Shrinking Reductions in SML.NET

Nick Benton[1], Andrew Kennedy[1], Sam Lindley[2], and Claudio Russo[1]

[1] Microsoft Research, Cambridge {nick,akenn,crusso}@microsoft.com
[2] LFCS, University of Edinburgh Sam.Lindley@ed.ac.uk

Abstract. One performance-critical phase in the SML.NET compiler involves rewriting intermediate terms to monadic normal form and performing non-duplicating β-reductions. We present an imperative algorithm for this simplification phase, working with a mutable, pointer-based term representation, which significantly outperforms our existing functional algorithm.

1 Introduction

SML.NET [3,4] is a compiler for Standard ML that targets the .NET Common Language Runtime [7]. Like most other compilers for functional languages (e.g. GHC [10]), SML.NET is structured as the composition of a number of transformation phases on an intermediate representation of the user program. As SML.NET is a whole program compiler, the intermediate terms are typically rather large and good performance of the transformations is critical for usability.

Like MLj [5], SML.NET uses a monadic intermediate language (MIL) [2] that is similar to Moggi's computational metalanguage. Most of the phases in SML.NET perform specific transformations, such as closure conversion, arity raising or monomorphisation, and are run only once. In between several of these phases, however, is a general-purpose 'clean-up' pass called *simplify*. Running *simplify* puts the term into *monadic normal form* [6,8], which we have previously called *cc-normal form* and is essentially the same as *A normal form* or *administrative normal form* for CPS [8]. The *simplify* pass also performs *shrinking reductions*: β-reductions for functions, computations, products that always reduce the size of the term.

Appel and Jim [1] describe three algorithms for shrinking reductions. The first 'naïve' and second 'improved' algorithms both have quadratic worst-case time complexity, and the third 'imperative' algorithm is linear, but requires a mutable representation of terms. Appel and Jim did not implement the third algorithm, which does not integrate easily in a mainly-functional compiler. Both SML/NJ and SML.NET use the 'improved' algorithm, which is reasonably efficient in practice. Nevertheless, SML.NET spends a significant amount of time performing shrinking reductions. We have now implemented a variant of the imperative algorithm in SML.NET, and achieved significant speedups.

This paper makes several contributions. It gives the first implementation and benchmarks of the imperative algorithm in a real compiler. It extends the imperative algorithm to a richer language than considered by Appel and Jim. It introduces a 'one-pass' traversal strategy, giving a weak form of compositionality. An extended version of this work appears in the third author's PhD thesis [9].

2 Simplified MIL

For purposes of exposition we present a simplified version of MIL:

Atoms	$a, b ::= x \mid * \mid c$
Values	$v, w ::= a \mid \mathsf{pair}(a,b) \mid \mathsf{proj}_1(a) \mid \mathsf{proj}_2(a) \mid \mathsf{inj}_1(a) \mid \mathsf{inj}_2(a)$
Computations	$m, n, p ::= \mathsf{app}(a,b) \mid \mathsf{letfun}\ f(x)\ \mathsf{be}\ m\ \mathsf{in}\ n$
	$\mid \mathsf{val}(v) \mid \mathsf{let}\ x\ \mathsf{be}\ m\ \mathsf{in}\ n \mid \mathsf{case}\ a\ \mathsf{of}\ (x_1)n_1\ ;\ (x_2)n_2$

where variables are ranged over by f, g, x, y, z, and constants are ranged over by c. Note that the letfun construct binds a possibly recursive function. We define the *height* of MIL terms $|\cdot|$ as:

$$\begin{array}{rcl} |a| = 1 & & |\mathsf{letfun}\ f(x)\ \mathsf{be}\ m\ \mathsf{in}\ n| = |m| + |n| + 1 \\ |\mathsf{proj}_i(a)| = |\mathsf{inj}_i(a)| = 2 & & |\mathsf{let}\ x\ \mathsf{be}\ m\ \mathsf{in}\ n| = |m| + |n| + 1 \\ |\mathsf{app}(a,b)| = |\mathsf{pair}(a,b)| = 3 & & |\mathsf{val}(v)| = |v| + 1 \end{array}$$

$$|\mathsf{case}\ a\ \mathsf{of}\ (x_1)n_1\ ;\ (x_2)n_2| = max(|n_1|, |n_2|) + 2$$

We say that a reduction is a shrinking reduction if it always reduces the height of terms. The most important reductions are given by the shrinking β-rules:

$$\begin{array}{lll} (\rightarrow.\beta_0) & \mathsf{letfun}\ f(x)\ \mathsf{be}\ n\ \mathsf{in}\ m \longrightarrow m, & f \notin fv(m) \\ (\rightarrow.\beta_1) & \mathsf{letfun}\ f(x)\ \mathsf{be}\ m\ \mathsf{in}\ C[\mathsf{app}(f,a)] \longrightarrow C[m[x := a]], & f \notin fv(C[\cdot], m, a) \\ (T.\beta_0) & \mathsf{let}\ x\ \mathsf{be}\ \mathsf{val}(v)\ \mathsf{in}\ m \longrightarrow m, & x \notin fv(m) \\ (T.\beta_a) & \mathsf{let}\ x\ \mathsf{be}\ \mathsf{val}(a)\ \mathsf{in}\ m \longrightarrow m[x := a] & \\ (\times.\beta) & \mathsf{let}\ y\ \mathsf{be}\ \mathsf{val}(\mathsf{pair}(a_1,a_2))\ \mathsf{in}\ C[\mathsf{proj}_i(y)] & \\ & \quad \longrightarrow \mathsf{let}\ y\ \mathsf{be}\ \mathsf{val}(\mathsf{pair}(a_1,a_2))\ \mathsf{in}\ C[a_i] & \\ (+.\beta) & \mathsf{let}\ y\ \mathsf{be}\ \mathsf{val}(\mathsf{inj}_i(a)) & \\ & \mathsf{in}\ C[\mathsf{case}\ y\ \mathsf{of}\ (x_1)n_1\ ;\ (x_2)n_2] & \\ & \quad \longrightarrow \mathsf{let}\ y\ \mathsf{be}\ \mathsf{val}(\mathsf{inj}_i(a))\ \mathsf{in}\ C[n_i[x_i := a]] & \end{array}$$

