Introduction to Optimization

Muntazir Fadhel

[1]Department of Computing and Software
McMaster University

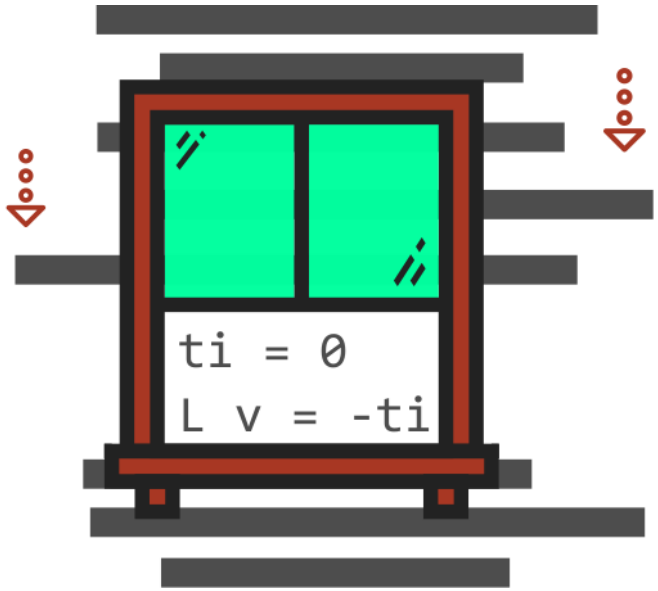
December 6, 2017

Overview

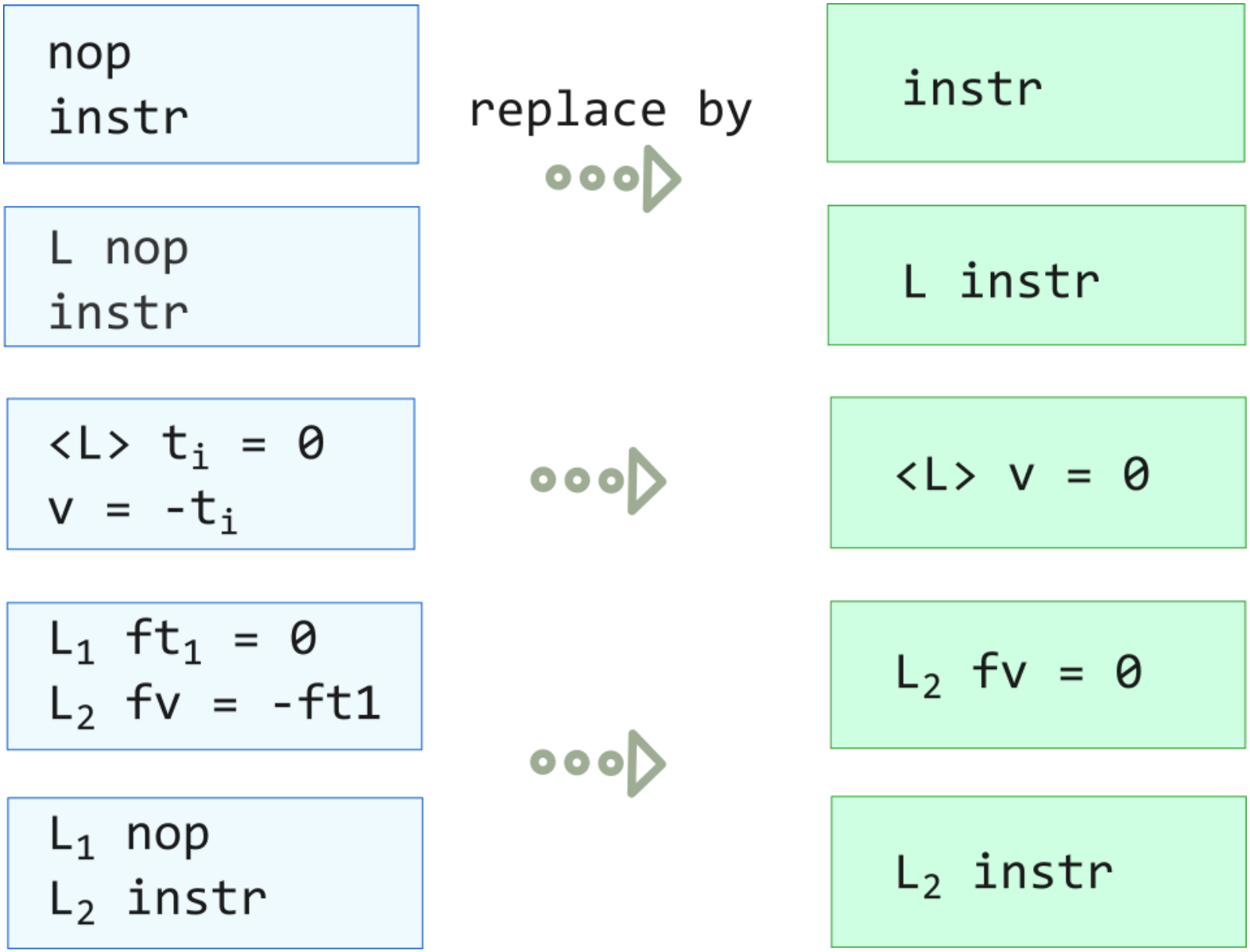
- The goal of optimization is to *reduce redundancy*.
- Must always be sound, i.e., semantics-preserving
- Must be cost-effective, the benefits of optimization must be worth the effort of its implementation

