A3

peter.chou

March 2023

1 Code

1.1 Loop Invariant Code Motion

1. How do you check for loop invariance, also, why do you think we need the additional conditions in Listing **??** (especially the first two)?

   (a) isSafeToSpeculativelyExecute checks if an instruction has a side effect such as a modifying memory in this case it would be very difficult to check if the instruction is invariant since we can analyze the memory of the executing program.

   (b) mayReadFromMemory check if an instruction may read from memory so it might or might not read from memory depending on optimizations if it does read from memory we can never be sure if it is invariant because we cannot analyze run time memory at compile time.

   (c) A LandingPadInst is an instruction for catching exceptions thrown by the program as such they can never be invariant

2. How do you hoist the code that is loop-invariant to the loop preheader, or sink to the loop exit?
   We hoist the code to the loop preheader if the instruction meets the following condition

   (a) The Instruction is in blocks that dominate all the exits of the loop

   (b) The Instruction is assign to variable not assigned to elsewhere in the loop

   (c) The Instruction is in blocks that dominate all blocks in the loop that use the variable assigned

3. Which one(s) of the loop structures (for, while, do-while) can LICM work naturally (i.e., without modifying the CFG) and why? What transformations do you need to make for those loop structures to which LICM is not directly applicable?

A do while loop naturally fits with LICM because the loop body is guaranteed to execute once thus hoisting the code out of the loop we can safely hoist the code out of the loop where as a while loop might not even execute depending on the condition which can cause us to execute invariant code that should not be executed. We could transform a while/for loop to a do while loop by introducing an if guard
Example: while loop → do While loop

```
// while loop
int i = 0;
while (i < n) {}
// transformed while loop
if (i < n) {
    do {} while (i < n)
}
```

Example: For loop → do While loop

```
// for loop
for (int i = 0; i < x; i++) {}
// transformed for loop
int i = 0;
if (i < x) {
    do {
        i++;
    } while (i < x)
}
```

2 Theoritical Questions

2.1 Pointer Analysis

```
S1:  p = malloc();
S2:  q = malloc();
S3:  p = q;
S4:  r = malloc()
S5:  q = r;
S6:  if (a)
S7:      p = q;
S8:  if (b)
S9:      p = r;
S10: if (!a)
S11:     p = malloc();
S12: ? = *p;
```

2.1.1 Flow Insensitive Analysis

$a_{s12} \rightarrow$ { heapS1, heapS2, heapS4, heapS11 }

2.1.2 Flow Sensitive Analysis

$a_{s12} \rightarrow$ { heapS2, heapS4, heapS11 }

2.1.3 Path Sensitive

$a_{s12} \rightarrow$ { heapS2, heapS4, heapS11 }

```
int a, b, c, *p, *q;
int main()
{
S1:      f1();
S2:      p = &a;
S3:      f2();
S4:      p = q;
S5:      f3();
}
void f1()
{
S6:      p = &a;
S7:      q = &c;
S8:      f2();
S9:      f3();
}
void f2()
{
S10:     p = &b;
S11:     q = &a;
S12:     f3();
}
void f3()
{
S13:     ? = *p;
}
```

