# C-SSAPRE: Eliminating redundancy over copies

Tom Murphy VII, Brendan McMahan

7 May 2003

## 1 Introduction

Partial Redundancy Elimination (PRE) is an optimization that prevents, where possible, the re-computation of an expression along some paths in a program. PRE is general enough to automatically implement loop-invariant code hoisting and global common sub-expression elimination. PRE appears to be especially beneficial for functional programs, where the class of expressions that may be moved is much richer: in addition to arithmetic and memory loads, we may also move and coalesce projections from tuples, calls to total functions, and even allocations of persistent values.

The SSAPRE [5, 7] algorithm implements Partial Redundancy Elimination for programs in a static single assignment (SSA) intermediate representation (IR). However, the algorithms makes several assumptions about the intermediate representation that do not hold in all cases. In particular, they do not hold for the SSA IR of MLton [2], a high performance whole-program compiler for Standard ML [8]. We introduce a new PRE algorithm, C-SSAPRE, which finds PRE optimization across copies, detecting redundencies not possible with the original SSAPRE algorithm. Our algorithm can be applied directly to MLton's IR, which lacks explicit copies.

We begin by describing the MLton SSA IR and its differences from what we will call "standard" SSA, as given by Cytron et al. [6]. This provides a concrete framework for our discussion of our generalization of the SSAPRE algorithm. We then provide detailed pseudocode for the pre-pass that we use before invoking a modified version of SSAPRE. Finally, we discuss our conclusions, important open questions, and potential ideas for future work. An appendix gives an improved pre-pass algorithm.

## 2 The MLton SSA IR

The MLton compiler performs a sophisticated transformation of the functional source language into an efficient form with explicit loops (much like a C or Java compiler might produce), but only a few simple optimizations are done on this representation. For instance, loop-invariant expressions are only hoisted via common sub-expression elimination if they happen to be computed in a block that dominates the loop header. Thus, MLton should benefit significantly from a PRE pass, making it a good testbed for our algorithm.

MLton's low-level intermediate representation is a Static Single Assignment Control Flow Graph (CFG). The representation is similar to SSA-based IRs in some modern SSA-based C compilers, but has some differences that will impact the way that we implement PRE. Figure (2) provides a concrete example.

The most obvious differences between the MLton IR and standard SSA is that MLton uses labeled GOTOs instead of $\phi$ functions, and that MLton does not keep track of variable versions. In this discussion and in the sequel unless otherwise stated we assume we are optimizing an arbitrary operator `op` that takes a single argument. The extension to operators on multiple operands is straightforward.

- **Lack of Variable Versions** In traditional SSA, different definitions of the same variable become "versions" of the variable $(a_0, a_1, \ldots)$. In MLton they are simply different variables. This decouples the IR variables from their source language identities, which has important repercussions for our PRE pass.

- **Labeled GOTOs instead of $\phi$-functions**. Blocks in the MLton IR may take arguments. Any jump to such a block must pass along arguments, which are then assigned into the formal parameter(s) of the block. Though this makes a procedure look like a series of mutually-recursive functions, the scope of these variables extends to the entire procedure. Each formal parameter of a block label corresponds to the left-hand-side of a $\phi$ function. Each $\phi$ argument appears in the appropriate position in call to the label in the corresponding predecessor block. Thus, labeled GOTOs are really just an alternate notation for $\phi$-functions; one can translate between this representation and standard SSA without any loss of information.

  Standard SSA form has the additional assumption that $\phi$-functions only have different versions of the same variable as operands, and then assign to a new version of this variable. It may be necessary to insert additional copies to maintain this property. MLton does not enforce this assumption.

  In Figure (2), the left CFG diagram shows code in the MLton IR form. Instead of having a $\phi$-function for block $L3$, the block takes an argument $a$, with $\phi$-operands $x$ and $y$ for blocks $L1$ and $L2$, respectively. The right diagram shows the corresponding code in standard SSA form. Note that an extra copy has been inserted and variable renaming performed so that the $\phi$ function in block $B3$ contains only different versions of $a$ and also assigns to a version of $a$.

  While the MLton IR is different than the standard SSA formulation, it is still a true SSA representation in that each variable has exactly one definition point, and this point dominates all of its uses.

