Making Lambda Calculus Smaller, Faster

Andrew W. Appel
Princeton University
appel@princeton.edu

Trevor Jim
Massachusetts Inst. of Technology
trevor@lcs.mit.edu

Princeton CS-TR-477-94
November 1994

Abstract

Some optimizations are speculative: they improve execution speed most but not all of the time, or they trade off faster execution for an increase in code size. Other optimizations, such as dead-variable elimination and algebraic simplification, are guaranteed to both reduce code size and speed execution. It makes sense to group all of these "contracting" optimizations together in a single phase of a compiler. This paper describes an efficient implementation of these optimizations for a language based on the lambda calculus, where a particularly interesting optimization of this sort is inlining for those functions called only once. And we prove that our contracting rewrites are confluent.

1 Introduction

The lambda calculus is a language of functions, so one of the most useful optimizations we can perform in a lambda-calculus-based language is function inlining. Inlining a function eliminates the expense of a procedure-call, and instantiating the function arguments may enable other optimizations. But indiscriminate inlining leads to the evaluation of the entire program at compile time, which can lead to code blowup or non-terminating compilation.

A simple solution to this problem is to inline only those functions that are used exactly once and whose actual parameters are just atoms (variables or literals). After the function has been inlined, its definition can be deleted, resulting in a smaller program. It makes sense to perform this optimization in concert with other optimizations that are guaranteed to make the code smaller, such as dead-variable elimination, and δ-reduction (the evaluation of side-effect-free primitive operators whose arguments are constants).

All of these optimizations either depend critically on the usage counts of variables, or change usage counts of variables, or both. Thus there is a challenge in applying them simultaneously and efficiently. We have previously described [2, 1] the *Contract* phase of the Standard ML of New Jersey compiler, which implements these optimizations by a naive algorithm. The naive *Contract* is effective: it improves the speed of the generated code by a factor of 2.5 [1, p. 183]. However, it is also expensive in terms of compile time. In this paper, we show a simple and practical improvement to the *Contract* algorithm that allows it to accomplish the same result in less time.

We also describe the *Eta* phase of the compiler, which performs another optimization guaranteed to make the program smaller. For practical reasons we have found it sensible to keep the *Eta* and *Contract* phases separate.

Speculative inlining (inlining of functions used more than once) is performed by a separate *Expand* phase [1, ch. 7], which we do not describe here.

Standard ML of New Jersey does contraction on a continuation-passing-style representation of programs; but our efficient contraction algorithms also work on ordinary lambda calculus. We will describe both versions.

2 Contraction

The *Contract* phase performs the following optimizations:

- Dead-variable elimination: any side-effect-free definition of a variable that is not subsequently used can be removed.
- δ-reduction: in a program such as

  $$\begin{array}{l}\textbf{let val } r = \langle v_1, \ldots, v_n\rangle \\ \textbf{in } \ldots \textbf{ let val } x = \#i(r) \textbf{ in } M\end{array}$$

  we can remove the definition of x and substitute v_i for every occurrence of x in M.
- Function inlining: a definition $\textbf{fun } f(\vec{p}) = B$ can be removed and an occurrence $f(\vec{a})$ can be

replaced by $B[\vec{a}/\vec{p}]$ if there are no other occurrences of f.

Other similar optimizations are also performed. We say that a "contract-normal form" contains none of these "redexes."

For these examples, we are using a variant of Continuation-Passing Style notation [2, 1] instead of ordinary lambda calculus. This ensures, for example, that the v_i replacing x in the δ rule is either a variable name or a constant, and thus δ-reduction can only reduce the size of the program. In section 7 we show how these ideas apply to ordinary lambda calculus.

These redexes are not very local. For example, to determine that $f(\vec{p})$ is eligible for reduction requires not only knowing how f is declared (whether it is an explicit function or is passed in as a parameter) but whether f has any other uses.

Thus, contraction is implemented in two passes. The first pass gathers information (a mapping of variable-name to function- or record-definition, and the usage count of each variable), and the second pass performs the reductions.

The contraction of a redex often produces new redexes. For example, our rule for δ-reduction removes a reference to the record r, which may then become a candidate for dead-variable elimination. This sort of dependency makes it difficult to perform all contractions in one pass. In fact, if we consider a "pass" over an expression tree as "down to the leaves and then back up to the root," it is provably impossible to produce a contract-normal form in one down-and-up pass [1, pp. 78–80], or any constant number of such passes (see section 4).

We were led astray by this theorem. We reasoned that if a bounded number of passes could not do the job, then several passes are necessary; and thus any reasonable multi-pass algorithm would suffice. Therefore we used the following strategy in our initial code optimizer:[1]

1. First, *Contract* is performed repeatedly until no more (or few more) redexes are contracted.

2. Then the speculative inlining heuristics of *Expand* are applied.

A pass of *Contract* can create *Expand* redexes, and vice versa; so steps 1 and 2 are repeated until quiescence. Two or three *Contract* passes are performed for each round of *Expand*; this is expensive.

[1] Up to version 0.96 of the compiler.

