Shrinking Lambda Expressions in Linear Time

Andrew W. Appel
Princeton University

Trevor Jim
University of Pennsylvania

February 1997

Abstract

Functional-language compilers often perform optimizations based on beta and delta reduction. To avoid speculative optimizations that can blow up the code size, we might wish to use only *shrinking* reduction rules guaranteed to make the program smaller: these include dead-variable elimination, constant folding, and a restricted beta rule that inlines only functions that are called just once.

The restricted beta rule leads to a shrinking rewrite system that has not previously been studied. We show some efficient normalization algorithms that are immediately useful in optimizing compilers; and we give a confluence proof for our system, showing that the choice of normalization algorithm does not affect final code quality.

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 nonterminating 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 the 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 describe simple and practical improvements to the *Contract* algorithm that allow it to accomplish the same result in less time. Because our new algorithms do their optimizing rewrites in a different order than the old algorithm, we have also found it reassuring to prove that our rewrite system is confluent—thus, all the algorithms produce the same output.

2 Syntax

The intermediate code of our compiler is a lambda calculus based on continuation-passing style (CPS). A representative subset of the language is defined by the following grammar:

$$\begin{array}{rll} M ::= & \mathbf{let}\ f(x_1,\ldots,x_n) = M\ \mathbf{in}\ N & \textit{recursive function definition} \\ \mid & f(a_1,\ldots,a_n) & \textit{function application} \\ \mid & \mathbf{let}\ r = \langle a_1,\ldots,a_n\rangle\ \mathbf{in}\ M & \textit{record creation} \\ \mid & \mathbf{let}\ x = \#i(a)\ \mathbf{in}\ M & \textit{record field selection} \end{array}$$

Here M and N range over terms, f, x, and r range over variables, and a ranges over atoms. The only primitive operators we treat here are record creation and selection, and the only atoms here are variables.

Of course, the calculus used by the compiler has more kinds of atoms (such as integer constants), and many more primitive operators. But the δ-rules for primitives such as arithmetic, branching, and constructor discrimination can be implemented in much the same way as the record primitives we discuss here. And all side effects and "observation" of side effects are restricted to particular primitives (they are syntactically evident), so side effects do not complicate optimizations such as dead-variable elimination. Thus our selection of primitives, while limited, suffices to illustrate the complexities of shrinking optimizations.

The syntax of our CPS language enforces an important property: every intermediate value computed by a program is named in the program. In particular, the allowable arguments of functions and primitives are severely restricted. For example, $f(\lambda x M)$ is not a CPS term; anonymous (nameless) functions are prohibited, because they compute a value (a closure). And $f(g(x))$ is not a CPS term, because the value computed by $g(x)$ is not named. The way to write such programs in our CPS language is to first name the complex argument ($\lambda x M$ or $g(x)$), then pass the name as the argument. Besides names, the only other permissible arguments are literals—in other words, all arguments are atoms.

Atomic arguments simplify the task of deciding when to inline. For example, inlining a function application $f(M)$ in a less restricted language may not be sound, because M may have side effects or be nonterminating; but it is always semantics-preserving to inline a function with atomic arguments. And it is easy to calculate the size of inlined function bodies: substituting atoms for formal parameters does not change the size of a term.

Indeed there are several intermediate codes now in use that require function arguments to be atoms: our own CPS [2] (but not the CPS of Steele [10] or Kranz [6]); the Bform of Tarditi [11] (but not the A-normal form of Flanagan et al. [5]); and the "core language" used by Peyton Jones [7].

The *continuation-passing* of our CPS language is not relevant to the shrinking reductions we describe in this paper. For example, Tarditi defines similar reductions in his Bform intermediate language, which is a direct-style calculus. But β-reduction is easier to express in CPS than in Bform, and we have implemented our algorithms in SML/NJ, which uses CPS. So CPS permits both a simpler exposition and real-world performance evaluation.

3 Rewriting rules

A *substitution* is a finite mapping from variables to atoms (but not to terms in general). A substitution may be written as $\{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$ where the x_i are distinct; we use σ to range over substitutions. The application of a substitution to a term is defined as usual (avoiding the capture of free variables), and is written postfix ($M\sigma$ or $M\{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$). Note that if M is a term and σ is a substitution, then $M\sigma$ is a term of the same size as M.

A *context* $C[\cdot]$ is a "term with a hole;" $C[M]$ indicates the term obtained by filling the hole of $C[\cdot]$ with the term M, possibly capturing free variables of M.

The *dead-variable-elimination rules* delete definitions that are not used:

$$\left.\begin{array}{rcl} (\mathbf{let}\ z(x_1,\ldots,x_n) = N\ \mathbf{in}\ M) & \rightarrow & M \\ (\mathbf{let}\ z = \langle a_1,\ldots,a_n\rangle\ \mathbf{in}\ M) & \rightarrow & M \\ (\mathbf{let}\ z = \#i(a)\ \mathbf{in}\ M) & \rightarrow & M \end{array}\right\}\text{where } z \text{ is not free in } M \text{ or } N$$

The *record selection rule* is a kind of constant-folding on field-selection expressions:

$$\left(\begin{array}{l} \mathbf{let}\ r = \langle a_1,\ldots,a_n\rangle \\ \mathbf{in}\ \ C[\ \mathbf{let}\ x = \#i(r)\ \mathbf{in}\ M\ ] \end{array}\right) \rightarrow \left(\begin{array}{l} \mathbf{let}\ r = \langle a_1,\ldots,a_n\rangle \\ \mathbf{in}\ \ C[\ M\{x \mapsto a_i\}\ ] \end{array}\right)$$

For soundness, we must ensure that if the atom a_i is a variable, then it is not captured by a binding in the context $C[\cdot]$; and that $C[\cdot]$ does not rebind r. This is accomplished by requiring that all bound variables be distinct from each other and from free variables. As an added benefit, this also simplifies the implementation of substitution in our compiler.

The *inlining rule* replaces a function call with the body of the function:

$$\left(\begin{array}{l} \mathbf{let}\ f(x_1,\ldots,x_n) = M \\ \mathbf{in}\ \ C[\ f(a_1,\ldots,a_n)\ ] \end{array}\right) \rightarrow \left(\begin{array}{l} \mathbf{let}\ f(x_1,\ldots,x_n) = M \\ \mathbf{in}\ \ C[\ M'\{x_1 \mapsto a_1,\ldots,x_n \mapsto a_n\}\ ] \end{array}\right)$$

where M' is obtained from M by renaming all bound variables to "fresh" variables. Renaming is necessary to preserve distinct bindings.

These rules are the CPS equivalent of the β- and δ-rules of the lambda calculus. In principle, we could use them to do "computation" on CPS, though it is more common to use CPS as an intermediate representation for optimization before translation to machine language.

The demands of optimization are different from those of computation. In particular, we demand that optimization terminate. A simple way of guaranteeing termination is to use only *shrinking reductions*, those that make the term smaller. Clearly the dead-variable rules and the record-selection rule are shrinking reductions. But the inlining rule is not a shrinking reduction.

We are not willing to abandon inlining, because it is such a useful optimization. Therefore we adopt the following *shrinking inlining rule* for functions called exactly once:

$$\left(\begin{array}{l} \textbf{let}\ f(x_1, \ldots, x_n) = M \\ \textbf{in}\ \ C[\ f(a_1, \ldots, a_n)\ ] \end{array} \right) \rightarrow C[\ M\{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}\ ],$$

where f does not appear in $C[\cdot]$, M, or in $\{a_1, \ldots, a_n\}$. Shrinking inlining combines inlining with dead-variable elimination—once the function is inlined into its single call site, it becomes dead and its definition can be deleted. Notice that in contrast to the general inlining rule, renaming is not required, because no duplication of the bindings in M has occurred.

We can simplify our implementation of shrinking inlining by requiring that *any* function called exactly once have its definition deleted. We do this by adding the following *recursive-dead-function* rule:

$$\left(\begin{array}{l} \textbf{let}\ f(x_1, \ldots, x_n) = C[\ f(a_1, \ldots, a_n)\ ] \\ \textbf{in}\ \ M \end{array} \right) \rightarrow M$$

where f does not appear in $C[\cdot]$, M, or in $\{a_1, \ldots, a_n\}$. Although we could have written a more general recursive-dead-function rule (permitting f to be free in $C[\cdot]$ or a_i), these cases don't come up much and we prefer to keep our algorithms simple.