- **Higher-level expression constructs**. SML does not offer primitive pointer operations; instead programmers have primitives for creating algebraic data types and records. Thus, the MLton IR has expression forms for allocating and selecting from records, building and case-analyzing data structures, and other high-level operations. Because these are effect-free (but expensive) operations, the potential benefit from PRE on these operations is large.

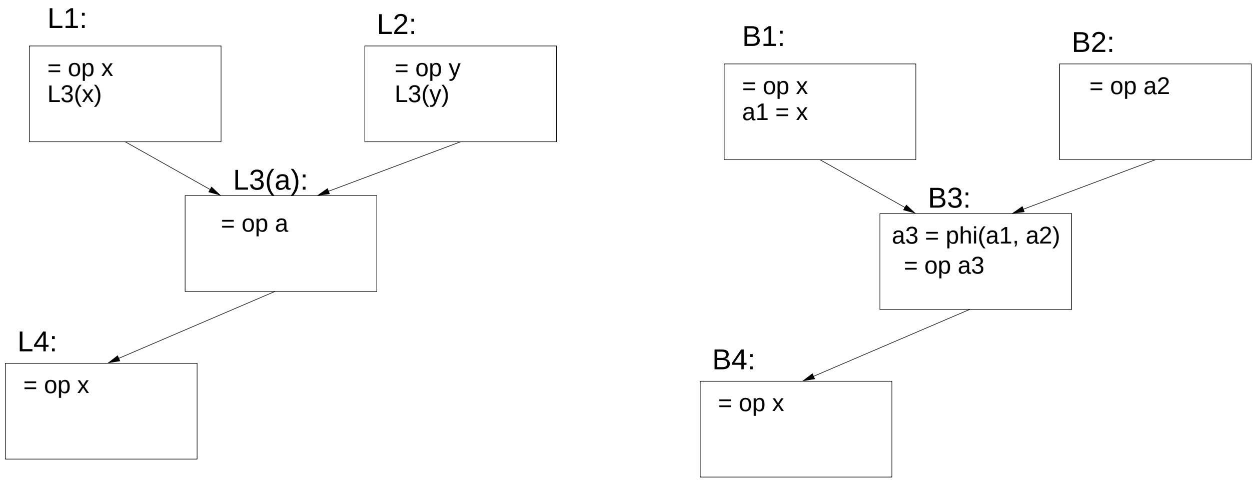


Figure 1: MLton SSA IR vs. Standard SSA

- **Overflow checking on integer computations**. SML provides overflow-checking for integer arithmetic. This means that these operations will be essentially immovable by PRE, since they are effectful[1]. On the other hand, `word` types and infinite-precision integers do not have overflow checking, and MLton provides a flag to disable overflow checking. In these cases we will be able to move arithmetic operations in addition to the functional expressions mentioned earlier.

# 3 Relaxing Assumptions on $\phi$-functions

In [7], the authors make the assumption that "each $\phi$ assignment has the property that its left-hand side and all of its operands are versions of the same original program variable" and that "the live ranges of different versions of the same original program variable do not overlap." That is, they assume the IR is in standard SSA form, as is that generate by the algorithms in [6].

The SSA representation of MLton does not enforce these requirements, and so to perform useful optimization the SSAPRE algorithm must be modified. We refer to SSA representations that allow $\phi$-functions on arbitrary variables, such as MLton's IR, as unrestricted-$\phi$ SSA representations. If we performed the naive code transformation outlined in the last section then our $\phi$-functions would have the proper form, but almost no optimizations would be found across $\phi$ boundaries. Better performance could be achieved by using the fast copy coalescing algorithm [4].

We also note that the reverse transformation can be performed. First, observe that copy operations will never be necessary in a program in unrestricted-$\phi$ SSA form. If we are given a program in standard SSA form, we can perform copy folding to eliminate all

[1]Being *anticipated* is not good enough, since we need to avoid moving such effects over the boundaries of different effects—SML has a precise definition of how a program evaluates.

copies. This may make some of the $\phi$-functions violate the single-variable assumption, and so we now have a program without copies in unrestricted-$\phi$ form. C-SSAPRE can be applied directly to the resulting copy-free SSA form. This transformation allows the discovery of redundancies that were previously masked by the copy. As an example, in Figure (2), our algorithm will detect that `op a` in block L3(a) is fully redundant (with `op x` in L1 and `op y` in L2), and it will find the partial redundancy between the `op` $x$ occurrences in blocks L1 and L4. Standard SSAPRE, however, would only detect partial redundancy in the first case, because the additional redundancy is masked by the copy in block B1. Thus, by designing an algorithm to perform SSAPRE on the MLton IR we are equivalently developing a generalization of SSAPRE that will find additional redundancies. We now address the question of designing such an algorithm.