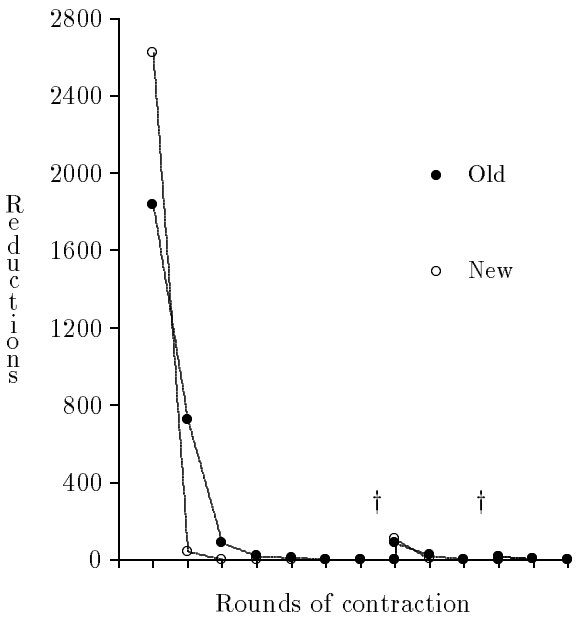

Figure 1: Redexes contracted in each round
† Each of the two rounds of in-line expansion introduces new redexes.

3 A better Contract

We have recently improved the *Contract* phase to be a quasi-one-pass algorithm. We do this by recording the effect of each optimization on usage counts, and by changing the order in which optimizations are applied. As a result we contract the vast majority of redexes in one pass, resulting in a program with very few contract redexes.

The first improvement is to carefully maintain usage counts. For example,

- In dead-variable elimination: if **fun** $f(\vec{p}) = B$ is deleted because f is a dead variable, the usage counts of the free variables of B are decremented.

- In δ-reduction: when we replace

  let val $r = \langle \vec{v} \rangle$
  in ... **let val** $x = \#i(r)$ **in** M

  by **let val** $r = \langle \vec{v} \rangle$ **in** $M[v_i/x]$, we decrement the count of r and adjust the count of v_i according to how many times x appears in M.

- In function inlining: a definition **fun** $f(\vec{p}) = B$ is removed and an occurrence $f(\vec{a})$ is replaced by $B[\vec{a}/\vec{p}]$; so the usage count of each a_i is adjusted according to how many times the corresponding p_i is used in B.

Previously, we had not decremented usage counts while doing reductions (some incrementing had

Program	Compile Time					Run Time	
	Old	%	New	%	New/Old Ratio	Old	New
Barnes-Hut	57.9	26	54.9	21	0.95	30.57	29.31
Boyer	25.1	20	24.7	22	0.98	2.72	2.76
CML-sieve	37.6	42	33.1	30	0.88	34.58	32.91
Knuth-B.	23.7	47	19.1	33	0.81	7.56	7.30
Lex	41.8	44	35.9	34	0.86	10.45	10.48
Life	7.2	50	6.3	28	0.88	1.46	1.42
Mandelbrot	0.54	13	0.51	11	0.94	17.52	16.97
Yacc	157.3	36	132.8	23	0.84	4.39	4.27
Ray	17.3	36	15.5	16	0.89	23.53	22.75
Simple	82.2	57	63.6	40	0.77	15.53	16.26
VLIW	236.6	50	183.9	33	0.78	13.69	13.10
Average					0.87		

Total compile time, percentage of compile time taken by optimization (*Eta, Contract-Expand*), and execution time are shown for each benchmark under "old" (multi-pass contract) and "new" (one-pass contract) compilers.

Figure 2: Compile-time improvement

been necessary just to avoid erroneous dead-variable eliminations). Consequently, *Contract* would overestimate usage counts, thereby doing fewer β-reductions and dead-variable eliminations than it otherwise could.

The second improvement concerns the order in which we perform dead-variable elimination. The "old" *Contract*, encountering an expression such as **let val** $r = \langle v_1, \ldots, v_n \rangle$ **in** M during its recursive descent, checks whether r is dead *before* processing M. We can achieve better results by performing dead-variable elimination both before *and* after the recursive call on M.

- We remove a dead r *before* recurring on M because it decrements the usage counts of the v_i. This can enable other optimizations; for example, if a v_i is a function called only from M, its usage count decreases and we may be able inline the function.

- A reference to r may occur in M, but be removed during the processing of M. Thus the earliest we can remove r is *after* recurring on M. Removing r may now cause one of the v_i to become dead, cascading this optimization on the way up. It turns out to be quite common to have long chains of variables that can be removed going up.

The "new" *Contract* examines usage counts and performs dead-variable elimination both before and after each recursive call. Consequently, for some inputs the number of passes required by the new *Contract* to reach contract-normal form is a factor of N better than that of the old *Contract*, where N is the input size.

Figure 1 shows an example (compilation of the SML-Lex lexical analyzer generator). The old *Contract* (solid circles) reduces 1839 redexes in the first pass, 722 redexes in the second pass, 85 in the third, and so on. The new *Contract* (white circles) reduces 2621 in the first pass, so that only 43 are left for all remaining passes.

We still cannot remove all the redexes in just one pass. But since the new *Contract* leaves so few residual redexes, it is reasonable to perform just one *Contract* pass before proceeding with *Expand*. There will still be several *Contract-Expand* rounds; but now each round will have only one *Contract* pass instead of two or three. Since *Contract* is expensive, this should save compile time.