We write R_β for the one-step reduction relation defined by the β-rules. The *simplify* transformation also performs commuting conversions. These ensure that bindings are explicitly sequenced, which enables further rewriting.

$$\begin{array}{ll} (T.CC) & \mathsf{let}\ y\ \mathsf{be}\ (\mathsf{let}\ x\ \mathsf{be}\ m\ \mathsf{in}\ n)\ \mathsf{in}\ p \\ & \quad \longrightarrow \mathsf{let}\ x\ \mathsf{be}\ m\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ n\ \mathsf{in}\ p \\ (\rightarrow.CC) & \mathsf{let}\ y\ \mathsf{be}\ (\mathsf{letfun}\ f(x)\ \mathsf{be}\ m\ \mathsf{in}\ n)\ \mathsf{in}\ p \\ & \quad \longrightarrow \mathsf{letfun}\ f(x)\ \mathsf{be}\ m\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ n\ \mathsf{in}\ p \\ (+.CC) & \mathsf{let}\ y\ \mathsf{be}\ (\mathsf{case}\ a\ \mathsf{of}\ (x_1)n_1\ ;\ (x_2)n_1)\ \mathsf{in}\ m \\ & \quad \longrightarrow \mathsf{letfun}\ f(y)\ \mathsf{be}\ m\ \mathsf{in}\ \mathsf{case}\ a\ \mathsf{of}\ (x_1)\mathsf{let}\ y_1\ \mathsf{be}\ n_1\ \mathsf{in}\ \mathsf{app}(f, y_1) \\ & \qquad\qquad ;\ (x_2)\mathsf{let}\ y_2\ \mathsf{be}\ n_2\ \mathsf{in}\ \mathsf{app}(f, y_2) \end{array}$$

We write R_{CC} for the one-step reduction relation defined by the CC-rules, and R for $R_\beta \cup R_{CC}$. Unlike the β rules, the commuting conversions are not actually shrinking reductions. However, $T.CC$ and $\rightarrow.CC$ do not change the size, whilst $+.CC$ gives only a constant increase in the size.

An alternative to the $+.CC$ rule is:

$$(+.CC') \quad \mathsf{let}\ y\ \mathsf{be\ case}\ a\ \mathsf{of}\ (x_1)n_1\ ;(x_2)n_2\ \mathsf{in}\ m$$
$$\longrightarrow\ \mathsf{case}\ a\ \mathsf{of}\ (x_1)\mathsf{let}\ y_1\ \mathsf{be}\ n_1\ \mathsf{in}\ m_1\ ;(x_2)\mathsf{let}\ y_2\ \mathsf{be}\ n_2\ \mathsf{in}\ m_2$$

where y_1, y_2 are fresh, $m_i = m[y := y_i]$. This rule duplicates the term m and can exponentially increase the term's size. The $+.CC$ rule instead creates a single new abstraction, shared across both branches of the case, though this inhibits some further rewriting. We write R'_{CC} for the one-step relation defined by the CC-rules where $(+.CC)$ is replaced by $(+.CC')$, and R' for $R_\beta \cup R'_{CC}$.

Proposition 1. *R' is strongly-normalising.*

Proof. First, note that R_β is strongly-normalising as R_β-reduction strictly decreases the size of terms. We define a measure $||\cdot||$ on terms:

$$||a|| = 1 \qquad ||\mathsf{letfun}\ f(x)\ \mathsf{be}\ m\ \mathsf{in}\ n|| = ||m|| + ||n|| + 1$$
$$||\mathsf{proj}_i(a)|| = ||\mathsf{inj}_i(a)|| = 2 \qquad ||\mathsf{let}\ x\ \mathsf{be}\ m\ \mathsf{in}\ n|| = ||m||^2 + ||n|| + 1$$
$$||\mathsf{app}(a,b)|| = ||\mathsf{pair}(a,b)|| = 3 \qquad ||\mathsf{val}(v)|| = ||v|| + 1$$
$$||\mathsf{case}\ a\ \mathsf{of}\ (x_1)n_1\ ;(x_2)n_2|| = max(||n_1||, ||n_2||) + 2$$

The lexicographic ordering $(|\cdot|, ||\cdot||)$ is a measure for R'-reduction. □

Proposition 2. *R is strongly-normalising.*

The proof uses R'-reduction to simulate R-reduction. The full details are omitted, but the idea is that for any R-reduction a corresponding non-empty sequence of R'-reductions can be performed. Thus, given that all R'-reduction sequences are finite, all R-reduction sequences must also be finite. The proof is slightly complicated by the fact that no non-empty sequence of R'-reductions corresponds with the β-reduction of a function introduced by the $+.CC$ rule. A simple way of dealing with this is to count a $+.CC'$-reduction as two reductions.

Note that R-reductions are not confluent. The failure of confluence is due to the $(+.CC)$ rule. Replacing $(+.CC)$ with $(+.CC')$ does give a confluent system. Confluence can make reasoning about reductions easier, but we do not regard failure of confluence as a problem. In our case, preventing exponential growth in the size of terms is far more important.

3 Previous Work

Appel and Jim [1] considered a calculus which is essentially a sub-calculus of our simplified MIL [3]. In our setting the reductions that their algorithms perform are

[3] In fact their calculus is n-ary, as is the full version of MIL.

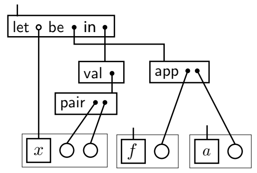

Fig. 1. Pictorial representation of let x be app(f, a) in val(pair(x, x))

equivalent to: $\rightarrow .\beta_1$-, $\times.\beta$-, $T.\beta_0$-, and a restriction of $\rightarrow .\beta_0$-reduction. Appel and Jim show that their calculus is confluent in the presence of these reductions, and other 'δ-rules' satisfying certain criteria.