The SSAPRE algorithm optimizes all occurrences of some expression at once. For example, all occurrences of the expression $a + b$ (ignoring version numbers on $a$ and $b$) are analyzed simultaneously. Thus, we are immediately at a loss for how to apply the algorithm to MLton; we will need an alternative method of constructing a set $E$ of related expression occurrences to optimize.

To find suitable sets $E$, it appears promising to precisely define the properties these sets of expression occurrences must have for SSAPRE to work correctly. For example, all the expressions in $E$ must involve the same operator (and hence the same number of operands). Informally, we might guess it must also be possible to use a single temporary variable to track the value of all expressions in the set $E$.

However, the idea of simply defining a property that all "acceptable" sets of expressions $E$ must have is inadequate; a richer framework is needed. To see this, consider the following example:

```
L(a,b):             [1]
     ... = op a     [2]
     GOTO L(b,a)    [3]
```

This snippet of MLton IR code repeatedly computes `op` $a$, `op` $b$, `op` $a$, etc. Certainly it should be possible to consider removing an expression from a loop, so we should consider lifting `op` $a$ from the loop. And, in a only slightly modified loop, such as:

```
L(a,b):             [1]
     ... = op a     [2]
     GOTO L(a,b)    [3]
```

this hoisting would be perfectly valid. However, in the swapping example, we cannot lift the operation. The key is to realize that the call `L(b,a)` assigns to the operand of [2], and hence prevents any partial redundancy. Our algorithm must detect this situation in the first example, while still allowing code hoisting in the second example. Thus, one cannot consider only a set of expressions and properties of those expressions. The program must be analyzed in a way that takes assignments to operands (including implicit ones in labeled block calls) into account. We now describe a framework for doing this kind of analysis.

PRE can be though of as answering the following question: given a single temporary variable, how can it be used to remove partial redundancy in a program? We turn our attention away from the set of expressions $E$ to be optimized, and focus on the behavior of this temporary. Recall we have fixed a the operator `op` for consideration. We now imagine "coloring" the CFG, where a color corresponds to an ordered list of variable arguments to `op`. For example, $\langle a \rangle$ and $\langle x \rangle$ are colors in Figure (2). We wish to maintain the property that our single temporary variable always tracks the value of `op` on the variables given by the color of the current program point. In order to insure that the temporary holds the correct value, computations must be inserted at certain program points.

Our prepass algorithm assigns such a coloring, and determines program points where computations would have to be inserted if we wished to insure the temporary always tracks the value of the expression on the color. The back edge in the argument-swapping example is such a location. The idea is to pick a coloring of the control flow graph so that at many occurrences of `op`, the expression is colored with the actual operands to `op`. In these cases, `op` is added to the set $E$ of expressions we will optimize. Occurrences of the wrong color are ignored.

# 4 Formalizing the CFG Coloring Pre-pass

We now formalize the CFG coloring notion and show that any coloring immediately yields a set of expressions that can be optimized by SSAPRE safely. In the next section we present an algorithm for finding good colorings that yields strictly stronger optimization than standard SSAPRE.

**Definition 1.** *A* ***Color*** *for an operator* `op` *of arity $k$ is a vector of variables of length $k$. A color is* ***valid*** *at a program point $p$ if the definitions of all variables in its vector reach $p$. If $r$ is a color for an operator of arity $k$, then $r_1, \ldots, r_k$ denote the variables assignd to each operand of* `op` *by $r$.*

For the following definition, we assume a CFG where each instruction has its own basic block. This is without loss of generality, as a CFG can easily be put in this form and then restored to its original basic block form. We use the terms block and node interchangeably in the following discussion.

**Definition 2.** ***CFG Coloring for*** `op`*: A CFG coloring for an operator* `op` *is a mapping from instructions to valid colors, or to the special color* ***uncolored****. A CFG coloring is* ***contiguous*** *if all the program points in each basic block are colored with at most 2 colors, one of which starts at the top and applies to all instructions in order down the block until the second color is reached, and the second color then colors the remainder of the instructions.*

Note that even though we consider colorings as mappings from instructions, contiguous colorings can be represented by storing only two colors per block, one at the top of the block and one at the bottom, and an integer indicating at which instructions the color changes. While this discussion applies to arbitrary colorings, our algorithms only consider the easier to manipulate contiguous colorings.