Figure 2 shows that using the new *Contract*, all of the benchmark programs[2] are compiled faster, by an average of 13%.

The quality of the code generated by the new *Contract* seems to be just as good as that of the code generated by the old *Contract*: execution time decreases by $1.8\% \pm 2.8\%$—the average decrease in execution time is less than the variance. This is as expected: the new algorithm typically contracts as many redexes in its one round as the old algorithm contracts in three.

Why is there any change in execution time at all? Neither algorithm reduces programs to contract-normal form (because the required extra rounds of *Contract* would be too expensive [1, p. 192]); each leaves a (slightly different) set of residuals. And

[2]From the benchmark set used by Shao and Appel[4].

the speculative inlining heuristics of *Expand* can be quite sensitive to small changes in the input.

4 Asymptotic complexity of Contract

Both the old and the new *Contract* algorithms reduce expressions to contract-normal form in worst case time complexity $\Theta(N^2)$.

The upper bound is easily established by considering separately the cost of finding redexes and the cost of performing contractions. We find a redex by making a down-and-up pass over the expression tree. Each pass takes time $\Theta(N)$ and finds at least one redex (if normal form has not yet been reached). Contracting a redex makes the graph smaller, so there are at most N contractions, and therefore at most N passes. This gives an upper bound of $O(N^2)$ on the time spent finding redexes. The cost of performing a contraction (substitution and updating usage counts) is at worst $O(N)$, and there are at most N contractions to perform, so the total cost of performing contractions is $O(N^2)$. The cost of the algorithm as a whole is the sum of these costs, or $O(N^2)$.

A simple example demonstrates the lower bound:

$$
\begin{array}{ll}
\textbf{let} & \textbf{fun}\ f_1(x_1, y_1) = 0 \\
& \quad \textbf{fun}\ f_2(x_2, y_2) = 0 \\
& \quad\quad \vdots \\
& \quad \textbf{fun}\ f_N(x_N, y_N) = 0 \\
& \quad \textbf{fun}\ h_1() = 0 \\
& \quad \textbf{fun}\ h_2() = f_2(f_2, f_2) \\
& \quad \textbf{fun}\ h_3() = f_3(f_3, f_3) \\
& \quad\quad \vdots \\
& \quad \textbf{fun}\ h_N() = f_N(f_N, f_N) \\
& \quad \textbf{fun}\ g_1() = h_1 \\
& \quad \textbf{fun}\ g_2() = f_1(g_1, h_2) \\
& \quad \textbf{fun}\ g_3() = f_2(g_2, h_3) \\
& \quad\quad \vdots \\
& \quad \textbf{fun}\ g_N() = f_{N-1}(g_{N-1}, h_N) \\
\textbf{in} & \quad g_N
\end{array}
$$

In the ith downward pass of the new *Contract*, the body of f_i is inlined in the application $f_i(g_i, h_{i+1})$ because the usage count of f_i is 1. On ith upward pass, functions g_i and h_i are removed because they are dead variables, reducing the usage count of f_{i+1} to 1. N passes are required to reach normal form, each taking linear time, giving $\Omega(N^2)$ as the lower bound.

This pathological case cannot be typical, given the data in Figure 1. A much more typical case, on which the old *Contract* took N passes and the new *Contract* takes one pass, is:

$$
\begin{array}{ll}
\textbf{let} & \textbf{val}\ r_1 = \langle 0, 0 \rangle \\
& \quad \textbf{val}\ r_2 = \langle r_1, 0 \rangle \\
& \quad\quad \vdots \\
& \quad \textbf{val}\ r_N = \langle r_{N-1}, 0 \rangle \\
\textbf{in} & \quad 0
\end{array}
$$

5 A linear-time algorithm

We also have an algorithm that reduces expressions to contract-normal form in linear time, in the worst case. The idea is to represent a program as a doubly linked tree of nodes, and maintain a doubly linked list of the occurrences of each variable. The use of in-place updating allows us to contract redexes in any order, freeing us from the restrictions of the down-and-up passes of *Contract*.

Formally, programs are represented as *expression trees*. We use $D, E, F, \ldots$ to range over *expressions*, which are the nodes of the trees. We use $v, w, x, \ldots$ to range over binding occurrences of variables, and $a, b, c, \ldots$ to range over non-binding occurrences of variables. Binding occurrences of variables will be called simply *variables*, while non-binding occurrences will be called *occurrences*.

We navigate the expression tree via the following functions:

var maps each occurrence to its binding variable;

occ maps each variable to its set of occurrences (represented as a doubly linked list);

def maps each variable to the expression that binds it; and

site maps each occurrence to the expression where it is used.