The reductions rely on knowing the number of occurrences of a particular variable. The quadratic algorithms store this information in a table *Count* mapping variable names to their number of occurrences. Appel and Jim's naïve algorithm repeatedly (i) zeros the usage counts, (ii) performs a *census* pass over the whole term to update the usage counts and then (iii) traverses the term performing reductions on the basis of the information in *Count*, until there are no redexes remaining.

The improved algorithm, used in SML/NJ and SML.NET, dynamically updates the usage counts as reductions are performed. This allows more reductions to be performed on each pass, and only requires a full census to be performed once. The improved algorithm is better in practice, but both algorithms have worst-case time complexity $\Theta(n^2)$ where n is the size of the input term.

Appel and Jim's imperative algorithm runs in linear time and uses a pointer-based representation of terms which directly links all occurrences of a particular variable. This enables an efficient test to see if removing an occurrence will create any new redexes, and an efficient way of jumping to any such redexes. The algorithm first traverses the program tree collecting the set of all redexes. Then it repeatedly removes a redex from the set and reduces it in-place (possibly adding new redexes to the set), until none remain.

4 A Graph-based Representation

Our imperative algorithm works with a mutable graph representation comprising a doubly-linked expression tree and a list of pairs of circular doubly-linked lists collecting all the recursive (respectively non-recursive) uses of each variable. Such graphs can naturally be presented pictorially as shown by the example in Fig. 1.

Figure 2 shows the β-reductions for functions in this pictorial form. We find the pictorial representation intuitively very useful, but awkward to reason with or use in presenting algorithms. Hence, like Appel and Jim, we will work with a more abstract structure comprising an expression tree and a collection of maps which capture the additional graphical structure between nodes of the tree.

The structure of expression trees is determined by the abstract syntax of simplified MIL. In order to capture mutability we use ML-style references. Each

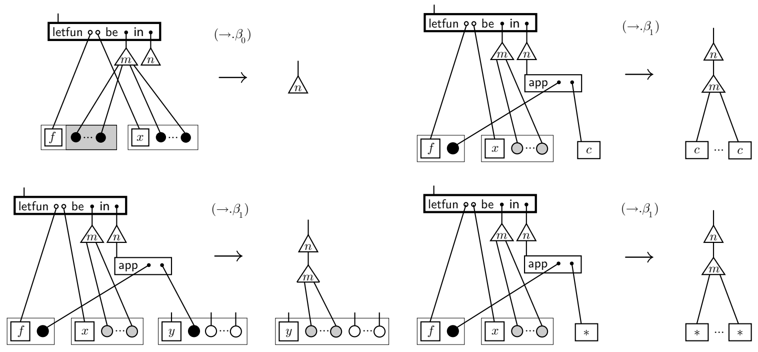

Fig. 2. Graph reductions

node of the expression tree is a reference cell. We call the entities which reference cells contain *objects*. Given a reference cell l, we write $!l$ to denote the contents of l, and $l := u$ to denote the assignment of the object u to l.

Atoms	$!a, !b ::=$	$r \mid * \mid c$
Values	$!v, !w ::=$	$a \mid \mathsf{pair}(a, b) \mid \mathsf{proj}_1(a) \mid \mathsf{proj}_2(a) \mid \mathsf{inj}_1(a) \mid \mathsf{inj}_2(a)$
Computations	$!m, !n, !p ::=$	$\mathsf{app}(a, b) \mid \mathsf{letfun}\ f(x)\ \mathsf{be}\ m\ \mathsf{in}\ n$
		$\mid \mathsf{val}(v) \mid \mathsf{let}\ x\ \mathsf{be}\ m\ \mathsf{in}\ n \mid \mathsf{case}\ a\ \mathsf{of}\ (x_1)n_1\ ;\ (x_2)n_2$

$$e ::= v \mid m \qquad d ::= e \mid x \mid r$$

where f, g, x, y, z range over defining occurrences, and r, s, t over uses. We write *parent*(e) for the parent of the node e. *root* is a distinguished sentinel, marking the top of the expression tree. The object dead is used to indicate a *dead* node. If a node is dead then it has no parent. The *root* node is always dead. We define the children of an expression node e, as the set of nodes appearing in $!e$.

Initially both *parent* and *children* are entirely determined by the expression tree. However, in our algorithm we take advantage of the *parent* map in order to classify expression nodes as active or inactive. We ensure that the following invariant is maintained: for all expression nodes e, either

- e is active: *parent*$(d) = e$, for all $d \in$ *children*(e);
- e is inactive: *parent*$(d) =$ dead for all $d \in$ *children*(e); or
- e is dead: $!e =$ dead.

We define *splicing* as the operation which takes one subtree m and substitutes it in place of another subtree n. The subtree m is removed from the expression tree and then reintroduced in place of n. The parent map is adjusted accordingly for the children of m. We define *splicing a copy* as the corresponding operation which leaves the original copy of m in place. The operation $\lceil q \rceil$ returns a node containing q, with parent *root*. We will also use $\lceil \cdot \rceil$ in pattern-matching.

The *def-use* maps abstract the structures used for representing occurrences:

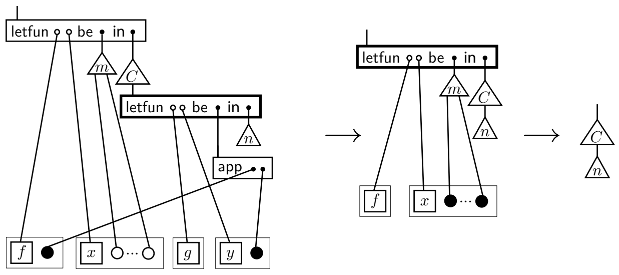

Fig. 3. Triggering non-local reductions

- $def(r)$ gives the defining occurrence of the use r.
- $non\text{-}rec\text{-}uses(x)$ is the set of non-recursive uses of the defining occurrence x.
- $rec\text{-}uses(x)$ is the set of recursive uses of the defining occurrence x.