Since we assume that all instructions have their own basic block, it suffices to insert computations for the temporary on edges. We now define a class of edges where such computations will be necessary.[2]

**Definition 3.** *Let $B$ be a block with color $r$, and $S$ be a successor of $B$ of color $g$. Suppose* `op` *is of arity $k$. Consider each pair of operand variables $(r_i, g_i)$, for $1 \leq i \leq k$. We say the pair is* ***compatible*** *with respect to the edge $(B, P)$ if $r_i$ equals $g_i$, or if $g_i$ is defined by a $\phi$-function in $P$, and the argument to the $\phi$-function in $B$ is $r_i$. We say the edge $(B, P)$ (or the pair of colors) is* ***compatible*** *with respect to the coloring if each pair of operands $(r_i, g_i)$ is compatible with the edge. If an edge is not compatiable with the coloring, then we call it a* $\perp$***-edge****.*

**Definition 4.** *An expression $e$ is* ***accepted*** *by a coloring $C$ if $e$ is an occurrence of operator* `op` *and $e$'s arguments match the color assigned to $e$ by $C$.*

With this definition we can now state a theorem that relates a coloring for `op` to an SSAPRE optimization.

**Theorem 1.** *Let $C$ be a CFG coloring for operator* `op`*, and let $E$ be the set of expressions accepted by $C$. Then, $C$ specifies a program transformation that allows SSAPRE to be run safely on the expressions in $E$.*

**Sketch of Proof:** We sketch a proof of the theorem by outlining the imagined transformation for the unary operator case. Consider introducing a new variable named $a$ into the program. For every edge $(B, P)$ in the CFG marked with bottom, insert a new assignment to $a$ of value of the variable corresponding to the color of $B$. Run the standard SSA $\phi$-insertion and rename steps; once these are complete only one definition of $a$ will reach each $e \in E$. It can be shown that this instance of $a$ will always be equal to $e$'s current operand variable. Thus, for each $e \in E$ we can safely replace $e$ with the instruction `op` $a_i$ for the appropriate version of $a$. Now, all the expressions in $E$ operate on different versions of the same variable, and all the $\phi$ functions for the variable $a$ have as operands different versions of the variable $a$. This transformation preserves program behavior, and now the set of expressions in $E$ can be optimized by SSAPRE. The case for multiple operands is similar. $\square$

We call the algorithm that first finds a coloring and then applies SSAPRE to the implied transformation the C-SSAPRE algorithm. The C is for copyless.

The optimization of the expressions in $E$ can be performed without actually implementing the full code transformation. Rather, the standard SSAPRE algorithm is run on the expressions of $E$, with the modification that its Rename step respects the $\perp$-edges already inserted. The $\Phi$-insertion step is also slightly different, because some of the insertions are done ahead of time during the construction of the coloring. The details of these changes for our implementation are given in Sections (5) and (6).

The following theorem indicates that by finding a good coloring we can do at least as well as standard PRE:

[2] These computations are necessary if the invariant that the temporary tracks the appropriate value at all colored program points. This tracking property will actually only be needed at expressions we wish to remove via PRE, so not all these computations will actually be inserted. These edges correspond directly to a certain class of $\Phi$-operands that are set to $\perp$ in the SSAPRE algorithm.

**Theorem 2.** *Consider optimizing a program in standard SSA form for the operator* `op` *of arity $k$ on variables $a^1, a^2, \ldots, a^k$. For any program in standard SSA form, if the variable versions are "forgotten" by arbitrarily renaming each versioned variable $a^i_j$, then there is a coloring that induces a transformation that allows the same optimizations to take place as if SSAPRE had been run on the program before variable versions were lost.*

**Sketch of Proof:** The coloring is the one that assigns to program point $p$ a color $c^p$ where $c_i$ is the new variable name associated with the version of $a^i$ that was active at $p$ before renaming occured. All edges between different colors in this coloring will be compatible, and so no edges are marked $\perp$. Thus, the SSAPRE algorithm will proceed exactly as it would on the original program before renaming. $\square$

**Definition 5.** *A coloring is* ***minimal*** *if every instruction that it colors is on some path $P$ through colored instructions between expressions in $E$ that does not cross a $\perp$ edge, where $P$ has the property that the direction of its arcs change at most once.*

The above definition is an attempt to define a property of colorings that do not color any "extra" instructions. That is, we would like colorings where instructions are only colored if they either improve availability or downsafety for expressions in $E$. We have not yet convinced ourselves that the above definition is the correct one, but it appears to be a good heuristic at least. Building up colorings so that they are minimal should help construct good colorings because it leaves maximum freedom to color nodes in other ways to obtain additional redundancies or downsafety. The algorithm in Appendix (9) tries to build such colorings, while the algorithm in the next section may color many blocks to no effect.

