbf{x_1 ... x_n}.\mathbf{exp}$ is the macro-combinator corresponding **Y fun**, then it is transformed into $\mathbf{f} = \lambda \mathbf{x_1}.(\lambda \mathbf{x_2 ... x_n}.\mathbf{exp})$ as shown.

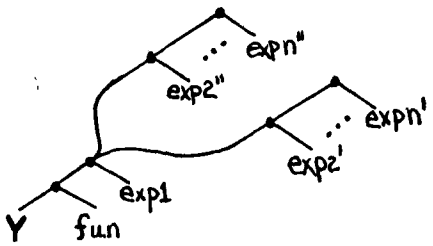


Figure 4-1: Shared High-Order Functions

We refer to the result of this pass of the compiler as the **macro-combinator graph**, or **MCG**. The MCG is built with slots to be filled in with optimization information gleaned from the next two passes, in preparation for a final pass to perform code-generation. Also in this pass a free-variable list is maintained for every subexpression as the dummy arguments are supplied -- this is needed to compute *shares* as described in the next section.

4.2. Computing Shares and Converting Normal-order to Applicative-order Evaluation

Phase IIb locates *shares* in a top-down pass over the MCG, and then bottom-up in the same pass computes information to convert normal-order to applicative-order evaluation wherever possible.

4.2.1. Shares

Consider the function (macro-combinator) **f = λn. if n=fac(x) then 1, else f(n-1)**, where **fac** is the factorial function, and **x** is free. This lambda expression represents a *closure* for **f**, which can only be fully created at run-time since **x** may take on different values dependent on the particular invocation of the context in which it is defined. A conventional implementation of **f** might compute **fac(x)** every time the closure is called. On the other hand, a fully lazy graph reduction strategy would only compute it the *first* time the closure is called, sharing the result with subsequent calls. This effect of "code

[11] Our use of this word is quite a bit different from that in [19], but our use is more faithful to "undo the effects of currying".

motion from a loop" (with the recursion behaving as the loop) comes free in graph reduction, but requires a little more work in a more conventional implementation. Specifically we must identify, for each macro-combinator **f** having set of bound variables **args**, the largest subexpressions in the body of **f** whose free-variable list is disjoint from **args** ∪ {**f**}. This is easily computed by a top-down scan of the MCG -- when a macro-combinator is encountered its arguments are remembered and compared to the free-variable lists of subexpressions found in the body. The top-down traversal guarantees that the first subexpressions thus found are maximal.

These expressions are similar to the *maximally-free expressions* discussed in [7]. In our implementation they are treated in the same way as common subexpressions, and we refer to them collectively as **shares**. Storage is allocated for a share's value in the local environment of the *outer-most* surrounding macro-combinator from which all of the share's free variables can be accessed. The notation shares⟦**f**⟧ denotes the set of all shares allocated space in **f**'s environment.

Conventional code-motion strategies would move all shares outside of (i.e., "in front of") any loop (recursion) in which they lie. This is satisfactory under an applicative-order evaluation, but under normal-order such code motion may result in evaluating something that was not intended to be, and the semantics may be violated if that evaluation fails to terminate. Thus code motions are performed only if that function is **strict** in the computation of that share;[12] that is, if it can be determined that the share's value will be computed if that function is called and terminates. A share that is not strictly computed we call a *lazy share*, for which it becomes necessary to dynamically create an object that can be evaluated on demand, and whose result can be shared. We accomplish this with "self-modifying thunks", to be described in Section 4.4.

The one other by-product of this pass of the compiler is the computation of a **free-object list** for every subexpression -- free-obs⟦**exp**⟧ denotes the set of all free variables and free shares contained in the expression **exp**.

4.2.2. Function Strictness

Whether or not a macro-combinator is strict in one of its shares, of course, is not all we wish to determine. Using applicative-order evaluation, a conventional call to a function **f** would involve computing the required number of actual parameters, pushing them onto a stack, and jumping to the code for **f**. However, lazy evaluation requires that one push some sort of closure for an argument, so that it is not computed unless the called function actually needs its value. Since, as disussed earlier, most functions do indeed evaluate their arguments, there is room for optimization here. To avoid the overhead of a closure we need to determine in which of its arguments each macro-combinator is strict.