In the real implementation occurrences are held in a pair of doubly-linked circular lists, such that each pair of lists intersects at a defining occurrence. We find it convenient to overload the maps to be defined over all occurrences and also define some additional maps:

$$non\text{-}rec\text{-}uses(r) = non\text{-}rec\text{-}uses(def(r))$$
$$rec\text{-}uses(r) = rec\text{-}uses(def(r)) \qquad def(x) = x$$
$$occurrences(r) = uses(r) \cup \{def(r)\} \quad uses(r) = non\text{-}rec\text{-}uses(r) \cup rec\text{-}uses(r)$$

None of these additional definitions affects the implementation.

The graph structure allows constant time movement up and down the expression tree in the normal way, but also allows constant time non-local movement via the occurrence lists. For example, consider the dead-function eliminations:

$$\begin{aligned} & \mathsf{letfun}\ f(x)\ \mathsf{be}\ m\ \mathsf{in}\ C[\mathsf{letfun}\ g(y)\ \mathsf{be}\ \mathsf{app}(f,y)\ \mathsf{in}\ n] \\ \longrightarrow_{(\rightarrow.\beta_0)} & \mathsf{letfun}\ f(x)\ \mathsf{be}\ m\ \mathsf{in}\ C[n] \quad \longrightarrow_{(\rightarrow.\beta_0)} \quad C[n] \end{aligned}$$

where $f, g \notin fv(C, n)$, illustrated in Fig. 3. After one reduction, g is dead, so its definition can be deleted, removing the use of f. But this use was connected to its defining occurrence, and f is now dead. The defining occurrence is connected to its parent, so the new dead-function redex can be reduced.

5 A One-pass Algorithm

In contrast to Appel and Jim's imperative algorithm, the algorithm we have implemented operates in one-pass. Essentially, the one-pass algorithm performs a depth-first traversal of the expression tree, reducing redexes on the way back up the tree. Of course, these reductions may trigger further reductions elsewhere in the tree. By carefully deactivating parts of the tree, we are able to control the reduction order and limit the testing required for new redexes. Here is an outline of our one-pass imperative algorithm:

```
$contract(e) = reduceCCs(e)$
               $deactivate(e)$
               apply $contract$ to children of $e$
               $reactivate(e)$
               $reduce(e)$
  $reduce(e) =$ if $e$ is a redex then
                    reduce $e$ in place
                    perform further reductions triggered by reducing $e$
```

The operation $reduceCCs(e)$ performs commuting conversions on the way down the tree. The order of commuting conversions can have a significant effect on code quality, a poor choice leading to many jumps to jumps. We have found that the approach of doing them on the way down works well in practice (although the contract algorithm would still be valid without the call to $reduceCCs$).

```
$reduceCCs(e) =$ case $!e$ of
    ($\mathsf{let}\ y\ \mathsf{be}\ e'\ \mathsf{in}\ p$) $\Rightarrow$
        if $reduceCC(e, t, e', p) \neq \emptyset$ then $reduceCCs(e)$ else skip
    ($\_$) $\Rightarrow$ skip
$reduceCC(e, y, e', p) =$ case $e'$ of
    ($\mathsf{letfun}\ f(x)\ \mathsf{be}\ m\ \mathsf{in}\ n$) $\Rightarrow$
        splice $\lceil \mathsf{let}\ y\ \mathsf{be}\ n\ \mathsf{in}\ p \rceil$ in place of $e'$
        splice $\lceil \mathsf{letfun}\ f(x)\ \mathsf{be}\ m\ \mathsf{in}\ e' \rceil$ in place of $e$
        return $\{e'\}$
    ... (let and case are similar)
    ($\_$) $\Rightarrow$ return $\emptyset$
```

Note that commuting conversions can also be triggered by other reductions. The return value for $reduceCC$ will be used in the definition of $reduce$ in order to catch reductions which are triggered by applying commuting conversions.

$deactivate(e)$ deactivates e: $parent(d)$ is set to dead for every $d \in children(e)$.
$reactivate(e)$ reactivates e: $parent(d)$ is set to e for every $d \in children(e)$.

Deactivating nodes on the way down prevents reductions from being triggered above the current node in the tree. On the way back up the nodes are reactivated, allowing any new redexes to be reduced. Because subterms are known to be normalised, fewer tests are needed for new redexes. Consider, for example:

$$\mathsf{let}\ y\ \mathsf{be}\ (\mathsf{let}\ x\ \mathsf{be}\ m\ \mathsf{in}\ n)\ \mathsf{in}\ p \ \longrightarrow_{T.CC}\ \mathsf{let}\ x\ \mathsf{be}\ m\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ n\ \mathsf{in}\ p$$

Because we know that $\mathsf{let}\ x\ \mathsf{be}\ m\ \mathsf{in}\ n$ is in normal form, m cannot be of the form $\mathsf{let}(\dots), \mathsf{letfun}(\dots), \mathsf{case}(\dots)$ or $\mathsf{val}(\dots)$. Hence, it is not necessary to check whether $\mathsf{let}\ x\ \mathsf{be}\ m\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ n\ \mathsf{in}\ p$ is a redex. (Of course, $\mathsf{let}\ y\ \mathsf{be}\ n\ \mathsf{in}\ p$ may still be a redex, and indeed exposing such redexes is one of the main purposes of performing CC-reduction.)

