Re-opening Closures

Andrew W. Appel

Department of Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

There are two different commonly-used evaluation methods for functional languages: normal-order graph reduction, and call-by-value execution of closure code. The former is usually more expensive per operation, but has the capability of partially evaluating functions before they are applied. The latter method usually leads to faster execution — and is thus used in most compilers — but can't ''optimize'' functions before they are called.

The different advantages of the two methods are particularly visible in the evaluation of higher-order functions. After a higher-order function is applied to one argument, the graph-reducer can begin evaluation, while the closure-code evaluator must wait until all arguments are present. On the other hand, because the closure-code evaluator executes the native code of the computer, it usually outperforms the graph-reducer.

The two evaluation algorithms can be combined to take advantage of the best behaviors of both. Fragments from programs that are already executing can be extracted, reduced, and re-compiled. This is done with an operator, **reduce,** that is semantically transparent: — **reduce**(*f*) does not change the behaviour of the program fragment *f*, but can make *f* much more efficient. This applies not just to functions *f* in the original program, but to functions constructed at runtime.

1. Higher-order functions

The notion of a higher-order function (a function that returns another function as a result) is present in many functional languages, but it is often found in conventional languages as well. For example, the C programming language has no built-in notion of higher-order functions, but there are many software-development tools (like the parser generator YACC[1]) that produce C programs as their output. It is inconvenient, inefficient, and ugly to implement higher-order functions by producing source code; so the very fact that it has been done repeatedly shows how important higher-order functions are.

Functional languages based on lambda-calculus have an inherent notion of higher-order function, so that they may be programmed in the language directly, without resorting to the output of source code to be fed back into the compiler. This is a great advantage, since now functions like parser generators can be implemented as a higher-order function completely inside the language, perhaps with the type:

PG: Grammar → (String → ParseTree)

The parser-generator *PG* can then be applied to a grammar, yielding a parser *P* :

P = PG(MyGrammar)

This step may take a nontrivial amount of time, as it involves the construction of parsing tables. Once it is done, however, the parser *P* may be applied to many different input strings, without executing *PG* for each one.

At this point a user of C might object, ‘‘It is true that my higher-order function must produce source text and compile it, but the result will be a more efficient function’’ (in this case, the parser P). And he will be right. The functional programming language will have produced a *closure*[2] — a tuple consisting of a generic parser, with a parsing table and other information about the grammar bound to it. The C-language parser-generator can specialize the C program that it produces to the particular grammar being compiled, while the functional programming language won't be able to do this. (This will be explained in section 2.)

This paper shows how a functional programming language can be implemented so that higher-order functions do not pay a penalty for their generality. Thus, there need never be any reason to for a program to produce source text as its output.

2. Evaluating functional programs

There are two commonly used methods to evaluate lambda-calculus programs: graph reduction, and closure-code execution. In either case, one can consider the starting point to be a representation of the program as a graph containing nodes for variables, for function application, and for lambda-abstraction.

Graph reduction[34] involves the repeated stepwise simplification of the graph by substituting function-arguments for bound variables. When no more substitutions are possible, the graph has reached ‘‘normal form;’’ the answer is considered to have been reached.

Instead of reducing a graph, it may be translated into machine code for a conventional (von Neumann) machine [25]. In this method, the argument substitutions are not represented by re-writing the graph; instead, intermediate values are represented by tuples (called *closures*) of program-text and environment. The program-text is the translation of some fragment of the original graph — possibly a fragment that has references to free variables. The environment is a mapping from the free variables to their values in this context.

For example, the lambda-expression

$$(\lambda f.\ \lambda x.\ f(x)+f(0)\ )\quad(\lambda e.\ 3)$$

can be evaluated by graph reduction, as shown in Figure 1; or by applicative-order closure-code execution, yielding the machine configuration shown in Figure 2. In the closure-code, the translation of a graph fragment into machine code is indicated by surrounding it with a box.