# 5 A Simple, Fast CFG Coloring Algorithm

This section describes a simple coloring algorithm that leads to a C-SSAPRE implementation at least as good as SSAPRE. This algorithm works on blocks containing multiple instructions, as opposed to the previous section where we assumed one instruction per block for convienence. Our simple algorithm for coloring starts from a single seed expression. It then greedily scans up the control flow graph, marking all instructions it reaches so that they track the color of the seed expression (variables of the seed expression). If a $\phi$-function definining one of the variables in the color is reached, coloring proceeds to the predessors of the $\phi$-function(s) with new colors chosen to make the edges compatible. The algorithm continues upwards until a real (non-$\phi$) definition is reached for some variable of the color on each path up from the seed's block,[3] or until all blocks that can be colored have been. The algorithm marks edges with $\perp$ when appropriate as it proceeds.

Then, for every block we scanned in this first pass up (`fromBottomScan` in the pseudocode below), we try to scan down through blocks that have not yet been scanned (`fromTopScan`). The goal is to add more expressions in order to increase down-safety. Once this step is complete, some new blocks may have been colored that were not colored before. These blocks are now scanned in the upwards direction, looking for more partial redundancies.

[3] It can be shown that there can only be one such blocking definition

These steps alternate until no more blocks are added. It is also possible to run the algorithm so that only two scans are made, one initial set of fromBottomScan scans, and one set of fromTopScan scans to increase downsaftey. Only expressions added in the fromBottomScans are considered to have been optimized.

The driver function given below alternates between the upward scans and the downward scans, until the queues for both scan types are empty:

```
simpleColor(Expr seed)
    UpScanQueue.add(seed.block, seed.opnds, true)
    while(!upScanQueue.empty())
        while(!upScanQueue.empty())
            (B, S, color) = upScanBlock.pop()
            fromBottomScan(B, S, color)
        end while
        while(!downScanQueue.empty())
            (B, P, color) = upScanBlock.pop()
            fromTopScan(B, P, color)
        end while
    end while
end driver
```

The code for fromBottomScan and fromTopScan are similar. Both methods first check if their color has already been set; if it has, they only check the edge that caused them to be scanned to see if it needs to be marked with $\perp$. The method `isCompatibleEdge(B,S)` compares the bottom color of $B$ to the top color of $S$ , and returns true if and only if they are compatible according to the definition of given in the previous section.

```
fromBottomScan(Block B, Succ S, Color color)
    if(B.botColor == SET)
        // no work to do, just test the edge
        if(isCompatibleEdge(B, S))
            // edge is OK
        else
            // edge has hallucinated assignment
            S.addPhi()
            S.phi.setPhiArg(B, Bottom)
        end if
        return
    end if
    B.botColor = color
    enqueueSuccs(B, color)
    // scan the relevant instructions
    instrList = instrListFromBottom(B, color)
    for each e in instrList
        if(isExpression(e) AND (e.opnds == color))
             add e to E
        else if (e is definition of some opnd in color)
             B.splitPoint = e
             return
        end if
    end
    // if we reach this point we didn't split,
    // so continue scanning up our predecessors
    enqueuePreds(B, color)
    B.topColor = color
end fromBottomScan
```

```
fromTopScan(Block B, Pred P, Color color)
    if(B.topColor == SET)
        if(! isCompatibleEdge(P, B))
            B.addPhi()
            B.phi.setPhiArg(P, Bottom)
        end if
        return
    end if
    B.topColor = color
    enqueuePreds(B, color)
    instrList = instrListFromTop(B, color)
    for each e in instrlist
        if(isExpression(e) AND (e.opnds == color))
            add e to E
        else if (e == B.splitPoint)
            return
        end if
    end
    enqueueSuccs(B, color)
end
```

Two methods handle the enqueuing of predecessors and successors to the queues that manage the traversal of the algorithm and expansion of the coloring. Both are fairly straightforward. Note that the order in which operands are considered is chosen by a heuristic, as this order may have a significant impact on the quality of the coloring obtained; our current implementation, however, uses an arbitrary order. When enqueuing predecessors we may also insert some $\Phi$ functions, simplifying the $\Phi$-insertion step later; see the next section for details.