For example, the program fragment

$$
\begin{array}{l}
\textbf{let val}\ v = \langle q, r \rangle \\
\textbf{in let val}\ w = \#1(v) \\
\qquad \textbf{in}\ \ w(v, r)
\end{array}
$$

is represented by the following expression tree fragment:

$$
\begin{array}{lcl}
D & = & \mathsf{let}\ v = \langle a, b \rangle\ \mathsf{in}\ E, \\
E & = & \mathsf{let}\ w = \#1(c)\ \mathsf{in}\ F, \\
F & = & d(e, f),
\end{array}
$$

where

$$\begin{aligned}
&\mathbf{var}(a) = q,\\
&\mathbf{var}(b) = \mathbf{var}(f) = r,\\
&\mathbf{var}(c) = \mathbf{var}(e) = v,\\
&\mathbf{var}(d) = w,\\
&\mathbf{occ}(v) = \{c, e\},\\
&\mathbf{occ}(w) = \{d\},\\
&\mathbf{def}(v) = \mathbf{site}(a) = \mathbf{site}(b) = D,\\
&\mathbf{def}(w) = \mathbf{site}(c) = E,\\
&\mathbf{site}(d) = \mathbf{site}(e) = \mathbf{site}(f) = F.
\end{aligned}$$

Our example program fragment contains a δ redex that can be reduced by: (1) deleting the definition of w; and (2) substituting q for w. The reduction can be carried out in the expression tree by: (1) updating E to F in D; and (2) updating the set of occurrences of q, by

$$\mathbf{occ}(\mathbf{var}(a)) := \mathbf{occ}(\mathbf{var}(a)) \cup \mathbf{occ}(w).$$

(Recall that $q = \mathbf{var}(a)$).

Notice that this update can be performed in constant time, even if $\mathbf{var}(a)$ is not known. The occurrence a is part of the doubly linked occurrence list of some unique variable, in this case, q. From a we can find its neighbors in the list, and then we can remove a from the list and splice the doubly linked list $\mathbf{occ}(w)$ in its place, all without knowing q.

The reason we might not know $\mathbf{var}(a)$ is that it is too expensive, in general, to update the **var** function to maintain the invariant $g \in \mathbf{occ}(x)$ iff $\mathbf{var}(g) = x$, for any occurrence g and variable x. In our update above, for example, it would require visiting all the elements of $\mathbf{occ}(w)$; and we might have to perform **var** updates many times for a single occurrence. We describe below how we obtain **var** when necessary while staying within our linear time bound.

Figure 3 shows the algorithm. The algorithm maintains a set redexes, each of which has one of the following forms:

inline(v) marks a function bound to v that can be inlined;

dead(v) marks a variable v that has no occurrences; and

select(a) marks an occurrence a of a record which is being selected.

Redexes in the set may be removed and reduced in any order, and reduction may add newly discovered redexes to the set.

Much of the work is done by the two auxiliary functions, **delete** and **subst**. **Delete**(a) removes a from the occurrence list of $\mathbf{var}(a)$. This can be done in constant time, just as for the update

Auxiliary functions:

delete(a) =
  ; remove a from occurrence list
  $\mathbf{occ}(\mathbf{var}(a)) := \mathbf{occ}(\mathbf{var}(a)) - \{a\}$
  ; check for new redexes
  if $|\,\mathbf{occ}(\mathbf{var}(a))\,| \leq 1$
    compute $v = \mathbf{var}(a)$
    if $|\,\mathbf{occ}(v)\,| = 0$
      add $\mathbf{dead}(v)$ to redex set
    if $\mathbf{occ}(v) = \{c\}$, and $\mathbf{site}(c)$ is $c(\vec{c_j})$,
    and $\mathbf{def}(v)$ is let $v(\vec{w_i}) = E$ in F
      add $\mathbf{inline}(v)$ to redex set

subst(w, a) =
  ; check for new record redexes
  if $\mathbf{var}(a)$ defined *; a is a record*
    for each b in $\mathbf{occ}(w)$
      $\mathbf{var}(b) := \mathbf{var}(a)$ *; **var** update*
      if $\mathbf{site}(b)$ is let $x = \#i(b)$ in D
        add $\mathbf{select}(b)$ to redex set
  ; perform substitution
  $\mathbf{occ}(\mathbf{var}(a)) := \mathbf{occ}(\mathbf{var}(a)) \cup \mathbf{occ}(w)$

Reduction algorithm:

while redex set is not empty
  remove r from redex set
  case r of
  dead(v) :
    if $\mathbf{def}(v)$ is D = let $v(\vec{w_i}) = E$ in F
      splice F in place of D in expression tree
      for each occurrence a in E
        delete(a)
    if $\mathbf{def}(v)$ is D = let $v = \langle b_1, \ldots, b_n \rangle$ in E
      splice E in place of D in expression tree
      for $1 \leq i \leq n$
        delete(b_i)
  inline(v) :
    $\mathbf{def}(v)$ is D = let $v(w_1, \ldots, w_k) = E$ in F
    $\mathbf{occ}(v) = \{a\}$
    $\mathbf{site}(a)$ is $G = a(b_1, \ldots, b_k)$
    splice F in place of D in expression tree
    splice E in place of G in expression tree
    for $1 \leq i \leq k$
      subst(w_i, b_i)
      delete(b_i)
  select(a) :
    ; a is a record, so $\mathbf{var}(a)$ *is defined*
    $\mathbf{var}(a) = v$
    $\mathbf{def}(v)$ is let $v = \langle b_1, \ldots, b_n \rangle$ in D
    $\mathbf{site}(a)$ is E = let $x = \#i(a)$ in F
    splice F in place of E in the expression tree
    subst(x, b_i)
    delete(a)

Figure 3: Linear-time *Contract* algorithm

above. Deleting an occurrence can create new dead-variable or function inlining redexes, so **delete** also checks for this. Note that this involves testing the cardinality of occurrence sets; but we only need to know whether the cardinality is zero, one, or greater than one. This test can be done in constant time on doubly linked lists. **Delete** also computes **var**(a), but only when the occurrence list is of length 1 or less. If we give each occurrence list a "header" node that indicates the **var** value, we can compute **var** from an occurrence just by searching down the list. When the list is of length 1 this takes constant time. Thus **delete** as a whole is a constant time operation.