In addition to locating shares, Phase IIb determines the strictness information that we need.[13] This is done through a simple bottom-up analysis by first computing the **used-objects list** of every subexpression, defined as follows:

1. used-obs⟦**c**⟧ = Ø, where **c** is a constant.
2. used-obs⟦**v**⟧ = {**v**}, where **v** is a variable.
3. used-obs⟦**x op y**⟧ = used-obs⟦**x**⟧ ∪ used-obs⟦**y**⟧, where **op** is any strict binary operator.
4. used-obs⟦**op x**⟧ = used-obs⟦**x**⟧, where **op** is any strict unary operator.
5. used-obs⟦**if x y z**⟧ = used-obs⟦**x**⟧ ∪ (used-obs⟦**y**⟧ ∩ used-obs⟦**z**⟧)
6. used-obs⟦**f** $\mathbf{arg_1 ... arg_n}$⟧ = if **f** is a macro-combinator $\lambda \mathbf{x_1 ... x_n}.\mathbf{exp}$, then $\{\mathbf{x_1},...,\mathbf{x_n},\mathbf{f}\}$; if **f** is a primitive function with args $\mathbf{x_1},...,\mathbf{x_n}$, then {**f**} ∪ used-obs⟦$\mathbf{arg_i}$⟧ for each $\mathbf{x_i}$ in which **f** is strict; otherwise Ø.
7. used-obs⟦$\mathbf{f}=\lambda \mathbf{x_1 ... x_n}.\mathbf{body}$⟧ = used-obs⟦**body**⟧ - $\{\mathbf{x_1},...,\mathbf{x_n},\mathbf{f}\}$ - shares⟦**f**⟧

Intuitively, used-obs⟦**n**⟧ contains all variables and shares whose value will definitely be computed if **n**'s value is computed, with one important exception: In order to determine the strictness of recursive functions in a single bottom-up sweep over the MCG, the used-object list of a function call to a macro-combinator is simply the set of bound variables for that function (see rule 6). The idea is that if such a variable **v** propagates to the used-object list of the body of the function **f** having **v** as an argument, then **f** must be strict in **v**. More specifically, the **strict-object list** of a macro-combinator **f** is defined by:

strict-obs⟦$\mathbf{f}=\lambda \mathbf{x_1 ... x_n}.\mathbf{body}$⟧ = used-obs⟦**body**⟧ ∩ ($\{x_1,...,x_n\}$ ∪ shares⟦**f**⟧)

This strategy allows us, for example, to determine that this tail-recursive version of factorial is strict in *both* of its arguments:

fac n a == n=0 → a, fac (n-1) (n*a)

[12] Mathematically speaking, a function $f(x_1,x_2,...,x_n)$ is *strict* in x_i if $f(x_1,...,x_{i-1},\perp,x_{i+1},...,x_n) = \perp$ for all values of x_j, $j \neq i$. [17] We extend this definition somewhat by saying that a function may be strict in one of its *subexpressions*, namely a share.

[13] Earlier designs of our compiler performed share detection and strictness analysis on the combinator graph itself. Since our target machine requires a lambda representation, it turned out to be more convenient to do the analysis on the MCG.

4.3. Escape Analysis

The purpose of Phase IIc is to determine when the extent of a variable or share might exceed the lifetime of the activation record in which it resides. Many of the issues faced here are not too different from those faced by implementors of any language that treats functions as "first-class citizens" and in addition uses a stack implementation of function calls for efficiency.[14]

We say that a variable or share **escapes** the context (i.e., macro-combinator) in which it is defined if it is a free object in either a function appearing in a non-functional position (i.e., it appears in a function that is not in the function position of function call), or it is free in the expression for a lazy argument in a function call (which includes the arguments to a CONS). Phase IIc simply walks through the MCG, marking all variables and shares that escape, which for a macro-combinator **f**, is denoted escapes⟦**f**⟧.