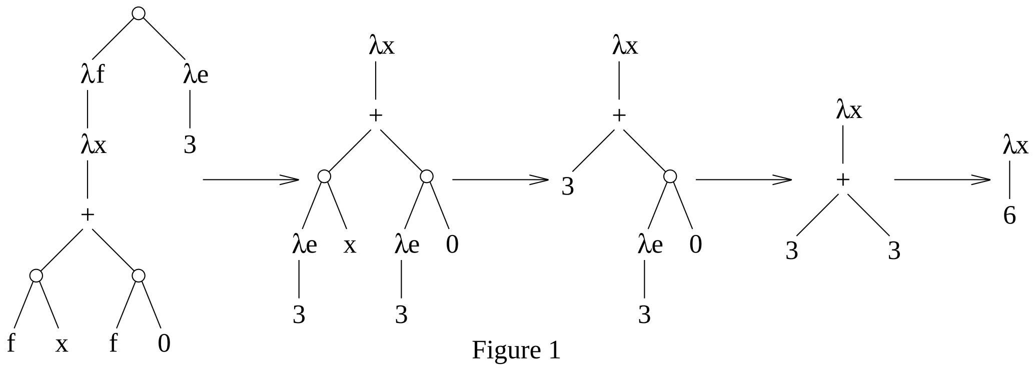

Figure 1

In Figure 2, the root cell is a closure, whose *text pointer* points to a lambda expression with a free variable f , and whose second element is the binding of the variable *f.* In general, there may be more than one free variable, so the closure would contain more than one binding cell.

Semantically, Figure 2 is equivalent to any of the graphs in Figure 1; but the applicative-order evaluator is unable to continue evaluating this configuration until a value is provided for *x*. One solution to this problem is to use graph reduction to as the evaluation algorithm; but this does not approach the efficiency of applicative-order closure codes for most programs. This paper shows how the two methods may be effectively combined, producing an evaluator that keeps the best features of both.

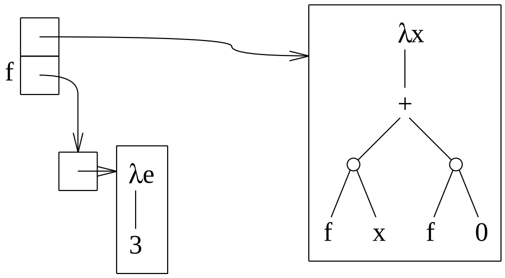

Figure 2

3. Compilers can do graph-reduction

A typical compiler for a functional language uses an intermediate representation similar to the lambda-calculus. That is, the compiler consists of a *front end*, which translates the source program into lambda-calculus; and a *code generator*, which translates the lambda-calculus into machine code. In such a compiler, it is possible to insert a *reducer*, which performs optimizations on the lambda-calculus representation[56]. The phases of the compiler would interact as diagrammed in Figure 3.

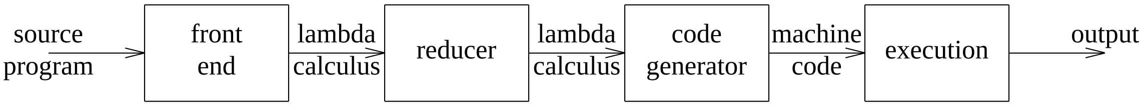

Figure 3

The reducer can be very much like a general-purpose graph-reduction evaluator for lambda-calculus. However, it is desirable (and we shall require) that the compiler (including the reducer) must terminate on any input. A general normal-order graph-reduction evaluator can evaluate any function, including those that do not yield answers.

By restricting the power of the reducer, we can guarantee that it will always terminate. For example, we could require that any substitution reduce the number of nodes in the graph. This is an extremely conservative restriction. On the other hand, we could simply require that any fixed point combinator (i.e., any ‘‘loop’’) be expanded only a constant number of times; this is sufficient to guarantee termination but may be unmanageable in practice.

In any case, there are a number of ‘‘safe’’ substitutions that will improve the graph without danger of having the reducer loop indefinitely. The implementation of such reducers is a non-trivial problem that will be addressed in section 7. Some of the examples in later sections rely on more powerful reducers than other examples, and these will be noted.