We write $M \rightarrow N$ if N is obtained from M by transforming some subterm by one of our shrinking reductions: dead-variable elimination, including recursive-dead-function elimination; record selection; and shrinking inlining. We write $\rightarrow^*$ for the reflexive and transitive closure of the relation $\rightarrow$. A term M is in *shrink-normal form* if there is no term N such that $M \rightarrow N$.

Our shrinking reduction system is confluent, or Church-Rosser:

Theorem (Confluence): *If $M_0 \rightarrow^* M_1$ and $M_0 \rightarrow^* M_2$, there is some M_3 such that $M_1 \rightarrow^* M_3$ and $M_2 \rightarrow^* M_3$.*

Proof: See Appendix A.

Figure 1: Gathering usage counts; use $\Delta = +1$ to increment.

$$
\begin{array}{ll}
\text{census} & (\Delta,\ \mathbf{let}\ f(x_1,\ldots,x_n) = M\ \mathbf{in}\ N\ ) = \\
& \text{census}(\Delta, M);\ \ \text{census}(\Delta, N) \\
\\
\text{census} & (\Delta,\ f(a_1,\ldots,a_n)\ ) = \\
& \text{Count}_{\text{app}}[\sigma(f)] \leftarrow \text{Count}_{\text{app}}[\sigma(f)] + \Delta \\
& \text{Count}_{\text{esc}}[\sigma(a_i)] \leftarrow \text{Count}_{\text{esc}}[\sigma(a_i)] + \Delta, \quad 1 \le i \le n \\
\\
\text{census} & (\Delta,\ \mathbf{let}\ r = \langle a_1,\ldots,a_n\rangle\ \mathbf{in}\ M\ ) = \\
& \text{Count}_{\text{esc}}[\sigma(a_i)] \leftarrow \text{Count}_{\text{esc}}[\sigma(a_i)] + \Delta, \quad 1 \le i \le n \\
& \text{census}(\Delta, M) \\
\\
\text{census} & (\Delta,\ \mathbf{let}\ x = \#i(a)\ \mathbf{in}\ M\ ) = \\
& \text{Count}_{\text{app}}[\sigma(a)] \leftarrow \text{Count}_{\text{app}}[\sigma(a)] + \Delta \\
& \text{census}(\Delta, M)
\end{array}
$$

The important consequence of confluence is that every CPS program has a unique shrink-normal form. So although the three *Contract* algorithms we describe in this paper apply the shrinking reductions in very different orders, the final output will be identical. Therefore in comparing the algorithms, we only have to compare running times, and not the programs produced by the algorithms.

We have also proved confluence for shrinking reductions on ordinary lambda calculus—where function arguments can be terms, not just atoms [3].

4 A naive *Contract* algorithm

The *Contract* phase of our compiler does just the shrinking reductions: dead-variable elimination, record-field selection, and inlining of functions called only once. Because we compile ML, our optimizer can assume that programs are well typed, so that no program applies a function to the wrong number of arguments, or selects a nonexistent field from a record.

Redexes of the shrinking inlining rule depend on a rather global property: to determine whether an application $f(a)$ should be inlined requires knowing whether f has any other uses.

Thus, contraction is implemented in two passes. The *census* pass (Figure 1) gathers the usage count of each variable, and the *contract* pass (Figure 2) performs the reductions.

The *census* and *contract* passes use several global mapping tables:

Bind A table mapping function variables to (argument,body) pairs, and record

Figure 2: Performing reductions (old algorithm).

$$
\begin{array}{ll}
\text{contract} & (\ \textbf{let}\ f(x_1,\ldots,x_n) = M\ \textbf{in}\ N\ ) = \\
& \text{Bind}[f] \leftarrow ((x_1,\ldots,x_n), M) \\
& \text{if Count}_{\text{app}}[f] \leq 1 \text{ and Count}_{\text{esc}}[f] = 0 \\
& \text{then contract}(N) \\
& \text{else } \textbf{let}\ f(x_1,\ldots,x_n) = \text{contract}(M)\ \textbf{in}\ \text{contract}(N) \\
\\
\text{contract} & (\ f(a_1,\ldots,a_n)\ ) = \\
& \text{if Count}_{\text{app}}[\sigma(f)] = 1 \text{ and Count}_{\text{esc}}[\sigma(f)] = 0 \\
& \qquad\qquad \text{and Bind}[\sigma(f)] = ((x_1,\ldots,x_n), M) \\
& \text{then } \sigma \leftarrow \sigma + \{x_1 \mapsto \sigma(a_1),\ldots,x_n \mapsto \sigma(a_n)\}; \quad \text{contract}(M) \\
& \text{else } \sigma(f)(\sigma(a_1),\ldots,\sigma(a_n)) \\
\\
\text{contract} & (\ \textbf{let}\ r = \langle a_1,\ldots,a_n\rangle\ \textbf{in}\ N\ ) = \\
& \text{Bind}[r] \leftarrow \langle a_1,\ldots,a_n\rangle \\
& \text{if Count}_{\text{esc}}[r] = 0 \\
& \text{then contract}(N) \\
& \text{else } \textbf{let}\ r = \langle\sigma(a_1),\ldots,\sigma(a_n)\rangle\ \textbf{in}\ \text{contract}(N) \\
\\
\text{contract} & (\ \textbf{let}\ x = \#i(a)\ \textbf{in}\ N\ ) = \\
& \text{if Count}_{\text{app}}[x] + \text{Count}_{\text{esc}}[x] = 0 \\
& \text{then contract}(N) \\
& \text{else if Bind}[\sigma(a)] = \langle b_1,\ldots,b_n\rangle \\
& \text{then } \sigma \leftarrow \sigma + \{x \mapsto \sigma(b_i)\}; \quad \text{contract}(N) \\
& \text{else } \textbf{let}\ x = \#i(\sigma(a))\ \textbf{in}\ \text{contract}(N)
\end{array}
$$

variables to tuples of atoms;

σ A substitution mapping variables to atoms;

Count$_{\text{app}}$ A table mapping function variables to their number of occurrences in function-call position, and record variables to their number of occurrences in selected-from position.

Count$_{\text{esc}}$ A table mapping variables to their number of occurrences as record fields or function arguments.

The contraction of a redex often produces new redexes. For example, our record selection rule removes a reference to a record, 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 shrink-normal form in one down-and-up pass [1, pp. 78–80], or any constant number of such passes (see section 6). At most we will need a linear number of passes, since each pass removes at least one node from the expression tree.

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 code optimizer:

> **repeat**
>
> Initialize σ, Bind, $\text{Count}_{\text{app}}$, and $\text{Count}_{\text{esc}}$, to empty.
> Gather usage counts (*census*).
> Perform contractions based on usage counts (*contract*).
>
> **until** no redexes left.

The Glasgow Haskell optimizer uses the same methodology, described by Santos [8] as "Analyse—Simplify—Iterate."

As contractions were done, we did not update the usage counts to reflect the changed program. Since usage counts can increase (by shrinking inlining or record selection) as well as decrease (by any shrinking rule), this might seem dangerous. But the two rules that depend on usage counts are dead-variable elimination and shrinking inlining. The usage count of a dead variable can never increase, so dead-variable elimination is safe with nonupdated usage counts; and if $\text{Count}_{\text{esc}}[f] = 0$ then $\text{Count}_{\text{app}}[f]$ can only decrease, so shrinking inlining is safe with nonupdated usage counts.

The real problem is that the algorithm iterates too many times before reaching shrink-normal form. In practice the last several iterations of the algorithm contract very few redexes, so we adjusted the algorithm to be

> **repeat**
>
> Initialize σ, Bind, $\text{Count}_{\text{app}}$, and $\text{Count}_{\text{esc}}$, to empty.
> Gather usage counts (*census*).
> Perform contractions based on usage counts (*contract*).
>
> **until** only a dozen contractions done in this round.