Peephole Optimization

- Performed over a small set (peephole) of instructions
- This peephole slides across the code, during which optimizations matching certain patterns is performed.



Peephole Optimization - Patterns



Dead-code Elimination

- An optimization method focused on detecting and eliminating dead instructions.
- An instruction is dead if it only computes values not used in any instruction on any execution path leading from the instruction.

```
int f(int x, int y) {
  int z = x * y;   DEAD
  return x + y;
}
```

Approach: on a pass mark some instructions as essential, and iterate the process in order to find the maximal set of essential instructions.

- Remaining non-essential instructions are considered dead.

Unreachable-code Elimination

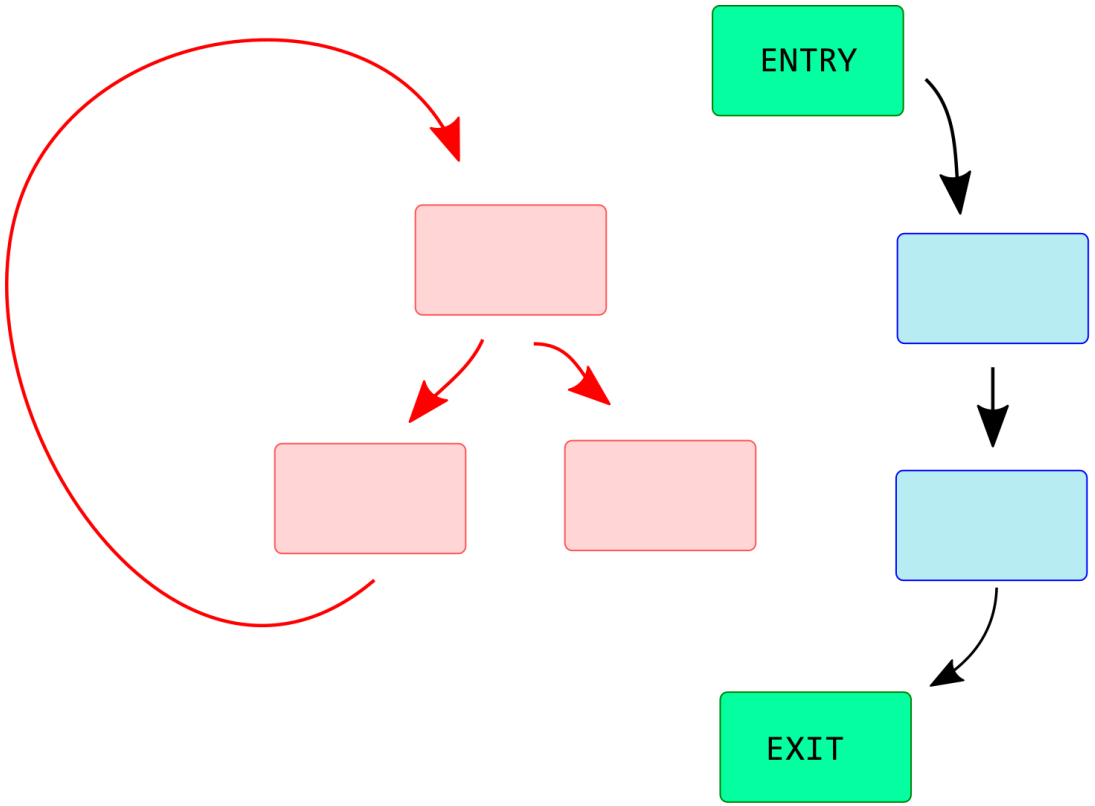
- Identify and remove block that are not executed under any conditions (waste of space)
- An instruction is deemed unreachable if it does not lie on any execution path.
- Can be run at any level of intermediate or target code.

```
int f(int x, int y) {
  return x + y;
  int z = x * y;
}
```