By putting a graph-reduction evaluator in the compiler, and generating closure code to execute on the ‘‘bare machine,’’ we have partially integrated these two forms of functional-language evaluation. However, this integration is not sufficient to handle the optimization of higher-order functions. By the time a higher-order function is actually applied to a specific argument value, we are already executing the program — we have left the realm of the compiler and reducer.

4. Integrating reduction and execution

No matter how much optimization is done in the compilation phase, certain values can use more optimization at runtime. This is because some combinations of program fragments are created only as a response to input data, and it is not possible to optimize these combinations before they occur. For example, suppose the function $(\lambda f.\lambda x.\ f(x)+f(0)\ )$ is applied to the argument $(\lambda e.\ 3)$ as a result of some runtime calculation. The runtime structure built for this combination is shown in Figure 2. Although the function is semantically equivalent to $(\lambda x.\ 6)$, its representation is much more complicated and inefficient. If this combination had occurred at compile time, then it could have been optimized; but at runtime it is ‘‘too late’’ to improve the representation.

Suppose we could ‘‘re-open’’ this closure feed it back into the reducer. The function could then be optimized into $(\lambda x.\ 6)$, and all the applications of this function would then run much faster. Of course, there will be significant overhead in arranging for this reduction, so this would be worth doing only if we expected the function to be applied to many different arguments.

How is it possible to take a piece of the executing program and reduce it? After all, the representation is no longer lambda-calculus, but machine language. The boxed trees in Figure 2 represent *native code* for the corresponding lambda-expressions, and this is *not* a representation that the reducer can understand.

One might consider trying to extract the lambda-tree from the block of machine code. This not impossible, but it is likely to be difficult and unreliable. Instead, we will use a trick borrowed from symbolic debuggers. A symbolic debugger allows the manipulation of machine programs, but displays them to the user in their source representation. The debugger does not accomplish this by un-compiling the machine code; instead, it uses markers in the machine code that indicate from which source statements the machine-instructions were derived.

To each fragment of machine code we will prepend a pointer to the original lambda-expression from which it was derived. Then it will not be necessary to understand the structure of machine-code fragments — following the pointer will be sufficient. Figure 4 illustrates this.

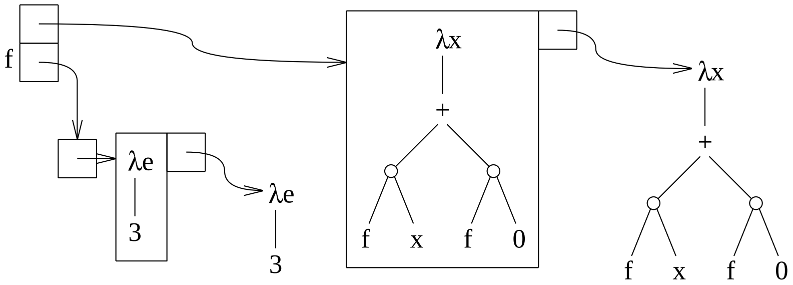

Figure 4

Now, suppose we want to ‘‘un-compile’’ the program fragment shown in Figure 4. We take the tree for the fragment $(\lambda f.\lambda x.\ f(x)+f(0)\ )$, and substitute the tree $(\lambda e.\ 3)$ for each occurrence of *f*. There is certainly enough information in the data structure to do this. The intermediate form has thus been recovered from the runtime program fragment.

Now it is a simple matter to feed this lambda-graph into the reducer. The reducer can optimize the graph into $(\lambda x.\ 6)$. Afterwards, we can generate machine code for the optimized function. The result is shown in figure 5; there is a trivial closure (one with no free variables) representing a very efficient function.

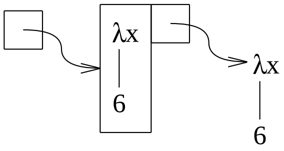

Figure 5

This illustrates that it is not difficult to re-compile a piece of an executing program to take advantage of new information (in this case, the new information was that *f* is bound to $\lambda e.\ 3$). The phases of compilation and execution are now connected as illustrated in figure 6.