Subst(w, a) substitutes **var**(a) for w by updating the occurrence list of **var**(a), as described above. **Subst** can create new **select** redexes: **var**(a) may be bound to a record, and some occurrence b of w may be selected from. When we later need to reduce the redex **select**(b), we will have to compute **var**(b). In this case, b may be only one of many occurrences of the record variable, so that we cannot use the trick of searching down the occurrence list for the header node. Instead, we will faithfully maintain **var** for every occurrence of a record. This means updating **var** for an occurrence in **subst** when splicing it into the occurrence list of a record. Once an occurrence is bound to a record, it can never be rebound; so a **var** update will be performed at most once per occurrence. Thus the total cost of maintaining **var** for records is at most $O(N)$.

We can now analyze the total running time of the algorithm. It is the sum of the times needed to reduce all the redexes in the input.

To reduce a redex **dead**(v), we must first remove its defining expression from the expression tree, which takes constant time. We must also traverse the definition of v, removing each occurrence in the definition from its occurrence list. Traversing a dead definition takes time linear in the size of the definition. But we can delete any given definition or occurrence only once; so over a complete run of the algorithm, the total time spent reducing dead-variable redexes is $O(N)$.

Reducing a redex **inline**(v) involves deleting a call expression $a(b_1, \ldots, b_k)$ from the expression tree (constant time), and performing k substitutions and deletions. But any call can be inlined at most once; so the total time spent on inlining is $O(N)$, plus the cost of the **var** updates performed by **subst**.

The reduction of a redex **select**(a) involves one substitution and one deletion, and at most $O(N)$ **select** redexes can be reduced by the program. Thus the total time spent on **select** redexes is $O(N)$, plus the cost of the **var** updates performed by **subst**.

We have already seen that the total cost of **var** updates is $O(N)$, and so the algorithm runs in worst-case linear time. Since it is trivial to construct an example with $c \cdot N$ redexes, the time complexity is $\Theta(N)$.

We believe that the *Expand* phase could be integrated into this algorithm, so that each in-line expansion of a function called more than once would also take constant (or near-constant) time.

We have not implemented this algorithm. It spends all its time doing in-place updates of doubly linked lists and of expression tree nodes. This style of programming, while implementable straightforwardly in ML using `ref` variables, does not mesh well with the rest of our compiler. There is a significant advantage, in ease of correct implementation and readabilty, in a style of optimization that uses rewriting by structural induction. The near-linear *Contract* described in this paper is easily implemented in such a style; the linear-time algorithm is not. Since the near-linear algorithm finds almost all redexes in one linear-time pass (as shown in figure 1, it's practically as good.

6 Eta-reduction

In our intermediate language, eta-reduction is the "copy propagation" of function definitions. A definition of the form

$$\textbf{fun } f(x_1, \ldots, x_n) = g(x_1, \ldots, x_n)$$

simply assigns a new name f to the function g; we can remove the definition of f, and use g for f in the rest of the program.[3] These eta redexes can be introduced by the programmer, but are more commonly introduced by code transformations performed by the compiler.

Note that contracting an eta redex can create further redexes:

$$\begin{array}{l}\textbf{let fun } f(x) = \textbf{let fun } g(y) = h(y) \\ \qquad\qquad\qquad\quad \textbf{in } g(x) \\ \textbf{in } \ldots\end{array}$$

Here we can reduce one eta redex, removing the definition of g and using h instead; this produces a new redex, **fun** $f(x) = h(x)$.

In contrast with the *Contract* phase, however, our *Eta* phase can produce an eta-normal form in at most two passes. In the first pass, we maintain a renaming map, and traverse the expression from root to leaves and back. When we reach a function definition **fun** $f(x_1, \ldots, x_n) = M$, we first reduce M, obtaining M'. If M' is of the form $g(x_1, \ldots, x_n)$, we

[3] Provided that $g \notin \{f, x_1, \ldots, x_n\}$.

have found an eta redex, so we remove the definition for f and record that we should use g in place of f in the renaming map.

This strategy can fail to produce an eta-normal form in some cases involving mutually recursive functions, for example:

$$\begin{array}{l}\textbf{let fun } f(x_1,\ldots,x_n) = M \\ \quad \textbf{and } g(y_1,\ldots,y_n) = f(y_1,\ldots,y_n) \\ \textbf{in } N.\end{array}$$

Here we first traverse M, obtaining M'; then remove the definition of g; and finally traverse N, renaming g to f. However, g may still appear in M'; we may need to traverse M', renaming g to f.

