15-745: Graduate Compilers Project
SSAPRE in MLton (Notes for Milestone)

Tom Murphy VII, Brendan McMahan

Due: 14 Apr 2003

1 Introduction

MLton [1] is a high performance whole-program compiler for Standard ML [2]. One of MLton's intermediate representations is a form of static single assignment control flow graphs ["SSA"; 5]. Though much care is taken to transform the functional source language into an efficient form with explicit loops (much like a C or Java compiler might produce), 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.

Our project seeks to develop a version of Partial Redundancy Elimination [6] that is appropriate for MLton's modified SSA IL, and implement it in the compiler.

Partial Redundancy Elimination 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 [3, 4] algorithm implements Partial Redundancy Elimination for SSA form. However, the SSA form that they assume is not directly compatible with MLton's IR. Therefore, although we intend to use their algorithm, we need to develop a pre-pass that meets SSAPRE's invariants, or modify the SSAPRE algorithm to not require those invariants.

This milestone report proceeds as follows. First, we describe the MLton SSA intermediate representation. Next, we discuss some progress on fitting the SSAPRE algorithm to MLton's IR. Then, we review the SSAPRE algorithm with notes regarding our implementation strategy.

2 The MLton SSA IR

MLton's low-level intermediate representation is a Static Single Assignment [5] Control Flow Graph. The representation is generally like the one you might find in a modern SSA-based C compiler, but has some differences that will impact the way that we implement PRE. This section describes these differences.

- **Labeled GOTOs instead of ϕ-functions**. Blocks in the MLton IR may take arguments. Any jump to such a block must pass along arguments, which are assigned into the argument temporaries. Though this makes a procedure look like a series of mutually-recursive functions, these variables are not lexically scoped. This is really just an alternate notation for ϕ-functions; one can translate between this and standard SSA fairly easily and without any loss of information. *XXX example for comparison?*
- **No 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 will have important repercussions for our PRE pass. The next section explores this in more detail.
- **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, etc. Because these are effect-free (but expensive) operations, the potential benefit from PRE on them is large!
- **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.

Despite our deemphasis of arithmetic expressions, we still use expressions such as $a + b$ in examples in order to conserve keystrokes and remain faithful to the literature.

3 Constructing Expression Classes

In this section we discuss the problem of adapting the SSAPRE algorithm to MLton's particular form of SSA IR.

The MLton SSA form can be thought of as the following generalization of standard SSA: Some number of ϕ functions may appear at the beginning of each basic block. Each of these ϕ functions assigns to one variable, and has one parameter for each predecessor of the block. In standard SSA form, each ϕ only operates on different "versions" of the same variable; the MLton IR does not have this restriction.

For example, we allow the following:

```
variables x,y defined earlier
B1:
   a = x
   b = 5
   goto B3
B2:
   goto B3
B3:
   w = φ(a, y)
   z = φ(b, x)
   ... = w + z
```

[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.

This is still an SSA representation because each variable has exactly one definition point that dominates all of its uses. In "standard" SSA the ϕ functions have the property that all of their arguments are different versions of the the same variable, and so on a given program path only one of these operands holds a value. This is not true for MLton's IR. The actual MLton IR represents its ϕ functions by giving blocks arguments. Thus, the above code in the MLton IR would look like this:

```
B1:
   goto B3(x,5)

B2:
   goto B3(y,x)

B3(w, z):
   = w + z
```

XXX This is a rather poor example, something cleaner is probably better.

Notice that the calls `goto B3(a,b)` perform implicit copies.

The SSAPRE algorithm assumes that ϕ functions have this only-one-active-version property, and so to implement the algorithm in MLton some modifications are necessary.

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. For example, if we fix the expression $a+b$ and scan some program we might find the set $E = \{a_1+b_2, a_1+b_3, a_2+b_1, a_4+b_5, \ldots\}$. We will need an alternative method of constructing these sets E for MLton, because MLton's IR does not work with variable versions; each name is unique. Further, our entries in E will be unique occurrences of an expression in the program; that is E may contain $a_1 + b_1$ when it occurs at one point in a program, but not when it occurs at another point in the program.