5. Interface to the reducer

The previous section describes how a fragment of an executing program may be extracted, reduced, and re-generated. This operation should not be applied indiscriminately; for if the fragment is to be used only once, then it will be much more efficient to simply execute it in its un-optimized form. Instead, the programmer should determine which applications of higher-order functions are likely to be used often, and are

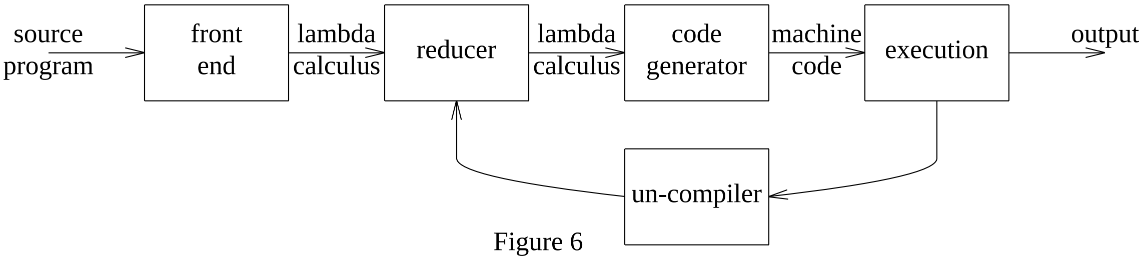

Figure 6

worth optimizing.

For example, a parser generator PG is applied once to a grammar G yielding a parser P. The parser P is applied to many different input strings. Thus, the closure representation of P is surely worth optimizing in the manner described above. On the other hand, if P were just applied to one input string, then the optimization might not be cost-effective.

We will introduce a new operator, **reduce,** into the set of primitive functions available to the programmer. Semantically, **reduce** will be the same as the identity function. That is, for any expression *e*, **reduce**(e) will be semantically indistinguishable from *e*. However, the implementation of **reduce**(e) will be to un-compile, reduce, and re-compile the fragment *e* as described in the previous section.

It is often the case that the programmer can easily identify the particular sub-expressions that will be applied to many different arguments. All that is necessary is to introduce an application of **reduce** to these subexpressions, and they will be ‘‘optimized’’ at runtime. Of course, the **reduce** operation is quite costly, but this cost will be more than balanced by the many applications of the optimized value, which are much cheaper than the corresponding applications of the unreduced value would have been.

A very important property of the **reduce** operator is that it is semantically transparent. Because **reduce** does not change the semantics of its argument in any way, all program transformations and proof rules continue to work in its presence. Any program that works correctly without **reduce** will produce the same results (although perhaps more quickly) when **reduce** is judiciously inserted.

6. An example in detail

Consider the implementation of a spreadsheet program. The user enters formulas into the cells, and then these formulas are evaluated numerically. In most spreadsheets, the evaluation is done by an interpreter, with a consequent penalty in execution time. Because the formulas are evaluated so many times, an obvious optimization is to compile them into machine code; the penalty in this case will be the reduced portability, readability, and provability of the spreadsheet program.

By re-opening closures, we can effectively compile the formulas into machine code, without introducing any machine-dependence or semantic ugliness into the spreadsheet program. We will use the back end of the functional-language compiler to optimize the closures arising from parsing the formulas. This is particularly notable because the language of spreadsheet formulas need not be closely tied to the implementation language.

Assume that the spreadsheet formulas have this syntax and semantics:

$$\begin{aligned}
&\text{exp} \rightarrow \text{exp} + \text{exp}\\
&\text{exp} \rightarrow \text{CELL [exp]}\\
&\text{exp} \rightarrow \text{NUM}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{sub}\text{: } \; \mathit{Array} \times \mathit{Value} \longrightarrow \mathit{Value}\\
&\mathbf{plus}\text{: } \; \mathit{Value} \times \mathit{Value} \longrightarrow \mathit{Value}\\
&E[[\text{exp}]] : \; \mathit{Array} \longrightarrow \mathit{Value}\\
&N[[\text{NUM}]] : \; \mathit{Value}
\end{aligned}$$

$$\begin{aligned}
&E[[\text{exp}_1 + \text{exp}_2]] \; = \; \lambda e.\; E[[\text{exp}_1]]\,e \;\mathbf{plus}\; E[[\text{exp}_2]]\,e\\
&E[[\text{CELL}[\text{exp}]]] \; = \; \lambda e.\; e \;\mathbf{sub}\; E[[\text{exp}]]\,e\\
&E[[\text{NUM}]] \; = \; \lambda e.\; N[[\text{NUM}]]
\end{aligned}$$