UNREACHABLE

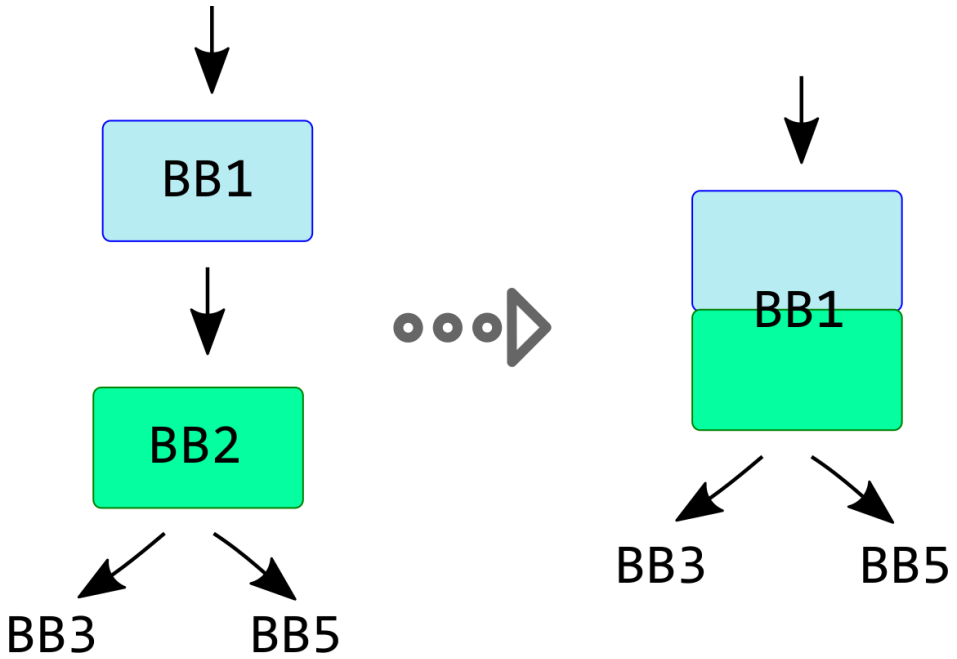
Unreachable-code Elimination - Method

1. mark the procedure's entry node as reachable
2. mark every successor of a marked node as reachable and repeat until no further marking is required



Straightening

- Join two blocks BB1 and BB2 into one block provided:
  - BB1 has no successor other than BB2
  - BB2 has no other predecessor other than BB1
- The successor of the fused block corresponds to the successors of BB2



If Simplification

1. Conditional jumps with constant value are simplified to unconditional jumps or deleted

```
if true goto L1
if false goto L1
```

```
goto L1
```

2. Conditional jumps followed by empty "fall through" branch are simplified

```
   t12 =  !t11
   if t12 goto L1
   goto L2
L1:...
   ...
L2:...
```

```
if t11 goto L2
L1:...
   ...
L2:...
```