To find suitable sets E, we must precisely define the properties these sets of expression occurrences must have for SSAPRE to work correctly. Clearly, all the expressions in E must involve the same operator (and hence the same number of operands). The critical additional property is this: it must be possible to use a single temporary variable to track the value of all expressions in the set E. We call this the SINGLE-VALUE property for a set of expressions. This property can be maintained if the following property holds for all the operands: If for all operand positions i in E, the value of all the variables occurring as the ith operand can be tracked by a single variable, then E has the SINGLE-VALUE property. We call this property the SINGLE-VALUE property for variables. This is clearly true in the "standard" SSA case, because all the variables that occur as a given operand in E are different versions of the same variable, and thus can only have one value at a time.

(XXX I'm not sure this is "clearly true", maybe it's worth explaining why it is true in the final paper – Tom)

Consider the following example of a loop that computes $x - y$, then $y - x$, then $x - y$, etc.

```
L1:
   x = ...
   y = ...
   goto L2(x,y)
L2(a, b):
   ... = a - b
   L2(b, a)
```

Note that both a and b can be assigned to both the first and second arguments to the block.

We can easily transform this into the somewhat more verbose "standard" SSA IR:

```
L1:
   x =
   y =
   a1 = x
   b1 = y
   goto L2
L2:
   a2 = φ(a1, a3)
   b2 = φ(b1, b3)
   ... = a2 - b2                [1]
   t = a2
   a3 = b2                      [2]
   b3 = t                       [3]
   goto L2
```

In the "standard" IR code, the SSAPRE algorithm realizes that $a - b$ is not really partially available at point [1], because of the assignments to a and b at points [2] and [3]. However, it is unclear how to recognize this from the simpler MLton IR representation.

BEGIN: some weak speculative discussion. but it points out that constructing E based on textually identical expressions actually doesn't work

There is a problem here. In the MLton IR for the above problem, the PRE algorithm should not apply, and yet one would think that if E contains the single expression $a-b$ then clearly a and b have the SINGLE-VALUE property, and so $a-b$ should have the SINGLE-VALUE property. Perhaps this isn't really a problem, and we simply need to explicitly make sure SSAPRE for MLton understands that a call like $L2(b2, a2)$ is really an assignment to $a2$ and $b2$; however, in the case that the loop was always computing $x - y$ (that is, the looping call is $L2(a2, b2)$, we need to realize that there isn't really a copy so we can safely do PRE; otherwise, our algorithm will make almost no optimizations.

However, I think it may also be possible to simply do some initial analysis and conclude that the set of equations $E = \{a2 - b2\}$ is not suitable for PRE transformation. If we rewrite the MLton IR using ϕs we have:

```
L1:
   x =
   y =
   goto L2
L2:
   a = φ(x, b)
   b = φ(y, a)
   ... = a - b
   goto L2
```

The goal is to examine the phi function and conclude that in fact the operands do not have the SINGLE-VALUE (defining the SINGLE-VALUE property for variables may not be the right definition). However, we cannot detect a problem directly from the ϕ functions as given above. However, what we may really want is a kind of "closure" of ϕ functions for the current loop: that is, we want a ϕ function that only references variables that are defined outside the loop:

```
a = φ(x,y)
b = φ(y,x)
```

The first phi function implies that x and y need to have the SINGLE-VALUE property; but the fact that x and y occur as the first argument to both ϕ functions implies that x and y could have different values (as they in fact do).

We need to formalize this intuition and try to develop an efficient algorithm that can create sets suitable for optimization by the standard SSAPRE algorithm. It may also be necessary to handle in some way the implicit copies performed by the basic block "calls" such as $L_2(b, a)$.

END: weak stuff. I think this next idea might be pretty close to what we need

Conjecture: Fix a set of expressions E. To determine if E is valid, we perform the following analysis on each basic block: For each operand position to any expression in E in the basic block, and (separately) for each control flow path into the block, determine the non-ϕ definition points that reach the operand (this is the same as the reaching definitions set). If a single operand position is reachable by multiple such def-points on a single control flow path into the block, then one or more of the equations in E must be removed before it is valid.

Example of what I mean by operand position: the first operand, or the second operand. So in the example below, in block 4, we're interested in all the def points reaching a, x on a particular control flow path in, and all the operand reaching b,y on some control flow path in.