4.4. Code Generation: Self-modifying Thunks

At this point in the compilation we are ready for code generation. To summarize, from the previous three passes the following information is known about every macro-combinator **f**:

1. args⟦**f**⟧ -- the arguments of **f**.
2. shares⟦**f**⟧ -- the set of shares allocated in **f**'s local environment.
3. free-obs⟦**f**⟧ -- the set of free variables and free shares in **f** (which is disjoint from args⟦**f**⟧ ∪ shares⟦**f**⟧).
4. strict-obs⟦**f**⟧ -- the set of arguments and local shares that **f** strictly computes (a subset of args⟦**f**⟧ ∪ shares⟦**f**⟧).
5. escape-obs⟦**f**⟧ -- the set of arguments and local shares that escape from **f** (a subset of args⟦**f**⟧ ∪ shares⟦**f**⟧).

Also, for each subexpression **exp** we know free-obs⟦**exp**⟧.

Phase IId generates code for each macro-combinator, which can be thought of as generating code for a conventional function with a local environment of argument values and shares. The default environment mechanism is a stack-allocated activation record, with access to objects in surrounding contexts accomplished through a register-allocated *display* [1]. The environment for functional values is kept on the stack whenever possible (dictated by the escape analysis) -- when not possible a mini-environment for that object is consed in the heap.

[14] In particular, our problem is similar to that faced by implementors of lexically-scoped LISPS such as SCHEME [16] or T [12].

Here is the code skeleton for a macro-combinator **f**:

```
<entry code>
<for each share s ∈ shares⟦f⟧ - strict-obs⟦f⟧,
  create self-modifying-thunk for s>
        ;;; for lazy shares
<for each share s ∈ strict-obs⟦f⟧,
  compute its value>
        ;;; for strict shares
<code for body>
        ;;; for the body of f
<exit code>
```

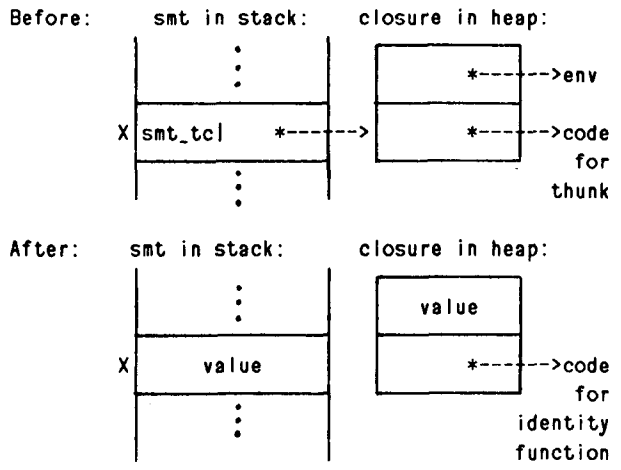
Shares are accessed from lower contexts by indexing through the appropriate display register. If it is a strictly computed share, its value may be directly fetched. If it is a lazy share, the self-modifying-thunk is used to compute its value, in a way to be described shortly.

Now consider a function call **f** $\mathbf{exp_1}$ ... $\mathbf{exp_n}$ where **f** is a macro-combinator $\lambda \mathbf{x_1 ... x_n}.\mathbf{body}$. If $\mathbf{x_i} \in$ strict-obs⟦**f**⟧ then the value of $\mathbf{exp_i}$ is computed prior to calling **f**. On the other hand, if $\mathbf{x_i} \notin$ strict-obs⟦**f**⟧, then a self-modifying-thunk is created for the computation of $\mathbf{exp_i}$, just as was done for a lazy share. (Of course, as a simple optimization, if the value of $\mathbf{exp_i}$ is already known (such as for a variable or constant), it can be pushed onto the stack directly.)

