Partial Redundancy Elimination (PRE) Optimization For MLton
Srinath Kanna Dhandapani | Advisor: Dr. Matthew Fluet
Rochester Institute of Technology

Introduction
- Partially redundant expressions are those expressions that are redundant through few of the control paths but not from all.
- The implementation of GVN-PRE should eliminate such redundancies along with performing the equivalence of both Common Subexpression Elimination and Loop Invariant Code Motion.

Motivation
- MLton being a whole program optimization compiler it was really exciting to try this algorithm and check its performance.
- The main advantage of this implementation is the benefit of using the same approach to eliminate large data structures like tuple and constructors.

Method
1. **BuildSets Phase 1**
   - This Phase populates the Global Table by going through each of the blocks in the top-down traversal of the dominator tree along with storing information related to each block by keeping track of side effect statements, temporaries, expressions, phi’s and available expressions.

2. **BuildSets Phase 2**
   - Finds the anticipated expressions to keep track of the expressions that are partially redundant from the merging blocks.

3. **Insert**
   - Inserts the expressions in the missing control paths by creating new expressions and return the values through goto transfers. And finally, catch the results as arguments from the merging blocks.

4. **Eliminate**
   - The code after the insert phase can have a lot of duplicate code having repeated copy of values. These statements can be directly eliminated through finding the leaders to all the expressions from the Global Table.

Implementation Variations
- Expressions like Tuple, Constructors, and Select can also be removed through GVN-PRE as they are immutable in Standard ML.
- This removes the register pressure while creating large data structures that are redundant.
- New statement creations are controlled based on operands knowledge in each block with simple incorporation of Args count.
- One time creation of optimized program after the convergence of insert phase avoids creation of new SSA tree after each insertion.
- Standard ML has many kinds of transfers which result in blocks having many input and output control flows, this is handled by creating landing pads with Break Critical Edges pass.
- Optimization additions from Common Subexpression Elimination including array length handling and canonizing expressions to translate expressions based on the operands available in registers through most recent loads.

Optimization Comparison

Results

Conclusion
- Increase in size of the binary by a small fraction is observed with the introduction of statements.
- The run time on benchmark tests have mixed results.
- The poor performance of GVN-PRE at times account to various reasons of having increased register pressure and lengthening control path.
- High compile time for large programs due to convergence as the halting criteria for BuildSets 2 & insert phase

Future Work
- GVN-PRE can be extended to LEPRE to decrease register pressure.
- Introducing common landing pads if more than one control path needs insertion to have optimized insertions.
- Known case improvements involving dead code elimination and duplicate argument elimination.

References