This seems to be a pathological special case, so our strategy is to defer the traversal of any such M' to a second pass (this also avoids a possible quadratic blowup in execution time). In compiling the 75,000-line SML/NJ compiler, the second pass of *Eta* is never invoked.

Perhaps the *Eta* phase could be combined with the *Contract* phase. But we keep the phases separate for two reasons: we don't find it useful to perform eta-reduction more than once, while we want to perform *Contract* a number of times; and our combined *Contract-Eta* (discarded several years ago) was never free of bugs.

7 Lambda calculus

Although our implementation operates on CPS code, the same ideas apply to the ordinary lambda calculus. In this section we formalize our optimizations as a confluent term rewriting system in a lambda calculus with records.

Free and bound variables and substitution are defined as usual; we refer the reader to Barendregt [3] for basic definitions. We write $\mathrm{fv}(M)$ for the free variables of M, and the substitution of M for x in N is written $N\{x \mapsto M\}$. We consider terms to be equal modulo renaming of bound variables. An *atom* is a variable or constant.

Function inlining can be implemented by the following two rules. The *operator rule* is

$$(\lambda f.C[f])M \to C[M],$$

where $C[\cdot]$ is a context with a single hole in operator position ($C[\cdot] = C'[([\cdot]N)]$ for some $C'[\cdot]$ and N), f does not appear in $C[\cdot]$, and no variable in $\mathrm{fv}(M)$ is bound in $C[\cdot]$. The *operand rule* is

$$(\lambda x.M)a \to M\{x \mapsto a\},$$

for any atom a.

The dead-variable elimination rule is

$$(\lambda x M)N \to M,$$

where $x \notin \mathrm{fv}(M)$.

The rule for selecting fields from records is

$$(\lambda r.C[\#i(r)])\langle P_1 \ldots P_n\rangle \to (\lambda r.C[P_i])\langle P_1 \ldots P_n\rangle,$$

where $C[\cdot]$ is a context with a single hole, r is not bound in $C[\cdot]$, and P_i is an atom not bound by $(\lambda r.C[\cdot])$.

The η rule is

$$\lambda x.(M\,x) \to M,$$

where x is not free in M.

Finally, let $\to$ be the reflexive, compatible closure of the rewrite rules above. This defines a rewrite system appropriate for a call-by-name language; the usual restriction of the η and operator rules gives a rewrite system compatible with a call-by-value style of evaluation.

We can show that both styles of evaluation result in confluent rewriting systems.

Theorem (Confluence): *If $M_0 \stackrel{\Delta_1}{\to} M_1$ and $M_0 \stackrel{\Delta_2}{\to} M_2$, there is some M_3 such that $M_1 \to M_3$ and $M_2 \to M_3$.*

Confluence can be proved by a long syntactic case analysis. We will omit the proof, but give here a few properties of $\to$ that are crucial to the proof. All of the properties can be proved by simple case analysis of the rewrite rules.

Property 1 *If Δ is a redex with contractum Δ', then $\mathrm{fv}(\Delta') \subseteq \mathrm{fv}(\Delta)$.*

Property 2 *If $\Delta \to \Delta'$, x is a variable, and a is an atom, then $\Delta\{x \mapsto a\} \to \Delta'\{x \mapsto a\}$.*

Property 3 *No atom is a redex.*

Property 4 *A descendant of an operator is an operator.*

Definition: Suppose P is a subterm of a redex Δ. We say *P is the operator of Δ* if $\Delta \equiv (PN)$ for some N. We say *P is inert in Δ* if none of the following cases holds:

- $P \equiv \Delta$;
- P is the operator of Δ;
- $\Delta \equiv (\lambda x.M)a$ is an operand redex and P is the subterm a; or
- $\Delta \equiv (\lambda r.C[\#i(r)])\langle P_1, \ldots, P_n\rangle$ and P is the atomic subterm P_i.

Property 5 *Suppose Δ is a redex with contractum Δ', and P is an inert subterm of Δ. Then P has at most a single descendant in Δ'.*

Property 6 *Suppose Δ is a redex with contractum Δ', P is an inert subterm of Δ, and no free variable of P is bound in Δ. Then*

$$\Delta \equiv C[P] \rightarrow C'[P] \equiv \Delta',$$

where $C[\cdot]$ is the context surrounding P in Δ, and $C'[\cdot]$ is the context surrounding the descendant of P in Δ'. Note that the descendant of P, if any, is exactly P.

Furthermore, $C[N] \rightarrow C'[N]$ for any N with no free variables bound in Δ.

Proof of Confluence: The details are in the Appendix.

8 Conclusion

Our improvements to the *Contract* phase of Standard ML of New Jersey yield a "functional style" algorithm that reduces "almost all" redexes in linear time, and we also describe an "updating style" worst-case linear-time algorithm. Our improved *Eta* phase reduces all η redexes in linear time. The algorithms that they replace both took worst-case quadratic time. The new algorithms allow us to greatly reduce the number of *Contract* and *Eta* passes performed by the compiler, without compromising the speed of the generated code.