This loop was a major part of Standard ML of New Jersey's optimizer, up to SML/NJ version 0.96. But as we will show in this paper, keeping the usage counts current is easy and practical, and greatly improves the speed of the compiler.

5 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 shrinking redexes. Our New Contract algorithm uses *ncontract* (Figure 3) in place of *contract*, but with the same *census* function of Figure 1.

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

- In dead-variable elimination: if **let** $f(x) = M$ is deleted because f is a dead variable, the usage counts of the free variables of M are decremented.

- In δ-reduction: when we replace

$$\begin{array}{l} \textbf{let } r = \langle \vec{a} \rangle \\ \textbf{in } \ C[\ \textbf{let } x = \#i(r) \textbf{ in } M\ ] \end{array}$$

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

- In shrinking inlining: a definition **let** $f(\vec{x}) = M$ is removed and an occurrence $f(\vec{a})$ is replaced by $M\{\vec{x} \mapsto \vec{a}\}$; so the usage count of each a_i is adjusted according to how many times x_i is used in M.

Previously, we had not adjusted usage counts while doing reductions. Typically, *Contract* would overestimate usage counts, thereby doing fewer inlinings and dead-variable eliminations than it otherwise could have.

The second improvement concerns the order in which we perform dead-variable elimination. The "old" *Contract*, encountering an expression such as

$$\textbf{let } r = \langle a_1, \ldots, a_n \rangle \textbf{ in } M$$

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 processing M.

- We remove a dead r *before* processing M because it decrements the usage counts of the a_i. This can enable other optimizations; for example, if an a_i is a function called only from M, its usage count decreases and we may be able to 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* processing M. Removing r may now cause one of the a_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.

Our adjustment of usage counts forces us to handle recursive-dead-function and shrinking-inlining redexes more carefully. Consider a definition let $f(\vec{x}) = M$ in N where $\text{Count}_{\text{app}}[f] = 1$ and $\text{Count}_{\text{esc}}[f] = 0$. This is either a recursive-dead-function or shrinking-inlining redex; it doesn't matter to the old *Contract*, which simply discards M and recurses on N. But the new *Contract* must

Figure 3: Performing reductions (new algorithm).

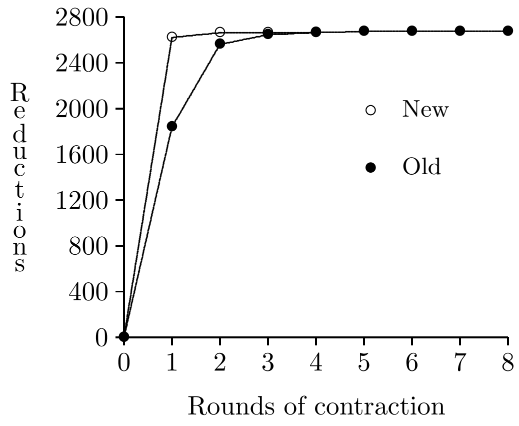
ncontract (**let** $f(x_1, \ldots, x_n) = M$ **in** N) =
Bind$[f] \leftarrow ((x_1, \ldots, x_n), M)$
if $\text{Count}_{\text{app}}[f] = 0$ and $\text{Count}_{\text{esc}}[f] = 0$
then census$(-1, M)$; ncontract(N)
else if $\text{Count}_{\text{app}}[f] = 1$ and $\text{Count}_{\text{esc}}[f] = 0$
then $N' \leftarrow$ ncontract(N)
  if Bind$[f] \neq$ **inlined** then census$(-1, M)$
  N'
else $N' \leftarrow$ ncontract(N)
  if Bind$[f] =$ **inlined** then N'
  else if $\text{Count}_{\text{app}}[f] = 0$ and $\text{Count}_{\text{esc}}[f] = 0$
    then census$(-1, M)$; N'
    else Bind$[f] \leftarrow ()$
      let $f(x_1, \ldots, x_n) =$ ncontract(M) **in** N'

ncontract ($f(a_1, \ldots, a_n)$) =
if $\text{Count}_{\text{app}}[\sigma(f)] = 1$ and $\text{Count}_{\text{esc}}[\sigma(f)] = 0$
    and Bind$[\sigma(f)] = ((x_1, \ldots, x_n), M)$
then $\sigma \leftarrow \sigma + \{x_1 \mapsto \sigma(a_1), \ldots, x_n \mapsto \sigma(a_n)\}$
  $\text{Count}_{\text{app}}[\sigma(a_i)] \leftarrow \text{Count}_{\text{app}}[\sigma(a_i)] + \text{Count}_{\text{app}}[x_i] - 1, \qquad 1 \leq i \leq n$
  $\text{Count}_{\text{app}}[\sigma(f)] \leftarrow 0$
  Bind$[\sigma(f)] \leftarrow$ **inlined**
  ncontract(M)
else $\sigma(f)(\sigma(a_1), \ldots, \sigma(a_n))$

ncontract (**let** $r = \langle a_1, \ldots, a_n \rangle$ **in** N) =
Bind$[r] \leftarrow \langle a_1, \ldots, a_n \rangle$
if $\text{Count}_{\text{app}}[r] = 0$ and $\text{Count}_{\text{esc}}[r] = 0$
then $\text{Count}_{\text{esc}}[\sigma(a_i)] \leftarrow \text{Count}_{\text{esc}}[\sigma(a_i)] - 1, \qquad i \leq 1 \leq n$
  ncontract(N)
else $N' \leftarrow$ ncontract(N)
  if $\text{Count}_{\text{app}}[r] = 0$ and $\text{Count}_{\text{esc}}[r] = 0$
  then $\text{Count}_{\text{esc}}[a_i] \leftarrow \text{Count}_{\text{esc}}[a_i] - 1, \qquad i \leq 1 \leq n$
    N'
  else **let** $r = \langle \sigma(a_1), \ldots, \sigma(a_n) \rangle$ **in** N'

ncontract (**let** $x = \#i(a)$ **in** N) =
if $\text{Count}_{\text{app}}[x] = 0$ and $\text{Count}_{\text{esc}}[x] = 0$
then $\text{Count}_{\text{app}}[\sigma(a)] \leftarrow \text{Count}_{\text{app}}[\sigma(a)] - 1$; ncontract$(N)$
else if Bind$[\sigma(a)] = \langle b_1, \ldots, b_n \rangle$
then $\sigma \leftarrow \sigma + \{x \mapsto \sigma(b_i)\}$
  $\text{Count}_{\text{app}}[\sigma(b_i)] \leftarrow \text{Count}_{\text{app}}[\sigma(b_i)] + \text{Count}_{\text{app}}[x]$
  $\text{Count}_{\text{esc}}[\sigma(b_i)] \leftarrow \text{Count}_{\text{esc}}[\sigma(b_i)] + \text{Count}_{\text{esc}}[x]$
  $\text{Count}_{\text{app}}[\sigma(a)] \leftarrow \text{Count}_{\text{app}}[\sigma(a)] - 1$
  ncontract(N)
else $N' \leftarrow$ ncontract(N)
  if $\text{Count}_{\text{app}}[x] + \text{Count}_{\text{esc}}[x] = 0$
  then $\text{Count}_{\text{app}}[\sigma(a)] \leftarrow \text{Count}_{\text{app}}[\sigma(a)] - 1$
    N'
  else **let** $x = \#i(\sigma(a))$ **in** N'

Figure 4: Cumulative reductions after each round



distinguish the two cases: if f is a dead function, the usage counts of variables in M must be decremented, while if f is inlined, they should not be decremented.

The way we tell the difference is by recurring on N, and arranging for Bind$[f]$ to be set to a special token, **inlined**, if f is inlined. Upon return, Bind$[f]$ is examined and census$(-1, M)$ called if it is not **inlined**. In either case, M is discarded as in the old *Contract*.