Rather than placing them in a redex set, as in Appel and Jim's imperative algorithm, *reduce*(e) reduces any new redexes created inside e (but none that are created above e in the expression tree). If *reduce*(e) is invoked on an expression node which is not a redex, then no action is performed. The *reduce* function also returns a boolean to indicate whether a reduction took place. As we shall see, this is necessary in order to detect the triggering of new reductions. We now expand the definition of *reduce*.

```
$reduce(e)$ = case $!e$ of
    ($\mathsf{letfun}\ f(x)\ \mathsf{be}\ m\ \mathsf{in}\ n$) $\Rightarrow$
        if $non\text{-}rec\text{-}uses(f) = \emptyset$ then
            splice $n$ in place of $e$
            $reduceOccs(cleanExp(m))$
            return true
        else if $rec\text{-}uses(f) = \emptyset$ and $non\text{-}rec\text{-}uses(f) = \{f'\}$ then
            let $focus = parent(parent(f'))$
            case $!focus$ of
            ($\mathsf{app}(f', a)$ $\Rightarrow$
                splice $n$ in place of $e$
                splice $m$ in place of $focus$
                let $(occs, redexes) = substAtom(x, a)$
                $reduceOccs(occs \cup cleanExp(a))$
                $reduceRedexes(redexes)$
                return true
            (_) $\Rightarrow$ return false
        else return false
    ($\mathsf{let}\ x\ \mathsf{be}\ \lceil \mathsf{val}(v) \rceil\ \mathsf{in}\ n$) $\Rightarrow$
        if $uses(x) = \emptyset$ then
            splice $n$ in place of $e$
            $reduceOccs(cleanExp(parent(v)))$
            return true
        else if $v$ is an atom $a$ then
            splice $n$ in place of $e$
            let $(occs, redexes) = substAtom(x, a)$
            $reduceOccs(occs \cup cleanExp(parent(a)))$
            $reduceRedexes(redexes)$
            return true
        else case $!v$ of
        ($\mathsf{pair}(a, b)$) $\Rightarrow$
            if $e$ is fresh then
                let $redexes = reduceProjections(e, x, a, b, uses(x))$
                if $redexes = \emptyset$ then return false
                else
                    $reduceRedexes(redexes)$
                $reduce(e)$
                return true
            else return false
        $(\mathsf{inj}_i(a)) \Rightarrow$
            if $e$ is fresh then
                let $(occs, redexes) = reduceCases(e, x, i, a, uses(x))$
                if $redexes = \emptyset$ then return false
                else
                    $reduceOccs(occs)$
                    $reduceRedexes(redexes)$
                    $reduce(e)$
                    return true
            else return false
        $(\_) \Rightarrow$ return false
    $(\mathsf{let}\ y\ \mathsf{be}\ e'\ \mathsf{in}\ p) \Rightarrow$
        let $redexes = reduceCC(e, y, e', p)$
        for $e'' \in redexes$ do $reduce(e'')$
        return true
    $(\_) \Rightarrow$ return false
```

The first case covers β-reductions on functions, with two sub-cases:

– $(\rightarrow.\beta_0)$ If the function is dead, its definition is removed, the continuation spliced in place of e, and any uses within the dead body deleted, possibly triggering new reductions.
– $(\rightarrow.\beta_1)$ If the function has one occurrence, which is non-recursive, it is inlined. The continuation of e is spliced in place of e, the function body is inlined with the argument substituted for the parameter, and the argument deleted. Substitution may trigger further reductions.

The second case covers β-reductions on computations as well as some instances of β-reduction on products and sums. It is divided into four sub-cases.

– $(T.\beta_0)$ If a value is dead, then its definition can be removed. The continuation is spliced in place of e. Then the uses inside the dead function body are deleted, possibly triggering new reductions.
– $(T.\beta_a)$ If a value is atomic, then it can be inlined. First the continuation of e is spliced in place of e. Then the atom is substituted for the bound variable. Finally the atom is deleted.
– $(\times.\beta)$ If a pair is bound to a variable x, and this is the first time e has been visited, then any projections of x are reduced. If this is the first time e has been visited, then we say that e is *fresh*. In practice freshness is indicated by setting a global flag. For efficiency, new projections will subsequently be reduced as and when they are created.
– $(+.\beta)$ This follows exactly the same pattern as $\times.\beta$-reduction. The only difference is that the reduction itself is more complex, so can trigger new reductions in different ways.

The third case deals with commuting conversions.

The algorithm ensures that the current reduction is complete before any new reductions are triggered. Potential new redexes created by the current reduction are encoded and executed after the current reduction has completed.

reduceUp(e) reduces above e as far as possible:

```
$reduceUp(e) = \text{if } reduce(e) \text{ then } reduceUp(parent(e)) \text{ else skip}$
```

reduceRedexes reduces a set of expression redexes, whilst *reduceOccs* reduces a set of occurrence redexes:

```
$reduceRedexes(redexes) = \text{for each } e \in redexes \text{ do } reduceUp(e)$
$reduceOccs(xs) = \text{for each } r \in xs \text{ do}$
    $\text{if } isSmall(r) \text{ then } reduceUp(parent(def(r))) \text{ else skip}$
$isSmall(r) = r \notin rec\text{-}uses(r) \text{ and } |non\text{-}rec\text{-}uses(r)| \leq 1$
```

cleanExp(e) removes all occurrences and subexpressions inside e and returns a set of occurrence redexes.

```
$cleanExp(e) = \text{case } !e \text{ of}$
    $(r) \Rightarrow$
        $e := \mathsf{dead}$
        $\text{return } deleteUse(r)$
    $(\mathsf{letfun}\ f(x)\ \mathsf{be}\ m\ \mathsf{in}\ n) \Rightarrow$
        $e, f, x := \mathsf{dead}$
        $\text{return } cleanExp(m) \cup cleanExp(n)$
    $(\mathsf{app}(a, b)) \Rightarrow$
        $e := \mathsf{dead}$
        $\text{return } cleanExp(a) \cup cleanExp(b)$
    $\ldots$
```

Remark Marking nodes as dead ensures that unnecessary work is not done on dead redexes. A crucial difference between the imperative algorithms and the improved quadratic one is that reduction in the former immediately detects new redexes, whereas the improved quadratic algorithm only detects new (non-local) redexes on a subsequent traversal.