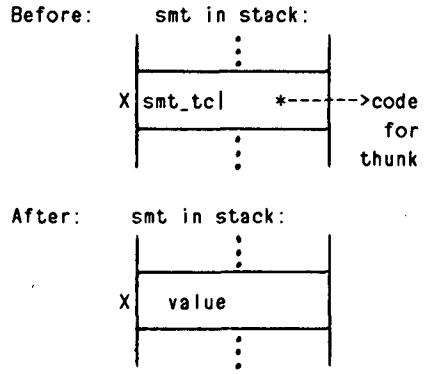
How is a self-modifying-thunk for an expression **x** created, and how is it evaluated? First we must use escape analysis to determine whether its environment needs to be consed in the heap. If **x** is a share, this arises if **x** ∈ escape-obs⟦**f**⟧, where **f** is the macro-combinator in which **x** is defined. If **x** is the ith argument in a call to a macro-combinator **g**, this arises if the corresponding formal parameter $\mathbf{x_i} \in$ escape-obs⟦**g**⟧. In either case there can be no guarantee that the objects on which **x** depends will be on the stack when it is eventually evaluated. A mini-environment **env** containing free-obs⟦**x**⟧ is consed in the heap, along with a closure containing a pointer to **env** and a pointer to the code for **x**. This allows us to *share environments* consed in the heap. The strategy is depicted in Figure 4-2a, both before and after evaluation, where "X" is the stack location (either local or in a surrounding context) of **x**, and "smt_tc" is the type-code of a self-modifying-thunk. Note that once the value is computed, the stack location for **x** and the **env** slot in the closure are overwritten with the result. In addition, the code-pointer of the closure is overwritten with a pointer to an identity function that simply retrieves **x**'s value from the **env** slot if the thunk's computation is attempted by some other reference.[15]

[15] This could be viewed as "self-modifying code", normally frowned upon, but in a purely functional setting there is no harm -- indeed the object never changes in *functionality*, and in a sense is not modified at all.

If **x** does *not* escape, then (1) all of the objects needed to compute its value will be somewhere on the stack at the time **x** is evaluated, and (2) all references to **x** will be to the same stack location. Thus there is no reason to create a mini-environment, and no reason to create a closure. Instead, the self-modifying-thunk simply contains a pointer to the code for **x**, where accesses to **x**'s



(a) Escaping Thunk



(b) Non-escaping Thunk

Figure 4-2: Self-Modifying Thunks

free objects are made through the display as in a normal function call. The value thus computed is simply written into the stack location for **x**. Figure 4-2b depicts this situation.

5. Results

Everything described in Sections 3 and 4 has been implemented, although space precludes providing further details. The compiler was written in T [12], and generates MACRO code for the PDP-10. Our primary goal was to demonstrate the feasibility of the ideas described in this paper, and we thus paid little attention to more conventional optimizations. Nevertheless, the results are very encouraging. As can be expected, executing our compiled code is a vast improvement over executing combinatory code using a normal-order reducer. However, it is difficult to make comparisons with, for example, LISP code, for two reasons. First, as mentioned above, we have ignored more traditional optimizations -- a just comparison is difficult to make until we do better at such things as register allocation and tail-recursion optimizations. Second, such comparisons are very problem-dependent. For example, in cases where an ALFL program packages data into lists to pass between procedures, the equivalent LISP program naturally incurs a greater overhead. Our compiler also does better in handling "lazy lists", which must be implemented as explicit closures in LISP. On the other hand, lists that do *not* require lazy evaluation are handled more efficiently in LISP. Similarly, our compiler does worse when it cannot determine when an argument to a function is strictly computed, such as **a** in:

```
g a == { result f 4;
         f x == x=0 → a, f (x-a) }
```

As is true of most compilers, ours has much room for improvement. Many of the optimization analyses are too conservative, and as mentioned, we paid little attention to certain more conventional optimizations. We see the following as particular areas for improvement:

1. The common interior subtree analysis described in Section 3.3.2 needs to be implemented.
2. The strictness analysis can be improved. For example, the strategy in Section 4.2 can be extended by adding an extra pass over the MCG in which the free objects in the arguments in a function-call are included in the used-obs analysis (this is sufficient to determine that **g** defined above is strict in **a**). Or, the technique described in [11] could be used for a more thorough analysis.
3. The escape analysis can also be improved, as well as the mechanism for consing environments onto the heap. Better safety analyses [9], environment structures [2], and calling strategies [14] could improve the handling of closures.
4. Commutative, associative, and distributive laws could be used to improve code generation as well as to reorganize shares (to make them larger, for example).
5. Certain other conventional optimizations could have a great impact, in particular converting tail recursions into loops, and doing a better job of register assignment and allocation.