Finally, consider let $f(\vec{x}) = M$ in N where $\text{Count}_{\text{app}}[f] > 1$ or $\text{Count}_{\text{esc}}[f] > 0$. There is no recursive-dead-function or shrinking-inlining redex. But it could be that during ncontract(N), the counts of f decrease because of other reductions. So when ncontract(N) returns, we check for three cases:

- Bind$[f]$ = **inlined**, meaning that during ncontract(N) the counts of f decreased and then $f(\vec{a})$ was found and was replaced by $M\{x_i \mapsto a_i\}$. We must now remove $f(\vec{x}) = M$ without adjusting the counts of variables in M.
- Bind$[f]$ $\neq$ **inlined**, but the counts of f are now zero. We can delete $f(\vec{x}) = M$ and perform census$(-1, M)$.
- Bind$[f]$ $\neq$ **inlined**, and f still has occurrences. We now perform ncontract(M); but any occurrence of $f(\vec{a})$ that we might find within M must not be inlined, because it is a recursive call. To disable inlining of f we set Bind$[f] \leftarrow ()$ before calling ncontract(M).

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

Table 1: Compile-time improvement.

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, and execution time are shown for each benchmark under "old" (multi-pass contract) and "new" (one-pass contract) compilers. The optimizations are *Contract* as well as eta-reduction and speculative inlining [1, Ch. 6 & 7].

Figure 4 shows an example of how *ncontract* finds more redexes in one pass. In a 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. Although we have not reduced all the redexes in just one pass, there are so few remaining that a second pass is not justified by the expected return.

Table 1 shows that using the new *Contract*, all of the benchmark programs (from the benchmark set used by Shao and Appel [9]) 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 completely to shrink-normal form (because the required extra rounds of *Contract* would be too expensive [1, p. 192]); each leaves a (slightly different) set of residuals.

6 Asymptotic complexity of Contract

Both the old and the new *Contract* algorithms reduce expressions to shrink-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 shrink-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 $\Omega(N^2)$ behavior:

$$
\begin{array}{ll}
\textbf{let} & f_1(x_1, y_1, z_1) = h(z_1) \\
& f_2(x_2, y_2, z_2) = h(z_2) \\
& \vdots \\
& f_N(x_N, y_N, z_N) = h(z_N) \\
& g_1() = f_2 \\
& g_2() = f_1(g_1, f_2, f_3) \\
& g_3() = f_2(g_2, f_3, f_4) \\
& \vdots \\
& g_N() = f_{N-1}(g_{N-1}, f_N, x) \\
\textbf{in} & h(g_N)
\end{array}
$$

In the ith pass of the new *Contract*, the body of f_i is inlined in the application $f_i(g_i, f_{i+1}, f_{i+2})$ because the usage count of f_i is 1. On the ith upward pass, function g_i is deleted because it is a dead variable, reducing the usage count of f_{i+1} to 1. Thus N passes are required to reach shrink-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 4. A much more typical case, on which the old *Contract* took N passes and the new *Contract* takes one pass, is:

$$
\begin{array}{ll}
\textbf{let} & r_1 = \langle x, x \rangle \\
& r_2 = \langle r_1, x \rangle \\
& \vdots \\
& r_N = \langle r_{N-1}, x \rangle \\
\textbf{in} & h(x)
\end{array}
$$

7 A linear-time *Contract*

We also have an algorithm that reduces expressions to shrink-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*.

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 readability, in a style of optimization that uses rewriting by structural induction. The new *Contract* described in this paper is easily implemented in such a style; the linear-time algorithm is not. But there are implementation styles in which doubly linked lists are natural, and our algorithm establishes the exact complexity of the problem.

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 nonbinding 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 smallest expression containing the occurrence (recall that an occurrence is not an expression).

rec indicates, for each occurrence c, whether it is a recursive occurrence. In the term $C[\textbf{let } f(x) = N \textbf{ in } M]$, the occurrences of f within N are recursive and the occurrences of f in M are not recursive.

For example, the program fragment

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

is represented by the following expression tree fragment:

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

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

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. We can splice the doubly linked list $\mathbf{occ}(w)$ next to a inside $\mathbf{occ}(q)$, 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 6 shows the algorithm. The algorithm maintains a set of 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 is therefore a dead variable), or that has only a recursive occurrence (and is therefore a recursive dead function); and

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

The initial redex set is obtained by the same *census* function used by all our *Contract* algorithms, modified to mark recursive occurrences as **rec**. Redexes in the set may be removed and reduced in any order, and reduction may add newly discovered redexes to the set.

Figure 5: Auxiliary functions for the linear-time algorithm.

```
delete(a) =
    ; remove a from occurrence list
    occ(var(a)) := occ(var(a)) − {a}
    ; check for new redexes
    if | occ(var(a)) | ≤ 1
        compute v = var(a)
        if | occ(v) | = 0
            add dead(v) to redex set
        if occ(v) = {c},
        and site(c) is c(c⃗_j),
        and def(v) is let v(w⃗_i) = E in F
            if rec(c)
                add dead(v) to redex set
            else add inline(v) to redex set


subst(w, a) =
    ; check for new record redexes
    if rec(a)  ; a is a recursive occurrence
        for each b in occ(w)
            rec(b) := true
    if var(a) defined  ; a is a record
        for each b in occ(w)
            var(b) := var(a)  ; var update
            if site(b) is let x = #i(b) in D
                add select(b) to redex set
    ; perform substitution
    occ(var(a)) := occ(var(a)) ∪ occ(w)
```

Figure 6: The linear-time *Contract* algorithm.

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

Much of the work is done by the two auxiliary functions, **delete** and **subst**. **Delete**(a) removes a from the occurrence list of **var**(a). This can be done in constant time, just as for the update above. Deleting an occurrence can create new dead-variable or function inlining redexes, so **delete** also checks for this. 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 has 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 $O(N)$.

Similarly, we propagate the **rec** property as occurrences are substituted. Consider a term $C[\ \textbf{let}\ v(x) = N\ \textbf{in}\ M\ ]$, where the occurrences of v within N are marked **rec** and the occurrences within M are unmarked. When we perform a reduction within N or M, this may create more occurrences of v; for example, N or M might be

$$C_1[\ \textbf{let}\ r = \langle c_v \rangle\ \textbf{in}\ C_2[\ \textbf{let}\ w = \#1(r)\ \textbf{in}\ K\ ]\ ]$$

where c_v is an occurrence of v, and K contains occurrences e_i of w that now become occurrences of v. If $r = \langle c_v \rangle$ was within N, then c_v would have been marked **rec** and **subst**(w, c_v) will mark all the e_i **rec**; and if $r = \langle c_v \rangle$ was within M, then c_v would not have been **rec** and the e_i will stay non-**rec**. An occurrence acquires the **rec** property at most once, so the total cost of **rec** propagation is linear.

We can now analyze the total running time of the algorithm. It is the sum of the times needed to reduce each redex.

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 $\Theta(N)$.

8 Eta-reduction

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

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

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—provided that $g \notin \{f, x_1, \ldots, x_n\}$. The result is a smaller program, and thus we consider eta-reduction a shrinking reduction. Eta redexes can be introduced by the programmer, but are more commonly introduced by code transformations performed by the compiler.

Contracting an eta redex can create further redexes:

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

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

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 $f(x_1, \ldots, x_n) = M$, we first reduce M, obtaining M'. If M' is of the form $g(x_1, \ldots, x_n)$, we 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:

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

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.

Our original implementation combined the *Eta* and *Contract* phases. But our current implementation keeps the phases separate, for two reasons.

First, we alternate *Contract* with other optimization passes that do speculative inlining and loop-invariant analysis; we iterate this alternation several times. But none of these optimizations introduce new eta redexes; so it suffices to do *Eta* just once, before the other optimizations.

Second, combining *Eta* and *Contract* results in a nonconfluent system. For example, the program

$$\begin{array}{l} \textbf{let } f(x) = M \\ \textbf{in } \ \textbf{let } h(y) = N \\ \quad\quad \textbf{in } \ \textbf{let } g(z) = f(z) \\ \quad\quad\quad\quad \textbf{in } \ h(g) \end{array}$$

rewrites by inlining f to

$$\begin{array}{l} \textbf{let } h(y) = N \\ \textbf{in } \ \textbf{let } g(z) = M\{x \mapsto z\} \\ \quad\quad \textbf{in } \ h(g) \end{array}$$

and by η-reducing g to

$$\begin{array}{l} \textbf{let } f(x) = M \\ \textbf{in } \ \textbf{let } h(y) = N \\ \quad\quad \textbf{in } \ h(f). \end{array}$$

No sequence of reductions can join the two.