3. conditional jumps to empty branch are simplified

```
   if t12 goto L1
   ...
   goto L2
L1:goto L2
L2:...
```

```
if t12 goto L2
...

L2:...
```

Value Numbering

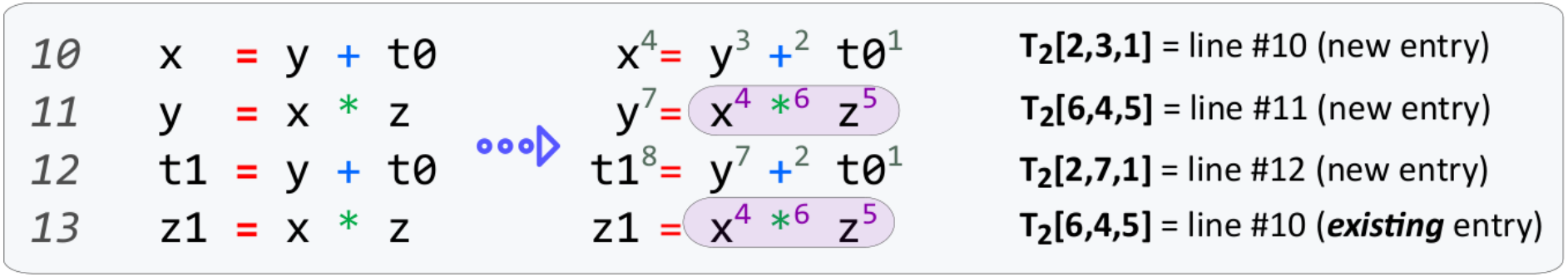
- Identify identical computations and remove one of them with a semantics preserving transformation.
- Assign a value number, V(n), to each expression:
  - V(x) = V(y) **iff** x and y are equivalent and therefore interchangeable.
  - Use hashing over the value numbers to make it efficient
- Replaces redundant expressions
- Simplify algebraic entities

```
10        x  = y + t0
11        y  = x * z
12        t1 = y + t0
13        z1 = x * z
```

Value Numbering - Algorithm

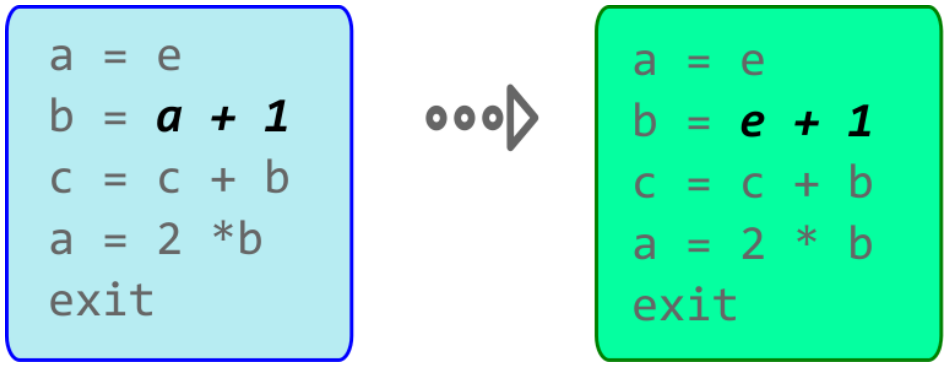
For each operation $o = \langle operator, o_1, o_2 \rangle$ in the block:

1. Get value numbers for operands from hash lookup
2. Hash $\langle operator, V(o_1), V(o_2) \rangle$ to get a value number for o
3. If o already had a value number, replace o with a reference
4. If o_1 and o_2 are constant, evaluate it and replace with a **loadI**



Copy Propagation

- A copy instruction is an instruction in the form: $\mathbf{x = y}$
- Copy propagation replaces later uses of x with uses of y provided **intervening instructions** do not change the value of either x or y
- Benefit: saves computations, reduces space; enables other transformations.