The upColor(color, B, P) method translates the current color into a compatible color based on any $\phi$-functions passed through while going up the edge; it makes no change to color if no such $\phi$-functions are present. When going down an edge past a $\phi$, a choice must be made: for each $\phi$-function on a variable argument to `op`, that same variable can continue to be tracked (this is the current implementation), or the variable resulting from the $\phi$ function can be tracked. For example, in Figure (2), if we are scanning down from $L1$ with color $\{x\}$, the algorithm can choose to continue scanning down for $\{x\}$ from the top of $L3$ (so that it will find and add [4] to the set), or it may scan down for color $\{a\}$, in which case it will find [3] and add it to $E$. Each of these choices is "right" in this case, and will result in different partial redundancy being eliminated.[4] In order to prove that the optimization performed by C-SSAPRE is at least as strong as standard SSAPRE we need to use the heuristic that tracks the newly defined variable rather than the original. We call this the $\phi$-following down coloring heuristic.

[4]It is somewhat better to defer the call to `downColorHeuristic` until (S,B,color) is popped off the stack, because if at that point S has already been colored, our only choice is wether or not the color of S is compatible with the color of B. Only if S is not colored do we need to apply the heursitic. However, to maintain symmetry and simplicity in the code we show the naive implementation.

```
enqueuePreds(Block B, Color color)
        predList = heuristicPickPredOrder(B)
        for each P in predList
                newColor = upColor(color, B, P)
                 if(newColor != color) // passed through a phi
                        B.addPhi()
                 end if
                 upScanQueue.add(P, B, newColor)
        end for
end enqueuePreds
enqueueSuccs(Block B, Color color)
    val succs = heuristicPickSuccOrder(B)
    for each S in succs
        newColor = downColorHeuristic(color, B, S)
        downScanQueue.add(S, B, newColor)
    end for
end enqueueSuccs
```

This algorithm is fairly straightforward to implement, and does at least as well as SS-APRE:

**Theorem 3.** *Let $P$ be a procedure in standard SSA form that contains some occurrences of the arity $k$ operator* `op` *on variables $a^1, a^2, \ldots, a^k$. Let $e$ be particular expression occurrence of* `op` *on these variables. Then, running C-SSAPRE with the* `simple` *coloring algorithm using the $\phi$-following down coloring heuristic on $P$ will optimize $e$ in the same manner that standard SSAPRE would.*

**Sketch of Proof:** The live ranges of the different versions of $a^i$ do not overlap, and so as the the coloring algorithm proceeds it will find a coloring that is a subset of the coloring constructed in the proof of Theorem (2). □

While this is a strong property, the colorings produced by `simple` are naive in the sense that they are not minimal. Appendex (9) outlines a coloring algorithm that produces a minimal coloring that also has the property that it eliminates at least as much redundancy as SSAPRE. It is straighforward to construct examples that show that the colorings it constructs lead to strictly more optimization than the `simple` algorithm.

# 6 The SSAPRE Algorithm

We now described the changes to the standard SSAPRE algorithm so that it can be combined with a coloring pre-pass to form C-SSAPRE, without explicitly performing the program transformation specified by the coloring.

- **$\Phi$-insert** is accelerated by the fact that we insert $\Phi$'s corresponding to $\phi$'s for `op` variables during the coloring phase. In particular, we need to insert a $\Phi$ for this reason if and only if on an enqueuePreds call one of the variables passes through a

$\phi$-function. Thus, in the $\Phi$-insertion phase we need only insert $\Phi$s at blocks in the iterated dominance frontier of blocks containing expressions in $E$; we avoid the need for the `set_var_phis` [7, pg. 661] step.

- **Rename** is implemented in a fashion similar to the naive algorithm described in [7, pg. 641]. However, in that version variable renaming stacks are necessary for the variables in the expression being optimized, as well as a renaming stack for the $\Phi$ functions. We can do away with the renaming stack for variables, because when we assign a class to a $\Phi$-operand we can look at the bottom color of the block instead. This gives us a sparse renaming algorithm with an easy implementation; however, it is not really a win over the algorithm in [7] because we are really just taking advantage of the information collected in the dense coloring pass.