The failure of confluence results in a system that is harder to analyze and debug; indeed, our combined *Contract-Eta* was never free of bugs, and was discarded several years ago.

9 Further work

It should be possible to formally relate each of our three algorithms to the rewriting system, and therefore to prove the algorithms correct. This will probably be easier for the linear time algorithm (which performs one reduction at a time) than for the tree-walk algorithms (which perform reductions incrementally).

10 Conclusion

Our improvements to the *Contract* phase of Standard ML of New Jersey yield an algorithm that reduces "almost all" shrink redexes in linear time. Our improved *Eta* phases reduces all eta 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. Furthermore, our rewriting system is confluent (Church-Rosser), so the optimizations are nicely deterministic.

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 *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, 1989.

[3] Andrew W. Appel and Trevor Jim. Making lambda-calculus smaller, faster. Technical Report CS-TR-477-94, Princeton University, November 1994.

[4] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics (Revised Edition)*, volume 103 of *Studies in Logic and the Foundation of Mathematics.* North-Holland, 1984.

[5] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, 1993.

[6] David Kranz. *ORBIT: An optimizing compiler for Scheme.* PhD thesis, Yale University, New Haven, CT, 1987.

[7] Simon L Peyton Jones. Implementing lazy functinal languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2:126–202, 1992.

[8] André Luís de Medeiros Santos. *Compilation by Transformation in Non-Strict Functional Languages.* PhD thesis, University of Glasgow, Glasgow, Scotland, 1995.

[9] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 150–161, 1994.

[10] Guy L. Steele. RABBIT: A compiler for SCHEME. Technical Report AI–TR–474, Artificial Intelligence Laboratory, M.I.T., May 1978.

[11] David Tarditi. *Optimizing ML.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1997. expected 1997.

A Proof of Confluence

We now prove confluence for a class of untyped term rewriting systems that generalizes the shrinking rewrite system of Section 3. Confluence is typically achieved by imposing some syntactic restrictions on the form of rules used to define the system. However, it is difficult to formulate a simple set of restrictions on rules that permit all of the rules we have in mind; and not all rewriting systems are defined by rules. Therefore, we will instead specify properties that the rewriting relation as a whole must satisfy in order to guarantee confluence.

The main results can be stated as follows.

Definition: A rewriting relation is a *shrinking rewriting relation* if it is substitutive, compatible, includes shrinking inlining, dead-function elimination, recursive-dead-function elimination, and satisfies Properties 1–5 below.

Confluence follows from the following stronger result. Let $\to_{\mathrm{r}}$ be the reflexive (but not transitive) closure of $\to$, so $M \to_{\mathrm{r}} M'$ if $M \to M'$ or $M \equiv M'$.

Theorem (Diamond Property): *Suppose $\to$ is a shrinking rewriting relation. If $M_0 \to_{\mathrm{r}} M_1$ and $M_0 \to_{\mathrm{r}} M_2$, there is some M_3 such that $M_1 \to_{\mathrm{r}} M_3$ and $M_2 \to_{\mathrm{r}} M_3$.*

Theorem: The rewriting relation of Section 3 is a shrinking rewriting relation, and therefore, confluent.

We now develop the necessary technical machinery for the proof of the Diamond Property. As we introduce each of the Properties 1–5, we will show that the system of Section 3 satisfies it.

We fix a set of constants, ranged over by δ. Typical δ's include record selection and creation operators, numerals and arithmetic functions, etc. The *CPS terms* are generated by the following grammar.

$$
\begin{array}{lll}
M ::= & \mathbf{let}\ f(x_1, \ldots, x_n) = M\ \mathbf{in}\ N & \textit{recursive function definition} \\
\mid & f(a_1, \ldots, a_n) & \textit{function application} \\
\mid & \mathbf{let}\ x_1, \ldots, x_n = \delta(a_1, \ldots, a_m)\ \mathbf{in}\ M & \textit{primitive operation}
\end{array}
$$

Note that we allow primitive operations to return more than one result. For example, we might want

$$(\mathbf{let}\ \mathit{quot}, \mathit{rem} = 9 \div 2\ \mathbf{in}\ N) \to N\{\mathit{quot} \mapsto 4,\ \mathit{rem} \mapsto 1\}.$$

We use some standard concepts (free and bound variables, occurrences, subterms, etc.) without formal definition; the interested reader may consult Barendregt [4]. We write $\mathrm{fv}(M)$ for the free variables of M, and $M \subset N$ to indicate that M is a subterm of N. We consider terms to be equal modulo renaming of bound variables. We have already mentioned that we require, in any mathematical context, that all bound variables be distinct from each other and from free variables; this is a standard requirement, sometimes called the *Variable Convention.*

We informally introduced the concept of a syntactic context as a "term with a hole." We formalize that idea by the following grammar.

$$\begin{aligned} C[\cdot] ::= & \ [\cdot] \\ & \mid \ \textbf{let}\ f(x_1,\ldots,x_n) = C[\cdot]\ \textbf{in}\ M \\ & \mid \ \textbf{let}\ f(x_1,\ldots,x_n) = M\ \textbf{in}\ C[\cdot] \\ & \mid \ \textbf{let}\ x_1,\ldots,x_n = \delta(a_1,\ldots,a_m)\ \textbf{in}\ C[\cdot] \end{aligned}$$

If $C[\cdot]$ is a context, then $C[M]$ is the term obtained by replacing the hole of $C[\cdot]$ by M, possibly capturing free variables of M; we omit a formal definition. Note that an atom a is *not* a CPS term, so $C[a]$ is not well defined.

In our proof we will need contexts with more than one distinct hole. For example,

$$C[\cdot]_1[\cdot]_2 \equiv \textbf{let}\ f(x_1,\ldots,x_n) = [\cdot]_1\ \textbf{in}\ [\cdot]_2$$

is a context with two distinct holes, which may be filled with two different terms, as in

$$C[M]_1[N]_2 \equiv \textbf{let}\ f(x_1,\ldots,x_n) = M\ \textbf{in}\ N.$$

We sometimes abbreviate $C[\cdot]_1[\cdot]_2$ by $C[\cdot][\cdot]$. See Barendregt [4, §14.4] for a formal definition of this sort of context.

A *(term) rewriting relation* is a binary relation on terms. A rewriting relation R is *compatible* if whenever $(M,N) \in R$, then $(C[M],C[N]) \in R$ for every context $C[\cdot]$. The *compatible closure* of a rewriting relation R is the least compatible relation containing R. The *kernel* of a compatible rewriting relation R is the least relation whose compatible closure is R. If (Δ,Δ') is in the kernel of R then Δ is called a *redex* and Δ' a *contractum (of* Δ*).* If $\rightarrow$ is a compatible rewriting relation, we write $M \stackrel{\Delta}{\rightarrow} M'$ to indicate that M rewrites to M' by contracting redex Δ, that is, we have $M \equiv C[\Delta]$ and $M' \equiv C[\Delta']$ for some context $C[\cdot]$ and contractum Δ' of Δ.

The *domain* of a substitution $\{\vec{x} \mapsto \vec{a}\}$ consists of the variables $\vec{x}$, and its *range* consists of the atoms $\vec{a}$. A substitution σ may be applied to: an atom a yielding an atom $a\sigma$; a sequence of atoms $\vec{a}$ yielding a sequence of atoms $\vec{a}\sigma$; a term M yielding a term $M\sigma$; or a context $C[\cdot]$ yielding a context $(C\sigma)[\cdot]$. A rewriting relation R is *substitutive* if whenever $(\Delta,\Delta') \in R$, then $(\Delta\sigma,\Delta'\sigma) \in R$ for any substitution σ.

Two standard results about substitutions will be useful.

Lemma 1 *If no variable in the domain or range of σ is bound in $C[\cdot]$, then*

$$(C[M])\sigma \equiv (C\sigma)[M\sigma]$$

for any term M.