Our rewriting system is confluent (Church-Rosser), so the linear-time algorithm is nicely deterministic. Our near-linear algorithm, when run for only one pass, does not take advantage of confluence because it does not reduce to normal forms.

References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Sixteenth ACM Symp. on Principles of Programming Languages*, pages 293–302, New York, 1989. ACM Press.

[3] Hendrik P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam, 1981.

[4] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, pages 150–161. ACM Press, 1994.

Appendix: Proof of Confluence

If Δ_1 and Δ_2 are disjoint, the result follows because $\rightarrow$ is compatible.

If $\Delta_1 \equiv \Delta_2$, in most cases $M_1 \equiv M_2$ already. The exception is when Δ_1 and Δ_2 are different record selection redexes. In this case,

$$\Delta_1 \equiv \Delta_2 \equiv (\lambda r.C[\#i(r)][\#j(r)])\langle P_1, \ldots, P_n\rangle$$

for some context $C[\cdot][\cdot]$ with two holes, and

$$\begin{aligned} M_0 &\equiv C_0[(\lambda r.C[\#i(r)][\#j(r)])\langle P_1, \ldots, P_n\rangle], \\ M_1 &\equiv C_0[(\lambda r.C[P_i][\#j(r)])\langle P_1, \ldots, P_n\rangle], \\ M_2 &\equiv C_0[(\lambda r.C[\#i(r)][P_j])\langle P_1, \ldots, P_n\rangle], \end{aligned}$$

for some $C_0[\cdot]$ with a single hole. Then define

$$M_3 \equiv C_0[(\lambda r.C[P_i][P_j])\langle P_1, \ldots, P_n\rangle],$$

and $M_1 \rightarrow M_3$ and $M_2 \rightarrow M_3$ by record selection.

Otherwise one of the redexes properly contains the other. Assume w.l.o.g. that Δ_2 is properly contained in Δ_1, that Δ_2' is the contractum of Δ_2, and $M_0 \equiv C_0[\Delta_1]$ for some $C_0[\cdot]$ with a single hole. The proof is by cases on the redex Δ_1.

- $\Delta_1 \equiv (\lambda x.M)N$ is a dead-variable redex.

  If $\Delta_2 \subset M$, choose $C_2[\cdot]$ so that $M \equiv C_2[\Delta_2]$. Then

  $$\begin{aligned} M_0 &\equiv C_0[(\lambda x.C_2[\Delta_2])N], \\ M_1 &\equiv C_0[C_2[\Delta_2]], \\ M_2 &\equiv C_0[(\lambda x.C_2[\Delta_2'])N]. \end{aligned}$$

  So let $M_3 \equiv C_0[C_2[\Delta_2']]$. $M_1 \stackrel{\Delta_2}{\rightarrow} M_3$ by compatibility of $\rightarrow$. And by Property 1, $x \notin \mathrm{fv}(C_2[\Delta_2'])$, so $M_2 \rightarrow M_3$ by the dead-variable rule.

  Otherwise $\Delta_2 \subset N$, and $M_1 \equiv M$. So if we choose $M_3 \equiv M$, we have $M_1 \rightarrow M_3$ by reflexivity, and $M_2 \rightarrow M_3$ by the dead-variable rule.

- $\Delta_1 \equiv (\lambda f.C_1[f])M$ is an operator redex.

  If $\Delta_2 \subset M$, choose $C_2[\cdot]$ so that $M \equiv C_2[\Delta_2]$. Then

  $$\begin{aligned} M_0 &\equiv C_0[(\lambda f.C_1[f])(C_2[\Delta_2])], \\ M_1 &\equiv C_0[C_1[C_2[\Delta_2]]], \\ M_2 &\equiv C_0[(\lambda f.C_1[f])(C_2[\Delta_2'])]. \end{aligned}$$

  In this case define

  $$M_3 \equiv C_0[C_1[C_2[\Delta_2']]].$$

  We have $M_1 \stackrel{\Delta_2}{\rightarrow} M_3$ by compatibility of $\rightarrow$, and $M_2 \rightarrow M_3$ by the operator rule.

  Otherwise $\Delta_2 \subset C_1[f]$.

If Δ_2 and f are disjoint, then $C_1[f]$ can be written $C_{12}[f][\Delta_2]$, for some context $C_{12}[\cdot][\cdot]$ with two holes. Then

$$\begin{aligned} M_0 &\equiv C_0[(\lambda f.C_{12}[f][\Delta_2])M], \\ M_1 &\equiv C_0[C_{12}[M][\Delta_2]], \\ M_2 &\equiv C_0[(\lambda f.C_{12}[f][\Delta_2'])M]. \end{aligned}$$

Then choose

$$M_3 \equiv C_0[C_{12}[M][\Delta_2']].$$

We have $M_1 \stackrel{\Delta_2}{\to} M_3$ by compatibility of $\to$, and $M_2 \to M_3$ by the operator rule.