- **Optimization Scheduling** Since we cannot partition the expressions into our programs into convenient sets based on the underlying variables used $(a + b)$ as in the original algorithm, it is unclear how best to schedule optimization passes generated from seeds using the coloring algorithm presented earlier. Currently, our implementation follows a naive approach whereby it continues trying seeds for a given `op` until all possible expressions have occurred in at least one optimization set $E$. This runs relatively few optimization sets; a much slower but more thorough algorithm might optimize the sets achieved by using each potentially optimizable expression in the procedure as a seed. These methods are naive, and better seed selection and organization of the coloring passes will probably be required for practical implementation.

# 7 Open Problems

Several important questions have been left unanswered by our work so far.

- Speed Improvements: Can the coloring phase be combined with the rest of the SSAPRE algorithm to obtain a more efficient solution? Can the coloring phase be re-designed so it can compute a set of colorings for a given procedure that express all the optimizations that should be done, rather than performing one pass for each optimization? (This would solve the scheduling problem).

- Find an efficient coloring algorithm such that if a coloring exists that allows an expression $e$ to be removed by PRE, then the algorithm will find such a coloring. Classify the set of expressions that can be removed using PRE with a single temporary.

- There are programs where an expression can be moved via PRE only if multiple temporaries are used. In the (a,b)-swapping example, the expression in the loop can be safely removed with two temporaries.[5]

[5]Note that to make the transformation down-safe two new blocks must be inserted, a landing-pad that executes once if the loop executes at least once, and another landing-pad that executes once if the loop executes at least twice.

With this in mind, extend PRE so that it finds redundencies that require the use of two temporary variables, or perhaps even $k$ temporary variables. This might also lead to algorithms with improved performance, possibly allowing a single PRE optimization pass to handle all the redundancy for a single operator.

- We have not been able to construct an example where two temporaries are required to remove an expression $e$ when it is possible to remove $e$ using partial redundancy with some expression other than $e$. That is, it appears to be the self-referential nature of the (a,b)-swapping example that makes the use of two temporaries necessary. Note that there is a coloring that removes `op` $a$ from

```
L(a,b):
    ... = op a
    ... = op b
    GOTO L(b,a)
```

  using redundancy with `op` $b$.

  Prove that if $e$ can be removed by using redundancy with some other expression, then there is a coloring that allows $e$ to be removed. Find an algorithm that always discovers such a coloring if it exists.

# 8 Discussion and Conclusions

We know of only two SSAPRE implementations that are readily available.[6] There is implementation in C for GCC in 3000 lines of fairly dense code [3], and a 4000-line implementation in Java as part of the BLOAT project [1]. The authors of both versions comment on the complexity of the algorithm and difficulty of the implementation. We point this out only to put our results in context. Our ML implementation, including our new coloring phase, optimization scheduling code, and a critical edge removal pass total around 2000 lines of code, a testament to the representative power of the language. Our code compiles and runs, but we have not yet had time to run interesting experiments; developing coloring algorithms took considerably more time than expected. Also, we expect that initial performance gains will not be outstanding, because we do not make a pass to transform loops to insert down-safe landing pads for loop invariant code motion. We intend to test, debug, and measure our code for potential inclusion into the MLton distribution.

We have described a substantial extension to the standard SSAPRE algorithm. We have explained the algorithm as it applies to the unrestrected $\phi$-functions from MLton's IR, and we have shown that by copy-folding a standard SSA representation our algorithm can be applied to detect additonal optimizations across copies. We hope to demonstrate that these additional optimizations have practical significance in the near future.

[6] We are unaware of the size and complexity of the original SGI implementation

# References

[1] Bloat. http://www.cs.purdue.edu/s3/projects/bloat/.

[2] The mlton compiler. http://mlton.org/.

[3] Ssa for trees. http://people.redhat.com/dnovillo/tree-ssa-doc/html/tree-ssa-pre_8c-source.html.

[4] Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast copy coalescing and live-range identification. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 25–32. ACM Press, 2002.

[5] Fred C. Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 273–286, 1997.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[7] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, 1999.

[8] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT, August 1990.