Lemma 2 *If no variable of $\vec{x}$ appears in the domain or range of σ, then*

$$(M\{\vec{x} \mapsto \vec{a}\})\sigma \equiv (M\sigma)\{\vec{x} \mapsto \vec{a}\sigma\}$$

for any term M and atoms $\vec{a}$.

We now develop the Properties needed for confluence. Our first Property says that every reduction deletes a definition, and that reduction is invariant with respect to certain changes in the syntax of terms. In stating the Property, we use D to range over definitions ($f(\vec{x}) = M$ or $\vec{x} = \delta(\vec{a})$), and we say $f(\vec{x}) = M$ *defines the variables* $\{f\}$, and $\vec{x} = \delta(\vec{a})$ *defines the variables* $\{\vec{x}\}$.

Property 1 *A rewriting relation* $\rightarrow$ satisfies Property 1 *if, whenever $M \xrightarrow{\Delta} M'$ and Δ is not a shrinking inlining redex, there exist a substitution σ, a unique context $C[\cdot]$, and a unique term* ($\mathbf{let}\ D\ \mathbf{in}\ N$) *such that*

$$M \equiv C[\ \mathbf{let}\ D\ \mathbf{in}\ N\ ] \xrightarrow{\Delta} C[\ N\sigma\ ] \equiv M',$$

the domain of σ contains only variables defined by D, and $C[\ \mathbf{let}\ D\ \mathbf{in}\ N'\ ] \rightarrow C[\ N'\sigma\ ]$ for any term N' that contains no more variables defined by D than N.

The term ($\mathbf{let}\ D\ \mathbf{in}\ N$) in Property 1 will be called the *focus* of the redex Δ, and $N\sigma$ will be called the *focal replacement* of Δ. For example, in the system of Section 3, we have the following focuses and focal replacements.

- If Δ is a dead-variable-elimination redex ($\mathbf{let}\ D\ \mathbf{in}\ N$), the focus of Δ is Δ and the focal replacement of Δ is N. Note, here we may take the substitution σ to be the empty (identity) substitution.

- If Δ is a record selection redex ($\mathbf{let}\ r = \langle\vec{a}\rangle\ \mathbf{in}\ C[\ \mathbf{let}\ x = \#i(r)\ \mathbf{in}\ N\ ]$), the focus of Δ is the subterm ($\mathbf{let}\ x = \#i(r)\ \mathbf{in}\ N$) and the focal replacement of Δ is $N\{x \mapsto a_i\}$.

It will be useful to extend this terminology to shrinking inlining redexes.

- If Δ is a shrinking inlining redex ($\mathbf{let}\ f(\vec{x}) = N\ \mathbf{in}\ C[\ f(\vec{a})\ ]$), the focus of Δ is Δ, and the focal replacement of Δ is $C[\ N\{\vec{x} \mapsto \vec{a}\}\ ]$.

Intuitively, the first part of Property 1 says that every rewrite rule of the system deletes a definition, and the focus of a redex is defined to be the smallest subterm containing the deleted definition. When a redex is contracted, only the focus is affected; the portion of the term outside of the focus is unchanged. Usually the focus of the redex is the redex itself, and the focal replacement is the contractum of the redex; but not always, as in the case of record selection.

To verify the second part of Property 1 for the system of Section 3, we consider two cases.

- If $C[\ \textbf{let}\ D\ \textbf{in}\ N\ ] \to C[\ N\ ]$ by dead-variable elimination, then N contains no variables defined by D. And $C[\ \textbf{let}\ D\ \textbf{in}\ N'\ ] \to C[\ N'\ ]$ for any N' containing no variables defined by D.
- If $C[\ \textbf{let}\ x = \#i(r)\ \textbf{in}\ N\ ] \to C[\ N\{x \mapsto a_i\}\ ]$ by record selection, then $C[\ \textbf{let}\ x = \#i(r)\ \textbf{in}\ N'\ ] \to C[\ N'\{x \mapsto a_i\}\ ]$ for any N', regardless of the variables it contains.

The next Property concerns the reduction of a redex properly containing its focus, e.g., record selection. Like Property 1, it states that such reductions are invariant under certain syntactic modifications to terms.

In a term ($\textbf{let}\ D\ \textbf{in}\ N$), we call the definition D the *head* and N the *body*. We say a definition is *dominant* in a context if its scope includes the hole of the context. Note that if Δ is a redex with focus $F \not\equiv \Delta$ in term $C[F]$, then the head of Δ is dominant in $C[\cdot]$.

Property 2 *A rewriting relation* $\to$ satisfies Property 2 *if, whenever* Δ *is a redex with focus* $F \not\equiv \Delta$ *and focal replacement* F'*, then*

$$C_1[\ (C_2[\ F\ ])\sigma\ ] \to C_1[\ (C_2[\ F'\ ])\sigma\ ]$$

for any $C_1[\cdot]$*,* $C_2[\cdot]$*, and* σ *such that the head of* Δ *is dominant in* $C_1[\cdot]$*, and the domain of* σ *includes no variable appearing in* $C_1[\cdot]$*.*

For the system of Section 3, the only case in which $F \not\equiv \Delta$ is when Δ is a record selection redex. In this case, $C_1[\cdot]$, F, and F' have the following forms:

$$\begin{array}{rcl} C_1[\cdot] & \equiv & C[\ \textbf{let}\ r = \langle\vec{a}\rangle\ \textbf{in}\ C'[\cdot]\ ], \\ F & \equiv & \textbf{let}\ x = \#i(r)\ \textbf{in}\ N, \\ F' & \equiv & N\{x \mapsto a_i\}, \end{array}$$

and we want to verify that

$$C_1[\ (C_2[\ \textbf{let}\ x = \#i(r)\ \textbf{in}\ N\ ])\sigma\ ] \to C_1[\ (C_2[\ N\{x \mapsto a_i\}\ ])\sigma\ ].$$

We assume that x does not appear in σ (else rename x). Then since r is not in the domain of σ, we have $(\textbf{let}\ x = \#i(r)\ \textbf{in}\ N)\sigma \equiv (\textbf{let}\ x = \#i(r)\ \textbf{in}\ (N\sigma))$. And since a_i is not in the domain of σ, by Lemma 2 we have $(N\{x \mapsto a_i\})\sigma \equiv (N\sigma)\{x \mapsto a_i\}$. This is enough to verify the reduction.

The following lemma summarizes an important special case of Property 2.

Lemma 3 *Suppose* $\to$ *satisfies Property 2, and* Δ *is a redex with focus* $F \not\equiv \Delta$ *and focal replacement* F'*. Then* $C[F] \to C[F']$ *for any context* $C[\cdot]$ *in which the head of* Δ *is dominant.*

Because of the shrinking inlining and dead-function rules, we must keep track of the number of *occurrences in function position* of a variable during the course of a reduction (e.g., in the term $f(a)$, f is in function position and a is not). The next property gives conditions guaranteeing that occurrences in function position *decrease*. It holds for our shrinking rewrite system, but not for rewrite systems in general; for example, it fails under the unrestricted inlining rule.

Property 3 *A rewriting relation* $\rightarrow$ satisfies Property 3 *if, whenever a variable f has n occurrences in function position in a term M, and no other occurrences, and $M \rightarrow M'$, then f has n or less occurrences in function position in M', and no other occurrences.*

Our final two properties concern overlaps, which can be particularly troublesome in proving confluence. The first property states that when overlaps occur, they do so in a harmless manner. The second states that a harmful kind of overlap does not occur. In practice, these two properties are the most difficult to prove of a rewrite system, because the number of cases to consider is quadratic in the number of rules.

Property 4 *A rewriting relation* $\rightarrow$ satisfies Property 4 *if, whenever two redexes have the same focus, then they have the same focal replacement. So, if Δ_1 and Δ_2 have the same focus, $M \overset{\Delta_1}{\rightarrow} M_1$, and $M \overset{\Delta_2}{\rightarrow} M_2$, then $M_1 \equiv M_2$.*

For the system of Section 3, a case analysis shows that if two distinct redexes have the same focus, then one redex is a record selection redex

$$\Delta_1 \equiv \textbf{let}\ r = \langle \vec{a} \rangle\ \textbf{in}\ C[\ \textbf{let}\ x = \#i(r)\ \textbf{in}\ N\ ],$$

and the other redex is a dead-variable-elimination redex

$$\Delta_2 \equiv (\textbf{let}\ x = \#i(r)\ \textbf{in}\ N).$$

That is, Δ_2 is the focus of Δ_1. Since x does not appear in N, the focal replacement, $N\{x \mapsto a_i\}$, of Δ_1 is the same as the focal replacement, N, of Δ_2, as desired.