deleteUse(r) removes r and returns a set of 0 or 1 occurrence redexes:

```
$deleteUse(r) =$
    $\text{if } r \text{ is already dead then return } \emptyset$
    $\text{let } s = nextOcc(r)$
    $uses(s) := uses(s) - \{r\}$
    $\text{return } \{s\}$

$nextOcc(r) =$
    $\text{let } x = def(r)$
    $\text{if } r \text{ is non-recursive then return } s \in (non\text{-}rec\text{-}uses(x) \cup \{x\}) - \{r\}$
    $\text{else if } r \text{ is recursive then return } s \in (rec\text{-}uses(x) \cup \{x\}) - \{r\}$
```

$\mathit{reduceProjections}(e, x, a_1, a_2, xs)$ reduces projections indexed by xs. e is an expression node of the form $\mathsf{let}\ x\ \mathsf{be}\ \mathsf{val}(\mathsf{pair}(a_1, a_2))\ \mathsf{in}\ m$, and xs is a subset of the uses of x.

```
$\mathit{reduceProjections}(e, x, a_1, a_2, xs) =$
    let $\mathit{redexes} := \emptyset$
    for each $s \in xs$ do
        let $\mathit{focus} = \mathit{parent}(\mathit{parent}(s))$
        case $!\mathit{focus}$ of
        $(\mathsf{proj}_i(s)) \Rightarrow$
            splice a copy of $a_i$ in place of $\mathit{focus}$
            $\mathit{redexes} := \mathit{redexes} \cup \{\mathit{parent}(\mathit{focus})\}$
        $(\_) \Rightarrow$ skip
    return $\mathit{redexes}$
```

All the projections in which a member of xs participates are reduced, and a set of expression redexes is constructed. Each projection can trigger the creation of a new $T.\beta_a$-redex. For instance, consider:

$$\begin{array}{rl} & \mathsf{let}\ x\ \mathsf{be}\ \mathsf{val}(\mathsf{pair}(a, b))\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \mathsf{val}(\mathsf{proj}_1(x))\ \mathsf{in}\ m \\ \longrightarrow_{\times.\beta} & \mathsf{let}\ x\ \mathsf{be}\ \mathsf{val}(\mathsf{pair}(a, b))\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \mathsf{val}(a)\ \mathsf{in}\ m \\ \longrightarrow_{T.\beta_a} & \mathsf{let}\ x\ \mathsf{be}\ \mathsf{val}(\mathsf{pair}(a, b))\ \mathsf{in}\ m \end{array}$$

$\mathit{reduceCases}(e, x, i, a, xs)$ reduces case-splits indexed by xs. e is an expression node of the form $\mathsf{let}\ x\ \mathsf{be}\ \mathsf{val}(\mathsf{inj}_i(a))\ \mathsf{in}\ m$, and xs is a subset of the uses of x.

```
$\mathit{reduceCases}(e, x, i, a, xs) =$
    let $\mathit{occs} := \emptyset$
    let $\mathit{redexes} := \emptyset$
    for each $s \in xs$ do
        let $\mathit{focus} = \mathit{parent}(\mathit{parent}(s))$
        case $!\mathit{focus}$ of
        $(\mathsf{case}\ s\ \mathsf{of}\ (x_1)n_1\ ;\ (x_2)n_2) \Rightarrow$
            $\mathit{occs} := \mathit{occs} \cup \mathit{cleanExp}(n_{3-i})$
            $\mathit{deleteUse}(s)$
            splice $n_i$ in place of $\mathit{focus}$
            let $(\mathit{occs}', \mathit{redexes}') = \mathit{substAtom}(x_i, a)$
            $\mathit{occs} := \mathit{occs} \cup \mathit{occs}'$
            $\mathit{redexes} := \mathit{redexes} \cup \mathit{redexes}' \cup \{\mathit{parent}(\mathit{focus})\}$
            $x_1, x_2 := \mathsf{dead}$
        $(\_) \Rightarrow$ skip
    return $(\mathit{occs}, \mathit{redexes})$
```

The structure of *reduceCases* is similar to that of *reduceProjections*. However, it is slightly more complex because a single $+.\beta$-reduction inlines multiple atoms, splices one branch of a case and discards the other. Discarding the branch which is not taken gives a set of occurrence redexes as well as the expression redexes.

$\mathit{substAtom}(x, a)$ substitutes the atom a for all the uses of the defining occurrence x. It returns a pair of a set of occurrence redexes and a set of expression redexes.

```
$\mathit{substAtom}(x, a) = \text{case } (!a) \text{ of}$
    $(r) \Rightarrow \mathit{substUse}(x, r)$
    $(\_) \Rightarrow$
        for each $r \in \mathit{uses}(x)$ do
            splice a copy of $a$ in place of $r$
            $x := \mathsf{dead}$
        return $(\emptyset, \emptyset)$
```

This is straightforward for non-variable atoms, as it cannot generate new redexes. In contrast, substituting a variable can trigger $\times.\beta$- and $+.\beta$-reductions.