A conventional implementation might translate the cell formulas into polish notation, and then interpret the polish code. In a functional language, we may use the denotational semantics to express the formulas directly in terms of higher-order functions. Assuming that **plus** and **sub** are primitives of the implementation language, we can use the semantic equations above as the parsing actions for the corresponding grammar reductions, as in a semantics-directed compiler. In a YACC-like notation, this would look like:

```
exp: exp + exp              { lambda e . plus($1(e),$3(e)) }
exp: CELL [ exp ]           { lambda e . sub(e, $3(e)) }
exp: NUM                    { lambda e . $1 }
```

Let us then parse a simple expression:

CELL[1] + 3

The closure representation of this expression's translation is shown in Figure 7. At this point, the expression has technically been ''compiled to machine code,'' but the machine program is certainly suboptimal; it contains several function calls as well as the fundamental subscript and add operations.

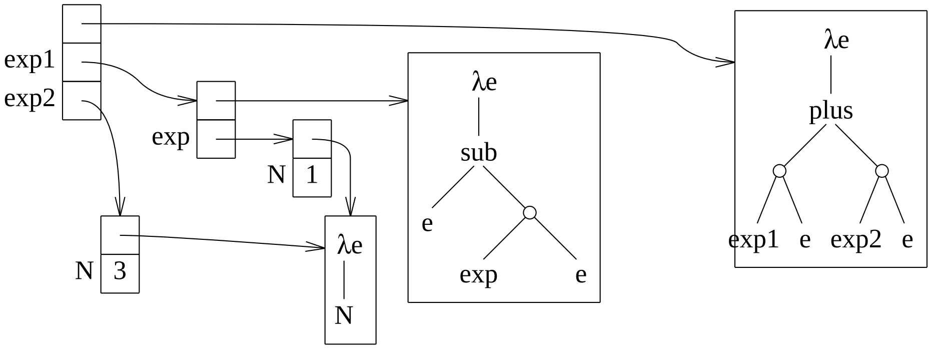

Figure 7

If we apply the **reduce** operator to the closure, we will obtain an equivalent machine program that looks much more like a ''compiled'' expression (Figure 8). Recall that a lambda-expression in a box denotes the machine-code translation of that lambda-expression. The function in figure 8 is a straightforward in-line compilation of the original expression.

Thus, the spreadsheet program can compile cell formulas into efficient machine code by using the back end of the implementation language's compiler. This is done without any complicated interface: neither source code nor any form of ''intermediate language'' needs to be generated. Instead, the semantics of formulas are simply and cleanly expressed in the functional language as higher-order functions, and **reduce** is applied to each cell's formula.

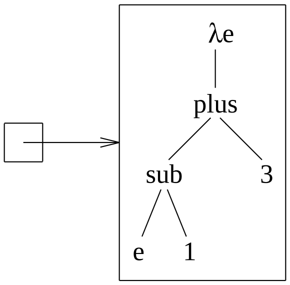

Figure 8

7. Safe and effective reducers

Reducers for the pure lambda calculus are relatively well understood[4]. Efficiency and termination (when possible) may be achieved by finding leftmost β-redexes and contracting them. In a compiler for a functional language, however, reducing expressions may be much more complicated. We desire that the reducer always terminate, and that the order of side-effects be preserved.

In most programming languages it is permissible to write programs that do not terminate. However, we expect that the compiler itself will terminate when compiling these programs. This is achieved by having the compiler avoid the evaluation of loops and recursions when translating the program into machine language. For most languages, including the typed lambda calculus, this is sufficient to guarantee that the compiler will terminate. Of course, it is usually desired that the compiler not only terminate, but do so in a reasonable amount of time.