# 9 Appendix: A Better Coloring Algorithm

We describe a coloring algorithm that is *minimal* in the sense of Section 3.

The algorithm tries to find a coloring that is good for removing a particular seed expression.

First, we add expressions in an up-scan phase that may make the seed partially redundant. As we scan up for a particular color (via scanUpForColor), we will encounter a unique stopBlock where one of the variables in the current color is defined. If the definition was by a $\phi$-function, we then recursively process the predecessors of this stopBlock, using new colors based on the $\phi$-function. We then "finalize" the colorings that are actually on interesting paths by scanning down from expressions and from the $\phi$ if it was "interesting." Note that this `finalizeColorDown` is different from the `scanDownForColor` that finds expressions that help to prove down safety.

The `scanDownForColor` portion of the algorithm is very similar to the `scanUpForColor`, and so pseudocode is ommited for that routine.

Each `scanUpForColor` call uses a ScanData structure to store certain information.

```
minimalColor(Expr seed)
    scanUpForColor(seed.block, seed.opnds)
    for each colored block B
        scanDownForColor(B)
    end for
end
struct ScanData
    stopBlock
    colorSet
    found
end
```

The up scanning phase is performed via a nested depth-first search. The outer search is controlled by scanUpForColor. A single call to this method does a scan up from the given block for a given color. This scan is controlled by the `innerScanUp` method. After the inner scan is complete, and any additional upScans for our predecessors through $\phi$-functions are completed, there is enough information to determine what edges tentatively colored (by `maybe clr`) are actually interesting. This finalization step is done via the `finalizeColorDown` procedure. A color is `final` if it is not a `maybe` color. We assume all values are initialized to `null` or `false` as appropriate.

```
boolean scanUpForColor(Block start, Color clr)
    assert start.botColor == UNCOLORED
    sd = new scanData
    innerScanUp(start, null, scanData)
    boolean ia = false // interesting above
    if(sd.stopBlock.splitPoint == null) // stopped by phi
        for each P in sd.stopBlock.predecessors
            newColor = up Color(clr, B, P)
            ia = ia OR scanUpForColor(P, newColor)
        end for
        if (ia)
            sd.colorSet.add((stopBlock, TOP_OF_BLOCK))
        end if
    end if
    for each (B,p) in sd.colorSet
        finalizeColorDown(B)
    end for
    clear all visited flags on edges we set
    remove all non-finalized phi's
    return ia || sd.found
end scanUpForColor
```

```
innerScanUp(Block B, Block S, Color clr, ScanData sd)
    if(B.scannedUpFor(clr)) return // already scanned this node
    if(B.botcolor != UNCOLORED)
        if((B.botcolor == clr) OR (B.botcolor == maybe clr))
            edge(B,S).visited = true
        else  // wrong color
            S.addMaybePhi() // finalized to real Phi if we color this node
            S.phi.argFor(B) = bottom
        end if
        return
    end if
    B.botColor = maybe clr
    B.scannedUpFor(clr) = true
    instr:ist = instrListFromBottom(B, clr)
    for each e in instrList
        if(isExpression(e) && e.opnds == clr)
            add e to E
            sd.found = true
            sd.colorSet.add((B,e))
        else if (e defines a variable in color)
            B.splitPoint = e
            assert (stopBlock == null)
            sd.stopBlock = B
            return
        end if
    end
        assert B.splitPoint == null
    B.topColor = maybe clr
    if (some variable in clr is defined by φ at start of this block)
       assert (stopBlock == null)
       sd.stopBlock = B
       return
    else
       for each P in B.predecessors
           scanUp(P, B, clr)
       end for
    end if
    return
end innerScanUp
```

```
finalizeColorDown(Block B, ProgramPointInBlock p, Color clr)
    if(p == TOP_OF_BLOCK)
        if(B.topColor == clr)   return // already colorDown'ed
        assert B.topColor == maybe clr
        B.topColor = clr
        if(B.phi == maybe phi)  // finalize phi if we have one
            B.phi.finalize()
        if(B.splitPoint != null)
            assert (b.botColor != clr AND b.botColor.isFinal())
            return
        end if
    else if(isExprOccurrence(p))
        assert B.splitPoint == p
    end if
    for each S in B.successors
        if(edge(B,S).visited)
            colorDown(S, TOP_OF_BLOCK, clr)
        end if
    end for
end colorDown

scanDownForColor(Block B)
    ...
end scanDownForColor
```