$\mathit{substUse}(x, r)$ substitutes r for all the uses of the defining occurrence x.

```
$\mathit{substUse}(x, r) =$
    let $\mathit{xs} = \mathit{uses}(x)$
    if $r \in \mathit{rec\text{-}uses}(r)$ then
        $\mathit{rec\text{-}uses}(r) := \mathit{rec\text{-}uses}(r) \cup \mathit{xs}$
    else if $r \in \mathit{non\text{-}rec\text{-}uses}(r)$
        $\mathit{non\text{-}rec\text{-}uses}(r) := \mathit{non\text{-}rec\text{-}uses}(r) \cup \mathit{xs}$
    $x := \mathsf{dead}$
    let $e = \mathit{parent}(\mathit{def}(r))$
    case $!e$ of
    $(\mathsf{let}\ y\ \mathsf{be}\ \mathsf{val}(\lceil \mathsf{pair}(a_1, a_2) \rceil)\ \mathsf{in}\ m) \Rightarrow$
        for each $s \in \mathit{xs}$ do $\mathit{def}(s) := \mathit{def}(r)$
        let $\mathit{redexes} = \mathit{reduceProjections}(e, y, a_1, a_2, \mathit{xs})$
        return $(\emptyset, \mathit{redexes})$
    $(\mathsf{let}\ y\ \mathsf{be}\ \mathsf{val}(\lceil \mathsf{inj}_i(a_i) \rceil)\ \mathsf{in}\ m) \Rightarrow$
        for each $s \in \mathit{xs}$ do $\mathit{def}(s) := \mathit{def}(r)$
        let $(\mathit{occs}, \mathit{redexes}) = \mathit{reduceCases}(e, y, i, a_i, \mathit{xs})$
        return $(\mathit{occs}, \mathit{redexes})$
    $(\_) \Rightarrow \text{return } (\emptyset, \emptyset)$
```

Substitution is implemented by merging two sets together. Concretely, this amounts to the constant-time operation of inserting one doubly-linked circular list inside another. In addition, if x is bound to a pair, then projections are reduced, or if x is bound to an injection, then case-splits are reduced.

6 Analysis

There are two obvious operations mapping terms from the functional to the imperative representations, which we call *mutify* and *demutify*, respectively. We have a semi-formal argument for the following:

Proposition 3. *Let e be a term and $e' = (\mathit{demutify} \circ \mathit{contract} \circ \mathit{mutify})(e)$. Then e' is a normal form for e.*

The argument uses the invariants of Sect. 4, plus the invariant that the children of the current node are in normal form. When new redexes are created, this invariant is modified such that subterms may contain redexes, but only those stored in appropriate expression redex sets or occurrence redex sets. It is reasonably straightforward to verify that the operations which update the graph structure do in fact correspond to MIL reductions. When *contract* terminates, all the redex sets are empty and the term is in normal form.

Although our approach of performing CCs on the way down the tree works well in practice, the worst case time complexity is still quadratic in the size of the term. The algorithms Appel and Jim considered do not perform commuting conversions. We define a version of our algorithm $contract_\beta$ which does not perform commuting conversions. This is obtained simply by removing the call to *reduceCCs* from *contract*, and the test for commuting conversions from *reduce*.

Proposition 4. *$contract_\beta(e)$ is linear in the size of e.*

The argument is very similar to that of Appel and Jim [1] for their imperative algorithm. Essentially most operations take constant time and shrink the size of the term. The only exception is substitution. In the case where a non-variable is substituted for a variable x, the operation is linear in the number of uses of x. But it is only possible to subsitute a non-variable for a variable once, therefore the total time spent substituting atoms is linear. In the case where a variable y is substituted for a variable x, the operation is constant, providing y is not bound to a pair or an injection. If y is bound to a pair or an injection, then the operation is linear in the number of uses of x. Again, once bound to a pair or an injection, a variable cannot be rebound, so the time remains linear.

Crucially, this argument relies on the fact that back pointers from uses back to defining occurrences are only maintained for pairs and injections. In our SML.NET implementation we found that maintaining back pointers from *all* uses back to defining occurrences does not incur any significant cost in practice. Even when bootstrapping the compiler ($\sim$ 80,000 lines of code) there was no discernible difference in compile time. Maintaining back pointers also allows us to perform various other rewrites including η-reductions. In the presence of all backpointers, optimising the union operation to always add the smaller list to the larger one guarantees $O(n \ log \ n)$ behaviour. Using an efficient *union-find* algorithm would restore essentially linear complexity.

7 Performance

We have extended our one-pass imperative algorithm *contract* to the whole of MIL and compared its performance with the current implementation of *simplify*. Replacing *simplify* with *contract* is not entirely straightforward, as all the other phases in the pipeline are written to work on a straightforward immutable tree datatype for terms, which is incompatible with the representation used in *contract*. We therefore make use of *mutify* and *demutify* to change representation before and after *contract*. Since both *mutify* and *demutify* completely rebuild

Table 1. Total compile time (seconds)

Benchmark	Lines of code	SML/NJ *simplify*	SML/NJ *mcd*	MLton *simplify*	MLton *mcd*
`sort`	70	2.11	3.47	0.46	0.52
`xq`	1,300	13.1	14.4	2.46	1.76
`mllex`	1,400	11.6	16.0	2.39	2.03
`raytrace`	2,500	18.1	24.0	4.30	3.03
`mlyacc`	6,200	57.3	43.8	10.0	6.04
`hamlet`	20,000	219	156	43.7	26.2
`bootstrap`	80,000	1310	1190	289	221

the term, they are very expensive – calling *mutify* and *demutify* generally takes longer than *contract* itself. Ideally, of course, all the phases would use the same representation. However, using two representations allowed us to compare the running times of *simplify* and *contract* on real programs.

Table 1 compares the total compile times of several benchmark programs for the existing compiler, using *simplify*, and for the modified one, using *demutify* ∘ *contract* ∘ *mutify*. Each benchmark was run under two different versions of SML.NET. One was compiled under SML/NJ and the other under MLton. Benchmarks were run on a 1.4Ghz AMD Athlon PC equipped with 512MB of RAM and Microsoft Windows XP SP1.

The first five benchmarks are demos distributed with SML.NET. The `sort` benchmark simply sorts a list of integers using quicksort. `xq` is an interpreter for an XQuery-like language for querying XML documents. `mllex` and `mlyacc` are ports of SML/NJ's ml-lex and ml-yacc utilities. `raytrace` is a port to SML of the winning entry from the Third Annual ICFP Programming Contest. The remaining benchmarks are much larger. `hamlet` is Andreas Rossberg's SML interpreter, whilst `bootstrap` is SML.NET compiling itself.