This property seems to work for the examples I've looked at, and I think it should be possible to prove something about it.

It also applies to the following example, detecting PRE optimizations that standard SSAPRE would not:

```
L1:
  b = ...
  x = ...
  y = ...
  bgte b L2 L3
L2:
  ... = x + y       [1]
  goto L4(x, y)
L3:
  c = ...
  ... = x + c       [2]
  goto L4(x, c)
L4(a,b):
  ... = a + b       [3]
  ... = x + y       [4]
```

This example can be converted to standard SSA form, and PRE on that form will not find total redundancy at point [3] (the expression is available from points [1] and [2]). Note that expression [4] is partial available (from [1]). Our algorithm should find both of these optimizations. First, we will consider optimizing with $E = \{[1], [2], [3], [4]\}$. However, by the above criteria, we note that when L4 is entered from L3, the second operand may to an expression in E could be either c or y. Thus, the second operand cannot be tracked by a single variable. When we consider the expressions separately, however, we realize that we can optimize with $\{[1], [2], [3]\}$ is a block of expressions, and $\{[1], [4]\}$ as a block of expressions, thus getting both optimizations. Notice that this requires inserting two temporaries and passing both as arguments to L4.

(XXX is it possible for the previous thing to then give a proof by establishing equivalence classes for each of the arguments (ie, sets of variables that may flow into there) so that we can use those as the "versions" of that variable for the remainder of the discussion? I will assume it is...)

4 SSAPRE Algorithm

Once we have selected a set of expressions to be considered for PRE, we can run the SSAPRE algorithm more-or-less as it is originally designed.

Data Structures. We do not expect to modify the CFG directly until the final step. This is because we do not want to change the IR's abstract data type (for a number of reasons). Instead, we will represent insertions of Φ-functions (below) as an external data structure—a multiset of blocks with flags and properties (see the next section). We also need to set properties of expression instances in our optimization set, which can be handled with a similar external data structure. Aside from a few local tables needed in some steps (Finalize_1), these are all the data structures we need.

We do need to compute some information about the CFG, like the iterated dominance frontier. These are mentioned in the summary below.

Further Preparation. Before we begin, we need to remove critical edges by inserting empty blocks along them. This is fairly straightforward. Since optimizing a set of expression occurrences does not alter the control flow or block layout, we can do this a single time before our pass.

Φ-insertion. SSAPRE uses Φ-functions to track the value of the expression being optimized with a temporary. These are really just ϕ-functions, so we will use MLton's counterpart to represent them: blocks

and GOTOs with arguments.

In this phase, Φ functions are inserted speculatively into the program. These mark the location of real ϕ functions that may eventually be placed.

Φ functions are inserted for two reasons. First, when an expression appears, Φs are inserted at its iterated dominance frontier, just as the definition of a temporary when converting to SSA form induces the insertion of ϕs at its IDF. Second, Φ functions are inserted when a variable in the expression is defined by a ϕ function, since this induces an alteration of the expression's value.

This step is fairly simple. MLton already has code for computing the IDF in the module that converts code to minimal SSA form. We can probably make use of that code; otherwise, it is not difficult to program given the dominator tree (which is already available).

(XXX optimization for not inserting Phi when expression is not anticipated.)

Rename. The rename step assigns classes to each expression instance. Each instance in a class has the same value.

This pass should not be difficult. We need only make a preorder traversal of the dominator tree keeping track of the "current version" of each expression operand (changing when we encounter a phi function) and a "current class" of the expression. Following the rules in the SSAPRE paper (pp 15), we assign classes to each expression instance during the traversal. (These classes will eventually be the SSA variables used to track the expression's value.)

Down-Safety. Down-safety needs to only be computed at each Φ node. Again, this is an easy step. We begin by marking any Φ node that can reach exit without crossing an occurrence of an expression in its redundancy class. Then, we propagate this Down-Dangerousness according to the rules on page 18.

It may be possible to relax the down-safety test since our expressions are mostly side-effect-free. This may make the code worse in some cases (increasing the number of computations along some paths), of course, but it might be interesting to see what performance impact this has.

WillBeAvail. WillBeAvail actually does redundancy elimination, by marking locations where the expression's value will be available as a result of insertions to be performed in a later step.