Similarly, we require that the **reduce** operator be total, since it is supposed to be semantically equivalent to the identity function. This means that **reduce** must similarly tread carefully among loops and recursions. One way of accomplishing this is to avoid the expansion of fixed-point functions by **reduce.** This will suffice — for example, the formulas shown in section 6 will be reduced by such an implementation — but it may be less powerful than we would like.

For a parser generator like YACC, it turns out that the primary operations that can be optimized by a **reduce** operator are the extraction of values from the semantic stack, guided by a list of symbols representing the production. This list of symbols is of finite and known length, and there is a fixed-point that must be expanded once for each element of the list. So a **reduce** function that is powerful enough to do a nontrivial job on a parser generator, must expand (‘‘unroll’’) fixed-points a constant number of times in some circumstances. A reducer that can do this appropriately is certainly nontrivial, and should be the subject of further research. Note that the LR(1) table construction is *not* done by the reducer. This would be extremely costly; by putting the call to **reduce** outside of the table-construction part of the parser-generator, only the application of the tables to the generic parser is evaluated in the reducer, as discussed above.

Another problem in implementing the reducer is that many functional languages have occasional side-effects. While some implementations of purely functional languages do exist, the idea of re-opening closures can be made to apply to both pure and impure functional languages. Since the side-effects are sparse in many impure functional programs, there is still much room for the reducer to work; but because the side-effects do exist, the reducer must check for them before making reductions that might interchange their order. This adds significantly to the complexity of the reducer.

8. Applications

Closure reduction is a simple and powerful technique with many applications. As the previous section hints, the **reduce** operator can serve as the implementation of semantics-directed compilers for arbitrary programming languages. All that is necessary is to express the semantics of the language in the lambda calculus in such a way that the normal forms of semantic parses are reasonably efficient. This, of course, is easier said than done — one can never produce a compiler of any sort without some careful thought.

Such a semantics-directed compiler might be significantly more efficient than similar compilers in the literature[678]. These compilers do all of their compile-time evaluation by graph-reduction, producing a normal form for the input program which is then translated to machine code. A semantics-directed compiler implemented using **reduce** could do most of its evaluation (looking up identifiers in environments, etc.) by closure-code evaluation; the result would be a closure to which it would apply **reduce.** The output of **reduce** would be a normal form of exactly the kind produced by the earlier semantics-directed compilers (subject to the limitations described in section 7). But because most of the compile-time evaluation would be done in (efficient) closure-code execution, the compiler itself would be much faster.

Section 6 shows that even the simplest of programs may have some use for a bit of denotational semantics. But the recompilation of partial results has application wherever higher-order functions are used, as long as the result-value of the higher-order functions is applied repetitively. Closure re-opening is a general and powerful partial evaluation technique.

References

1. S. C. Johnson, ‘‘YACC -- yet another compiler compiler,’’ CSTR-32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
2. P. J. Landin, ‘‘The mechanical evaluation of expressions,’’ *Computer J.*, vol. 6, no. 4, pp. 308-320, 1964.
3. Alonzo Church, *The Calculi of Lambda Conversion,* Princeton University Press, 1941.
4. C. P. Wadsworth, ‘‘Semantics and Pragmatics of the Lambda Calculus,’’ Ph.D. Thesis, Oxford University, 1971.
5. Guy L. Steele, ‘‘Rabbit: a compiler for Scheme,’’ AI-TR-474, MIT, 1978.
6. R. Sethi, ‘‘Control Flow Aspects of Semantics Directed Compiling,’’ *Trans. Prog. Lang. and Systems*, vol. 5, no. 4, pp. 554-595, ACM, October 1983.
7. Andrew W. Appel, ‘‘Compile-time Evaluation and Code Generation for Semantics-Directed Compilers,’’ Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA, 1985.
8. Andrew W. Appel, ‘‘Semantics-directed code generation,’’ *Twelfth ACM Symp. on Principles of Programming Languages*, pp. 315-324, ACM, 1985.