Otherwise Δ_2 contains f. Then for some $C_2[\cdot]$ and $C_{12}[\cdot]$, we have $\Delta_2 \equiv C_2[f]$ and $C_1[\cdot] \equiv C_{12}[C_2[\cdot]]$. Then

$$\begin{aligned} M_0 &\equiv C_0[(\lambda f.C_{12}[C_2[f]])M], \\ M_1 &\equiv C_0[C_{12}[C_2[M]], \\ M_2 &\equiv C_0[(\lambda f.C_{12}[C_2'[f]])M], \end{aligned}$$

where $C_2'[\cdot]$ is the context such that $C_2'[f] \equiv \Delta_2'$. Note that by Property 5, $C_2'[\cdot]$ has either one or zero holes. Define

$$M_3 \equiv C_0[C_{12}[C_2'[M]]].$$

If $C_2'[\cdot]$ has no holes, then $M_2 \to M_3$ by dead-variable elimination.

If $C_2'[\cdot]$ has a single hole, then by Property 4, f appears in operator position in $C_2'[f]$. Thus $M_2 \to M_3$ by the operator rule.

Finally, $M_1 \to M_3$ by Property 6.

- $\Delta_1 \equiv (\lambda r.C[\#i(r)])\langle P_1, \ldots, P_n\rangle$ is a record selection redex.

  If $\Delta_2 \subset P_j$, then $j \neq i$, because P_i is an atom, and no atom is a redex (Property 3). So $P_j \equiv C_2[\Delta_2]$ for some context $C_2[\cdot]$. Let $P_k' \equiv P_k$ for $k \neq j$, and $P_j' \equiv C_2[\Delta_2']$. Then

  $$\begin{aligned} M_0 &\equiv C_0[(\lambda r.C[\#i(r)])\langle P_1, \ldots, P_n\rangle], \\ M_1 &\equiv C_0[(\lambda r.C[P_i])\langle P_1, \ldots, P_n\rangle], \\ M_2 &\equiv C_0[(\lambda r.C[\#i(r)])\langle P_1', \ldots, P_n'\rangle]. \end{aligned}$$

  Define

  $$M_3 \equiv C_0[(\lambda r.C[P_i])\langle P_1', \ldots, P_n'\rangle].$$

  Then $M_1 \stackrel{\Delta_2}{\to} M_3$ by compatibility of $\to$, and $M_2 \to M_3$ by record selection (recall $P_i \equiv P_i'$).

  Otherwise $\Delta_2 \subset C[\#i(r)]$.

  The case where Δ_2 and $\#i(r)$ are disjoint is just as for the operator rule above. So assume Δ_2 properly contains $\#i(r)$, and $\Delta_2 \equiv C_2[\#i(r)]$, $C[\cdot] \equiv C_1[C_2[\cdot]]$ for some contexts $C_1[\cdot]$ and $C_2[\cdot]$. Then

  $$\begin{aligned} M_0 &\equiv C_0[(\lambda r.C_1[C_2[\#i(r)]])\langle P_1, \ldots, P_n\rangle], \\ M_1 &\equiv C_0[(\lambda r.C_1[C_2[P_i]])\langle P_1, \ldots, P_n\rangle], \\ M_2 &\equiv C_0[(\lambda r.C_1[C_2'[\#i(r)]])\langle P_1, \ldots, P_n\rangle], \end{aligned}$$

  where $C_2'[\#i(r)] \equiv \Delta_2'$. Note that by Property 5, $C_2'[\cdot]$ has either one or zero holes. Define

  $$M_3 \equiv C_0[(\lambda r.C_1[C_2'[P_i]])\langle P_1, \ldots, P_n\rangle].$$

  If $C_2'[\cdot]$ has zero holes, then $M_2 \equiv M_3$, so that $M_2 \to M_3$. If $C_2'[\cdot]$ has one hole, then $M_2 \to M_3$ by record inlining.

  Finally, $M_1 \to M_3$ by Property 6.

- $\Delta_1 \equiv (\lambda x.M)a$ is an operand redex.

  We must have $\Delta_2 \subset M$. Choose $C_1[\cdot]$ so that $M \equiv C_1[\Delta_2]$. Then

  $$\begin{aligned} M_0 &\equiv C_0[(\lambda x.C_1[\Delta_2])a], \\ M_1 &\equiv C_0[C_1[\Delta_2]\{x \mapsto a\}], \\ M_2 &\equiv C_0[(\lambda x.C_1[\Delta_2'])a]. \end{aligned}$$

  Then let

  $$M_3 \equiv C_0[C_1[\Delta_2']\{x \mapsto a\}].$$

  $M_1 \to M_3$ by Property 2, and $M_2 \to M_3$ by the operator rule.

- $\Delta_1 \equiv (\lambda x.(M\,x))$ is an η redex.

  We must have $\Delta_2 \subset M$. Choose $C_1[\cdot]$ so that $M \equiv C_1[\Delta_2]$. Then

  $$\begin{aligned} M_0 &\equiv C_0[(\lambda x.(C_1[\Delta_2]x))], \\ M_1 &\equiv C_0[C_1[\Delta_2]], \\ M_2 &\equiv C_0[(\lambda x.(C_1[\Delta_2']x))]. \end{aligned}$$

  Define

  $$M_3 \equiv C_0[C_1[\Delta_2']].$$

  Then $M_1 \stackrel{\Delta_2}{\to} M_3$ by compatibility of $\to$, and $M_2 \to M_3$ by η.

End proof.