WillBeAvail's goal is to set the `will_be_avail` flag for Φs where the expression will be made fully available by inserting in the immediate predecessors. First this means setting the `can_be_available` flag for Φs.

Begin with cba(Φ) set to true for every Φ. Mark any cba(Φ) to false wherever `down_safe` = false and at least one argument is ⊥. Then, propagate this to any Φ that is reachable without passing through a real occurrence.

Then, we compute the `later` flag. This starts at true wherever cba(Φ) is true. Then, wherever a real occurrence is found, we propagate later=false to the places beyond which insertions cannot be moved. `later` serves the purpose of delaying insertion until the last (optimal) moment in order to reduce register pressure.

Pseudocode for computing these is given on page 21. Again, these are just recursive walks of the dominator tree and (expression) use-def chains, which set and read properties of Φ nodes. This should not pose any implementation trouble.

The `will_be_avail` flag is set any place that `can_be_avail` is true and `later` is false.

Finalize$_1$. This is not the final step.

Here we decide which Φ functions will become real ϕ functions (block arguments) tracking the value of the expression. We set the `reload` flag to determine whether real occurrences of the expression will be actually computed or reloaded from the temporary. The algorithm to do this (pp 25) is fairly unenlightening, but no new data structures are needed except for a partial map `Avail_def[]` that keeps track of the available definition point for each redundancy class as the algorithm traverses the dominator tree.

Finalize$_2$. In this step, we set a flag `save` on real occurrences so that we know to save the computation into a temporary. This is fairly trivial.

I believe we can ignore most of this step as presented in the paper (pp 26: "extraneous Φ removal"), because MLton has a pass to restore minimal SSA form to a CFG. It is probably more efficient to do it here, but at least our first implementation attempt will be simpler.

CodeMotion. After gathering all of the information in our auxiliary data structure, we can finally modify the code to reflect the results of PRE.

The way this is done is highly dependent on the actual representation being used. This is where we will translate the Φ functions in our multiset into real block arguments, insert the computations marked to be inserted, and modify the code to save to and load from temporaries. I do not think this will be very tricky at all.

Finishing up. Of course, we repeat the above steps for each set of expressions we decide to optimize. When we're done, MLton has a set of standard cleanup routines that we will run. These will, for instance, remove the empty blocks that we inserted on critical edges if they did not have any computations inserted into them.

5 Index of flags/properties

The SSAPRE algorithm involves the use of many flags and properties. This lists their meaning for reference.

flag	applies to	meaning
`down_safe`	Φ-nodes	Down-safe if the redundancy class can be safely computed there (as an argument to this Φ function in an incoming block)
`will_be_avail`	Φ-nodes	Marks places where **Finalize** step will make the expression fully available. (= `can_be_available` ∧ ¬ `later`)
`can_be_avail`	Φ-nodes	False of no down-safe placement of computations can make the expression available here.
`later`	Φ-nodes	Indicates those `can_be_available` nodes that we will not perform insertions for, because we will make the insertion later.
`insert`	Φ-nodes	Marks `will_be_avail` nodes that we will actually insert for.
`reload`	real occurrences	True if the expression should be loaded from the temporary that tracks its value.
`save`	real occurrences	True if the expression should be saved to the temporary that tracks its value.

property	domain	meaning
`def(v)`	Φ operands	Gives the definition point (Φ node or real occurrence) of an operand to a Φ function.
`Avail_def[x]`	redundancy classes	Gives the def site (real occurrence or Φ with `will_be_avail` true that computes the value of the redundancy class, or ⊥.

6 Bibliography

1 *The MLton Compiler*. `http://mlton.org/`

2 *The Definition of Standard ML (Revised).* Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. MIT Press, Cambridge, Massachusetts, 1997

3 *A New Algorithm for Partial Redundancy Elimination based on SSA Form.* Fred C. Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, Peng Tu. SIGPLAN 1997

4 *Partial Redundancy Elimination in SSA Form.* Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, Fred Chow. TOPLAS 3.21, 1999

5 *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph.* Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck. TOPLAS 1991

6 *Global Optimization by Suppression of Partial Redundancies.* E. Morel and C. Renvoise. CACM 22 1979.