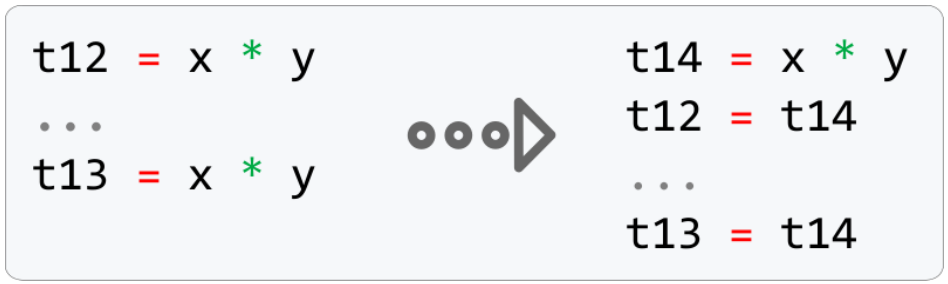


Common Sub-Expression Elimination (CSE)

- Finds two identical expressions and replaces the latter occurrence by a saved value.

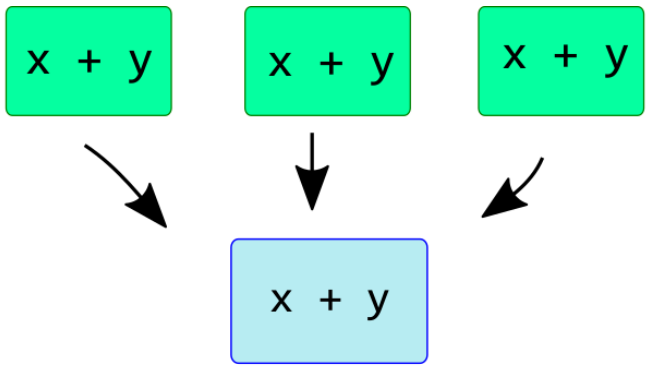
Local CSE Algorithm

1. Traverse basic block from top to bottom
2. Maintain table of expressions evaluated so far
   - if any operand of the expression is redefined, remove it from the table
3. Modify applicable instructions as you go
   - generate temporary variable, store the expression in it and use the variable next time the expression is encountered.

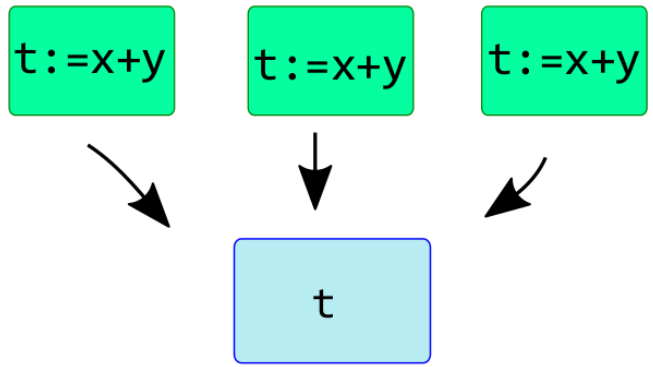


Global CSE

- An expression **e** is available at Entry to B if on every path in the flow graph from Entry to B, there is an evaluation of **e** at B that is not subsequently killed.
- Solve By:
  1. Find Available expressions (Data flow problem)
  2. For each available expression **e**:



Do backward search from e in CFG to find the evaluations of e



Create new temp t to hold previous evaluations, and replace e by t

Loop-invariant Code Hoisting

An expression is a loop invariant if all its operands are invariant. An operand is invariant if **one of the following** hold:

1. it is a constant
2. all definitions of the operand that reach this use are located outside the loop
3. there is one definition of the operand that reaches this use and is inside the loop and is an invariant

```
for(i = 0; i < 10 * x; i++)
  for(j = 0; j < 100; j++)
    y[i][j] = 10 * x;
```

```
z = 10 * x;
for(i = 0; i < z; i++)
  for(j = 0; j < 100; j++)
    y[i][j] = z;
```

Any candidate assignments to be hoisted must be in a basic block that dominates all uses of the left hand variable in the loop and all exit blocks of the loop.