Property 5 *A rewriting relation* $\rightarrow$ satisfies Property 5 *if, whenever $F = (\textbf{let}\ D\ \textbf{in}\ N)$ is the focus of a redex in a term M, F' is the focus of a second redex, Δ', in M, and F' is a proper subterm of F, then D is not the head of Δ'.*

If $F' \equiv \Delta'$, then Δ' is a proper subterm of F, so D is not the head of Δ'. Thus to verify Property 5, it is sufficient to consider those cases where $F' \not\equiv \Delta'$. For the system of Section 3, the only such case is when Δ' is a record selection redex. By way of contradiction, assume D is the head $r = \langle \vec{a} \rangle$ of Δ'. Then the focus (**let** D **in** N) can only be a dead-variable redex. But the record selection focus $F' \subset N$ must contain r, contradiction.

Proof of the Diamond Property:

If $M_0 \equiv M_1$ or $M_0 \equiv M_2$ the result follows trivially. So assume $M_0 \overset{\Delta_1}{\to} M_1$ and $M_0 \overset{\Delta_2}{\to} M_2$ for some redexes Δ_1 and Δ_2, with focuses F_1 and F_2, respectively. We consider the following cases.

If F_1 and F_2 are disjoint, then M_0, M_1, and M_2 have the following forms:

$$\begin{array}{rcl} M_0 & \equiv & C[\ F_1\ ][\ F_2\ ], \\ M_1 & \equiv & C[\ F_1'\ ][\ F_2\ ], \\ M_2 & \equiv & C[\ F_1\ ][\ F_2'\ ]. \end{array}$$

Here F_1' and F_2' are the focal replacements of F_1 and F_2, respectively.

Define $M_3 \equiv C[\ F_1'\ ][\ F_2'\ ]$. If $F_2 \equiv \Delta_2$, then $M_1 \overset{\Delta_2}{\to} M_3$ by compatibility. If $F_2 \not\equiv \Delta_2$, then the head of Δ_2 is dominant in $C[F_1][\cdot]$ and therefore also in $C[F_1'][\cdot]$. Then $M_1 \to M_3$ by Lemma 3. The same argument shows that $M_2 \to M_3$.

If F_1 and F_2 coincide, then by Property 4, $M_1 \equiv M_2$, so it is sufficient to choose $M_3 \equiv M_1$.

Otherwise the focus of one redex is properly contained in the focus of another; we assume without loss of generality that F_1 contains F_2. Let F_2' be the focal replacement of F_2. We consider the following cases.

- If F_1 is not a shrinking inlining redex, and F_2 is contained in the body of F_1, then M_0, M_1, and M_2 have the following forms:

$$\begin{array}{rcl} M_0 & \equiv & C_1[\ \mathbf{let}\ D\ \mathbf{in}\ C_2[\ F_2\ ]\ ], \\ M_1 & \equiv & C_1[\ (C_2[\ F_2\ ])\sigma\ ], \\ M_2 & \equiv & C_1[\ \mathbf{let}\ D\ \mathbf{in}\ C_2[\ F_2'\ ]\ ]. \end{array}$$

  Here D is the head of F_1 and σ is the substitution predicted by Property 1.

  Define $M_3 \equiv C_1[\ (C_2[\ F_2'\ ])\sigma\ ]$.

  If $\Delta_2 \subset C_2[F_2]$, then $M_1 \overset{\Delta_2\sigma}{\to} M_3$ by compatibility and substitutivity.

  Otherwise $F_2 \not\equiv \Delta_2$, and the head of Δ_2 is dominant in $C_1[\cdot]$ (the head of Δ_2 is not D by Property 5). Then $M_1 \to M_3$ by Property 2.

  Let y be a variable defined by D. Note that if y does not appear in $C_2[\ F_2\ ]$, it appears nowhere in M_0. Then by Property 3, y does not appear in $C_2[\ F_2'\ ]$, so $M_2 \to M_3$ by Property 1.

- If F_1 is a dead-function-elimination redex and F_2 is contained in the head of F_1, then M_0, M_1, and M_2 have the following forms:

$$M_0 \quad \equiv \quad C_1[\ \mathbf{let}\ f(\vec{x}) = C_2[\ F_2\ ]\ \mathbf{in}\ M\ ],$$

$$
\begin{aligned}
M_1 &\equiv C_1[\ M\ ], \\
M_2 &\equiv C_1[\ \textbf{let}\ f(\vec{x}) = C_2[\ F_2'\ ]\ \textbf{in}\ M\ ].
\end{aligned}
$$

Define $M_3 \equiv C_1[\ M\ ]$; then $M_1 \rightarrow_{\mathrm{r}} M_3$ by reflexivity.

Since f is a dead function, it has at most one occurrence in function position in $C_2[\ F_2\ ]$, and no other occurrences anywhere in M_0. Then by Property 3, f has at most one occurrence in function position in $C_2[\ F_2'\ ]$, and no other occurrences in M_2. So $M_2 \rightarrow M_3$ by eliminating the dead function f.

- If F_1 is a shrinking inlining redex and F_2 is contained in the head of F_1, then M_0, M_1, and M_2 have the following forms:

$$
\begin{aligned}
M_0 &\equiv C_1[\ \textbf{let}\ f(\vec{x}) = C_2[\ F_2\ ]\ \textbf{in}\ C_3[\ f(\vec{a})\ ]\ ], \\
M_1 &\equiv C_1[\ C_3[\ (C_2[\ F_2\ ])\{\vec{x} \mapsto \vec{a}\}\ ]\ ], \\
M_2 &\equiv C_1[\ \textbf{let}\ f(\vec{x}) = C_2[\ F_2'\ ]\ \textbf{in}\ C_3[\ f(\vec{a})\ ]\ ].
\end{aligned}
$$

Define $M_3 \equiv C_1[\ C_3[\ (C_2[\ F_2'\ ])\{\vec{x} \mapsto \vec{a}\}\ ]\ ]$.

Since f has a single occurrence in function position in M_0, and no other occurrences in M_0, by Property 3 the same is true of f in M_2. So $M_2 \rightarrow M_3$ by shrinking inlining.

If $\Delta_2 \subset C_2[F_2]$, then $M_1 \overset{\Delta_2\{\vec{x}\mapsto\vec{a}\}}{\rightarrow} M_3$ by compatibility and substitutivity. Otherwise $F_2 \not\equiv \Delta_2$, and the head of Δ_2 is dominant in $C_1[\cdot]$ (the head of Δ_2 is not $f(\vec{x}) = C_2[\ F_2\ ]$ by Property 5). Therefore, the head of Δ_2 is also dominant in $C_1[\ C_3[\cdot]\ ]$, and $M_1 \rightarrow M_3$ by Property 2.

- If F_1 is a shrinking inlining redex and F_2 is contained in the body of F_1 disjoint from the inlining site, then M_0, M_1, and M_2 have the following forms:

$$
\begin{aligned}
M_0 &\equiv C_1[\ \textbf{let}\ f(\vec{x}) = M\ \textbf{in}\ C_2[\ F_2\ ][\ f(\vec{a})\ ]\ ], \\
M_1 &\equiv C_1[\ C_2[\ F_2\ ][\ M\{\vec{x} \mapsto \vec{a}\}\ ]\ ], \\
M_2 &\equiv C_1[\ \textbf{let}\ f(\vec{x}) = M\ \textbf{in}\ C_2[\ F_2'\ ][\ f(\vec{a})\ ]\ ].
\end{aligned}
$$

Define $M_3 \equiv C_1[\ C_2[\ F_2'\ ][\ M\{\vec{x} \mapsto \vec{a}\}\ ]\ ]$.