6. Relationship to Other Work

We received most of our inspiration from Turner's work with combinators and normal-order graph-reduction [19, 20]. Our shares (except for common subexpressions) are similar to the maximally-free-expressions identified by Hughes cite[hugh82], although we make no attempt to make them larger. Also, if the shares are abstracted out from our macro-combinators (that is, made arguments to the function), then one essentially generates Hughes' *super-combinators*, although they are arrived at from quite different approaches. There is no need in our implementation to abstract the shares, since they are accessible via the environment structure; i.e., by indexing through a display register. It should be noted that the "fixed-program" code described in [10] is simply an iterative version of standard graph reduction, and does not have the fixed-program flavor that our compiler possesses. Some of our compilation techniques were influenced by Steele's work with lexically-scoped LISPs [15].

7. Acknowledgements

Thanks to Bob Paige for helpful comments, to Cathy Van Dyke for her support, to Cristina for her inspiration, to Leslie for the details, and to the Macaroni Group at Yale (especially Jonathan Rees, Norm Adams, and Jim Philbin) for many a mouthful.

References

[1] Aho, A.V. and Ullman, J.D.
Principles of Compiler Design.
Addison-Wesley, 1977.

[2] Bobrow, D.G. and Wegbreit, B.
A model and stack implementation of multiple environments.
CACM 16(10):591-603, October, 1973.

[3] Clark, D.W.
An empirical study of list structure in LISP.
CACM 20(2):78-87, February, 1977.

[4] Clarke, T., Gladstone, P., MacLean, Norman, A.
SKIM - the S, K, I reduction machine.
In Davis, R.E., and Allen, J.R. (editors), *The 1980 LISP Conference*, pages 128-135. Stanford University, August, 1980.

[5] Curry, H.K., and Feys, R.
Combinatoroy Logic.
Noth-Holland Pub. Co., Amsterdam, 1958.

[6] Henderson, P.
Functional Programming: Application and Implementation.
Prentice-Hall, Englewood Cliffs, NJ, 1980.

[7] Hughes, R.J.M.
Super-combinators: A new implementation method for applicative languages.
In Park et al. (editors), *Sym. on Lisp and Functional Prog.*, pages 1-10. ACM, August, 1982.

[8] Keller, R.M.
FEL programmer's guide.
AMPS TR 7, University of Utah, March, 1982.

[9] McDermott, D.
An efficient environment allocation scheme in an interpreter for a lexically-scoped LISP.
In Davis, R.E., and Allen, J.R. (editors), *The 1980 LISP Conference*, pages 154-162. Stanford University, August, 1980.

[10] Muchnick, S.S. and Jones, N.D.
A fixed-program machine for combinator expression evaluation.
In Park et al. (editors), *Sym. on Lisp and Functional Prog.*, pages 11-20. ACM, August, 1982.

[11] Mycroft, A.
Call-by-need = call-by-value + conditional.
Draft, Dept. of Comp. Sc., Univ. of Edinburgh, 1981.

[12] Rees, J.A., and Adams, N.I.
T: a dialect of LISP or, Lambda: the ultimate software tool.
In Park et al. (editors), *Sym. on Lisp and Functional Prog.*, pages 114-122. ACM, August,1982.

[13] Schonfinkel, M.
Uber die bausteine der mathematischen logik.
Mathematische Annalen 92:305, 1924.

[14] Steele, G.L.
Debunking the expensive procedure call myth.
In *Proc. ACM National Conference*, pages 153-162. ACM, 1977.

[15] Steele, G.L.
RABBIT: A Compiler for SCHEME.
AI 474, MIT, May, 1978.

[16] Steele, G.L. and Sussman, G.J.
The Revised Report on Scheme.
AI 452, MIT, January, 1978.

[17] Stoy, J.E.
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.
The MIT Press, Cambridge, Mass., 1977.

[18] Turner, D.A.
SASL language manual.
Technical Report, University of St. Andrews, 1976.

[19] Turner, D.A.
A new implementation technique for applicative languages.
Software-Practice and Experience 9:31-49, 1979.

[20] Turner, D.A.
Another algorithm for bracket abstraction.
The Journal of Symbolic Logic 44(2):267-270, June, 1979.