On small benchmarks, the current compiler is faster. But for medium and large benchmarks, we were surprised to discover that *demutify* ∘ *contract* ∘ *mutify* is faster than *simplify*, even though much of the time is spent in useless representation changes.

Table 2 compares the time spent in *simplify* with the time spent in each of *mutify*, *contract* and *demutify*. Figure 4 gives a graphical comparison. *tcontract* is the total compile time using the modified compiler, and *tsimplify* is the total compile time using the existing compiler. Under SML/NJ, a decrease of nearly 30% in the total compile time is seen in some cases. Under MLton, there is a decrease of up to 40% in total compile time. This is a significant improvement, given that in the existing compiler only around 50% of compile time is spent performing shrinking reductions. Comparing the actual shrinking reduction time, *contract* is up to four times faster than *simplify* under SML/NJ, and up to 15 times faster under MLton. The level of improvement under MLton is striking. Our results suggest that MLton is considerably better than SML/NJ at compiling ML code which makes heavy use of references.

Table 2. Shrinking reduction time (in seconds) under SML/NJ and MLton

	Under SML/NJ					Under MLton				
Benchmark	Total		Breakdown of *mcd*			Total		Breakdown of *mcd*		
	simplify	*mcd*	*m*	*c*	*d*	*simplify*	*mcd*	*m*	*c*	*d*
`sort`	1.00	2.00	0.87	0.70	0.43	0.22	0.11	0.02	0.07	0.02
`xq`	5.86	5.98	1.90	3.61	0.47	1.46	0.54	0.35	0.15	0.06
`mllex`	6.09	7.49	3.31	3.16	1.02	1.21	0.57	0.27	0.23	0.07
`raytrace`	9.32	11.8	5.16	5.44	1.17	2.13	0.65	0.37	0.19	0.09
`mlyacc`	33.2	20.0	9.42	8.60	1.94	5.63	1.26	0.68	0.37	0.21
`hamlet`	84.5	56.4	26.2	21.5	8.59	23.3	5.54	1.85	2.77	0.92
`bootstrap`	439	282	130	100	53.0	107	36.6	11.8	18.4	6.38

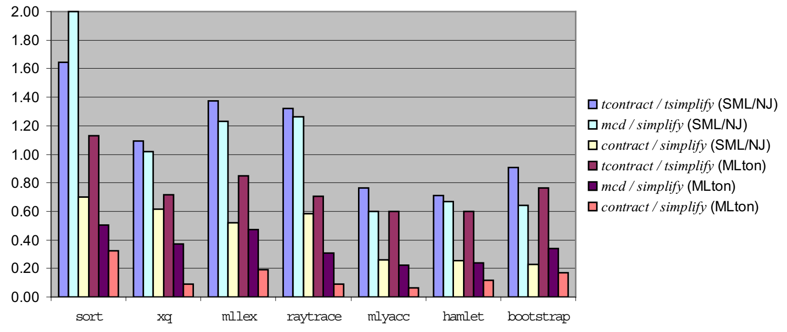

Fig. 4. Comparing *contract* with *simplify*

As an exercise, one of the other transformations *deunit*, which removes redundant units was translated to use the new representation. The *contract* function is called before and after *deunit*, so this enabled us to eliminate one call to *demutify* and one call to *mutify*. This translation was easy to do and did not change the performance of *deunit*. We believe that it should be reasonably straightforward, if somewhat tedious, to translate the rest of the transformations to work directly with the mutable representation.

8 Conclusions and Further Work

We have implemented and extended Appel and Jim's imperative algorithm for shrinking reductions and shown that it can yield significant reductions in compile times relative to the algorithm currently used in SML/NJ and SML.NET. The improvements are such that, for large programs, it is even worth completely changing representations before and after *contract*, but this is clearly suboptimal.

The results of this experiment indicate that it would be worth the effort of rewriting other phases of the compiler to use the graph-based representation.

Making more extensive use of the pointer-based representation would allow many transformations to be written in a different style, for example replacing explicit environments with extra information on binding nodes, though this does not interact well with the hash-consing currently used for types. We also believe that 'code motion' type transformations can be more easily and efficiently expressed.

More speculatively, we would like to investigate more principled mutable graph-based intermediate representations. There has been much theoretical work on graph-based representations of proofs and programs, yet these do not seem to have been exploited in compilers for higher-order languages (though of course, compilers for imperative languages have used a mutable flow-graph representations for decades). With a careful choice of representation, some of our transformations (such as $T.CC$) could simply be isomorphisms and we believe that a better treatment of shared continuations in the other commuting conversions would also be possible.

References

1. Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5):515–540, 1997.
2. N. Benton and A. Kennedy. Monads, effects and transformations. In *3rd International Workshop on Higher Order Operational Techniques in Semantics (HOOTS), Paris*, volume 26 of *ENTCS*. Elsevier, September 1999.
3. N. Benton, A. Kennedy, and C. Russo. SML.NET. `http://www.cl.cam.ac.uk/Research/TSG/SMLNET/`, June 2002.
4. N. Benton, A. Kennedy, and C. Russo. Adventures in interoperability: The SML.NET experience. In *Proc. 6th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP)*, August 2004.
5. Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *Proc. ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 129–140, 1999.
6. O. Danvy. A new one-pass transformation into monadic normal form. In *Proc. 12th International Conference on Compiler Construction*, number 2622 in Lecture Notes in Computer Science, pages 77–89. Springer, 2003.
7. Ecma International. ECMA and ISO C# and Common Language Infrastructure standards , December 2002. `http://www.ecma-international.org/publications/standards/Ecma-334.htm`.
8. J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *Proc. 21st Annual Symposium on Principles of Programming Languages*. ACM, January 1994.
9. Sam Lindley. *Normalisation by evaluation in the compilation of typed functional programming languages*. PhD thesis, The University of Edinburgh, September 2004. (Submitted for examination).
10. S. L. Peyton Jones and A. L.M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 1998.