Since f has a single occurrence in function position in M_0, and no other occurrences in M_0, by Property 3 the same is true of f in M_2. So $M_2 \rightarrow M_3$ by shrinking inlining.

If $F_2 \equiv \Delta_2$, then $M_1 \overset{\Delta_2}{\rightarrow} M_3$ by compatibility.

If $F_2 \not\equiv \Delta_2$, then the head of Δ_2 must be dominant in $C_1[\ \textbf{let}\ f(\vec{x}) = M\ \textbf{in}\ C_2[\cdot][\ f(\vec{a})\ ]\ ]$, and cannot be $f(x) = M$ by Property 5. Therefore the head of Δ_2 is dominant in $C_1[\ C_2[\cdot][\ M\{\vec{x} \mapsto \vec{a}\}\ ]\ ]$. So by Lemma 3, $M_1 \rightarrow M_3$.

- If F_1 and F_2 are shrinking inlining redexes, and the inlining site of F_1 appears in the head of F_2, then M_0, M_1, and M_2 have the following forms:

$$\begin{array}{rcl} M_0 & \equiv & C_1[\,\mathbf{let}\ f(\vec{x}) = M\ \mathbf{in}\ C_2[\,\mathbf{let}\ g(\vec{y}) = C_3[\,f(\vec{a})\,]\ \mathbf{in}\ C_4[\,g(\vec{b})\,]\,]\,], \\ M_1 & \equiv & C_1[\,C_2[\,\mathbf{let}\ g(\vec{y}) = C_3[\,M\{\vec{x} \mapsto \vec{a}\}\,]\ \mathbf{in}\ C_4[\,g(\vec{b})\,]\,]\,], \\ M_2 & \equiv & C_1[\,\mathbf{let}\ f(\vec{x}) = M\ \mathbf{in}\ C_2[\,C_4[\,(C_3[\,f(\vec{a})\,])\{\vec{y} \mapsto \vec{b}\}\,]\,]\,]. \end{array}$$

  Define $M_3 \equiv C_1[\,C_2[\,C_4[\,(C_3[\,M\{\vec{x} \mapsto \vec{a}\}\,])\{\vec{y} \mapsto \vec{b}\}\,]\,]\,]$.

  Then $M_1 \to M_3$ by shrinking inlining.

  Note that $\vec{y}$ and $\vec{b}$ are not bound in $C_3[\cdot]$, and are disjoint from $\vec{x}$. Then by Lemmas 1 and 2,

$$\begin{array}{rcl} (C_3[\,f(\vec{a})\,])\{\vec{y} \mapsto \vec{b}\} & \equiv & (C_3\{\vec{y} \mapsto \vec{b}\})[\,f(\vec{a}\{\vec{y} \mapsto \vec{b}\})\,], \\ (C_3[\,M\{\vec{x} \mapsto \vec{a}\}\,])\{\vec{y} \mapsto \vec{b}\} & \equiv & (C_3\{\vec{y} \mapsto \vec{b}\})[\,M\{\vec{x} \mapsto \vec{a}\{\vec{y} \mapsto \vec{b}\}\}\,]. \end{array}$$

  So $M_2 \to M_3$ by shrinking inlining.

- If F_1 and F_2 are shrinking inlining redexes, and the inlining site of F_1 appears in the body of F_2, then M_0, M_1, and M_2 have the following forms:

$$\begin{array}{rcl} M_0 & \equiv & C_1[\,\mathbf{let}\ f(\vec{x}) = M\ \mathbf{in}\ C_2[\,\mathbf{let}\ g(\vec{y}) = N\ \mathbf{in}\ C_3[\,f(\vec{a})\,][\,g(\vec{b})\,]\,]\,], \\ M_1 & \equiv & C_1[\,C_2[\,\mathbf{let}\ g(\vec{y}) = N\ \mathbf{in}\ C_3[\,M\{\vec{x} \mapsto \vec{a}\}\,][\,g(\vec{b})\,]\,]\,], \\ M_2 & \equiv & C_1[\,\mathbf{let}\ f(\vec{x}) = M\ \mathbf{in}\ C_2[\,C_3[\,f(\vec{a})\,][\,N\{\vec{y} \mapsto \vec{b}\}\,]\,]\,]. \end{array}$$

  Define $M_3 \equiv C_1[\,C_2[\,C_3[\,M\{\vec{x} \mapsto \vec{a}\}\,][\,N\{\vec{y} \mapsto \vec{b}\}\,]\,]\,]$. Note that f does not appear in $N\{\vec{y} \mapsto \vec{b}\}$, and g does not appear in $M\{\vec{x} \mapsto \vec{a}\}$. Then $M_1 \to M_3$ and $M_2 \to M_3$ by shrinking inlining.

- If F_1 is a shrinking inlining redex, F_2 is a dead-function redex, and the head of F_2 contains the inlining site of F_1, then M_0, M_1, and M_2 have the following forms:

$$\begin{array}{rcl} M_0 & \equiv & C_1[\,\mathbf{let}\ f(\vec{x}) = M\ \mathbf{in}\ C_2[\,\mathbf{let}\ g(\vec{y}) = C_3[\,f(\vec{a})\,]\ \mathbf{in}\ N\,]\,], \\ M_1 & \equiv & C_1[\,C_2[\,\mathbf{let}\ g(\vec{y}) = C_3[\,M\{\vec{x} \mapsto \vec{a}\}\,]\ \mathbf{in}\ N\,]\,], \\ M_2 & \equiv & C_1[\,\mathbf{let}\ f(\vec{x}) = M\ \mathbf{in}\ C_2[\,N\,]\,]. \end{array}$$

  Define $M_3 \equiv C_1[\,C_2[\,N\,]\,]$. Then $M_2 \to M_3$ by eliminating the dead function f.

  Since g is a dead function, it has at most one occurrence in function position in $C_3[\,f(\vec{a})\,]$, and no other occurrences anywhere in M_0. Then by Property 3, g has at most one occurrence in function position in $C_3[\,M\{\vec{x} \mapsto \vec{a}\}\,]$, and no other occurrences anywhere in M_1. So $M_1 \to M_3$ by eliminating the dead function g.

- If F_1 is a shrinking inlining redex, F_2 is not a shrinking inlining redex, and the inlining site of F_1 is in the body of F_2, then M_0, M_1, and M_2 have the following forms:

$$\begin{array}{lcl} M_0 & \equiv & C_1[\ \mathbf{let}\ f(\vec{x}) = M\ \mathbf{in}\ C_2[\ \mathbf{let}\ D\ \mathbf{in}\ C_3[\ f(\vec{a})\ ]\ ]\ ], \\ M_1 & \equiv & C_1[\ C_2[\ \mathbf{let}\ D\ \mathbf{in}\ C_3[\ M\{\vec{x} \mapsto \vec{a}\}\ ]\ ]\ ], \\ M_2 & \equiv & C_1[\ \mathbf{let}\ f(\vec{x}) = M\ \mathbf{in}\ C_2[\ (C_3[\ f(\vec{a})\ ])\sigma\ ]\ ]. \end{array}$$

  Here σ is the substitution predicted by Property 1.

  Define $M_3 \equiv C_1[\ C_2[\ (C_3[\ M\{\vec{x} \mapsto \vec{a}\}\ ])\sigma\ ]\ ]$.

  Because f is not in the domain of σ, and no variable of M is in the domain of σ, we have

$$\begin{array}{rcl} (C_3[\ f(\vec{a})\ ])\sigma & \equiv & (C_3\sigma)[\ f(\vec{a}\sigma)\ ], \\ (C_3[\ M\{\vec{x} \mapsto \vec{a}\}\ ])\sigma & \equiv & (C_3\sigma)[\ M\{\vec{x} \mapsto \vec{a}\sigma\}\ ]. \end{array}$$

  Then $M_2 \rightarrow M_3$ by shrinking inlining.

  Let y be a variable defined by D. Note that if y does not appear in $C_3[\ f(\vec{a})\ ]$, then it appears nowhere in M_0. Then by Property 3, y does not appear in $C_3[\ M\{\vec{x} \mapsto \vec{a}\}\ ]$, so $M_1 \rightarrow M_3$ by Property 1.

End proof.