2.1.4 context insensitive

$a_{s13} \rightarrow$ { a, b, q }

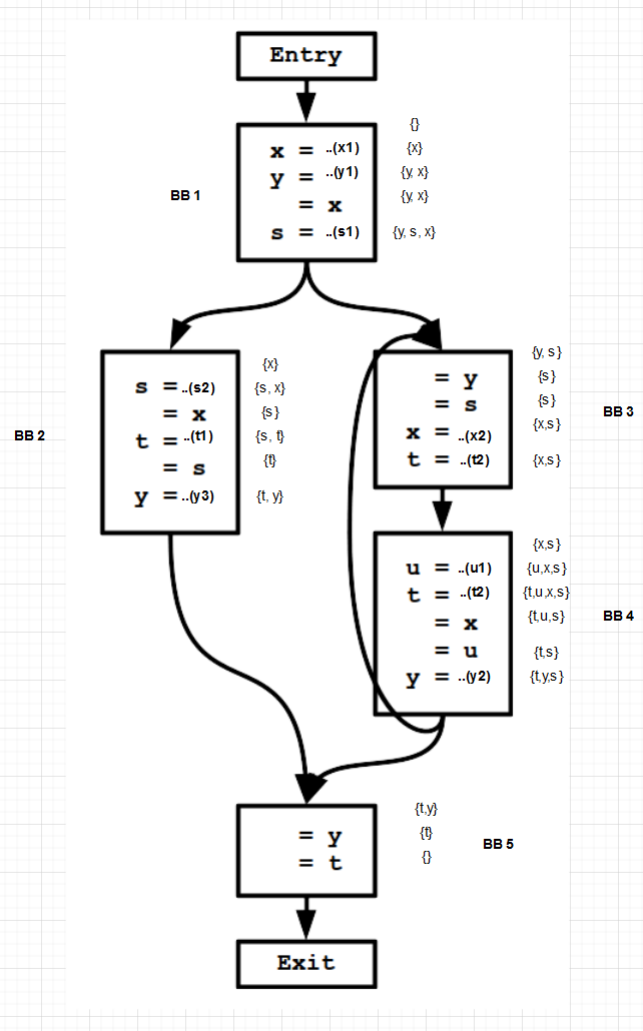
2.1.5 context sensitive

Called from S5: $a_{s13} \rightarrow$ { a }
Called from S9: $a_{s13} \rightarrow$ { b }

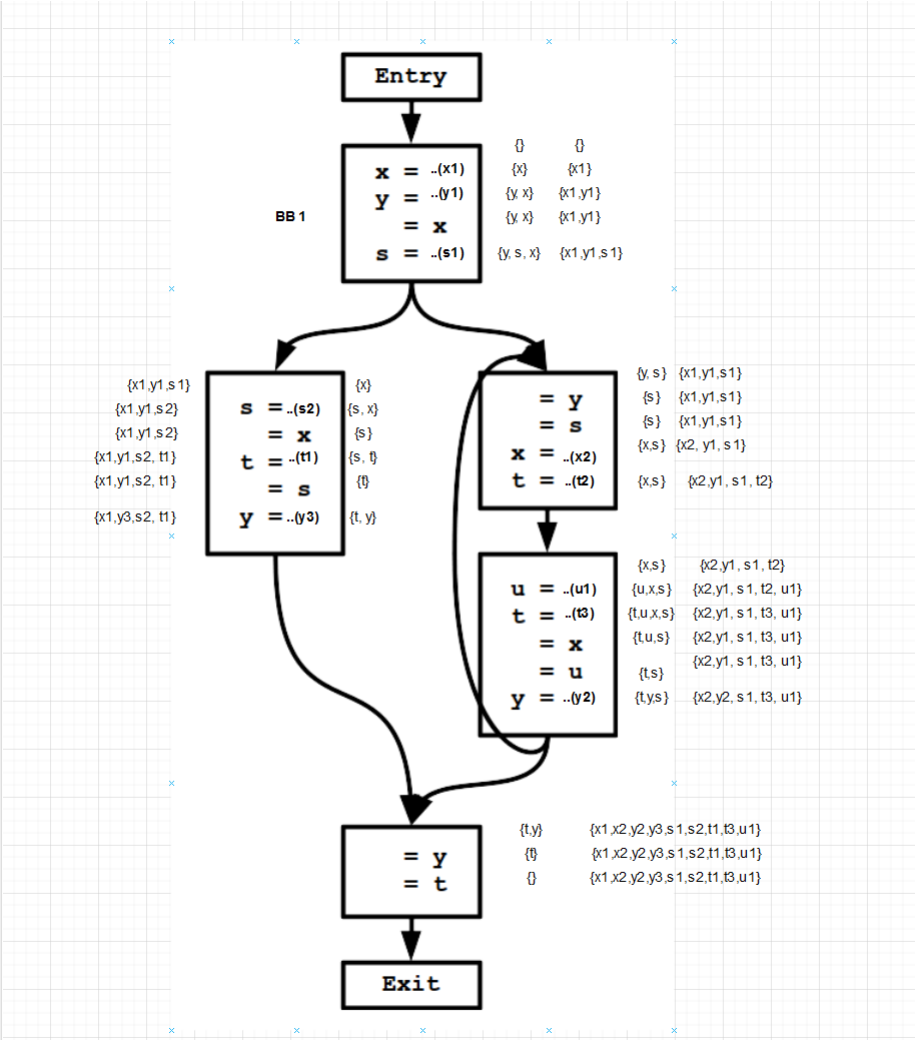
Called from S12: $a_{s13} \rightarrow$ { q }

2.2 Register Allocation

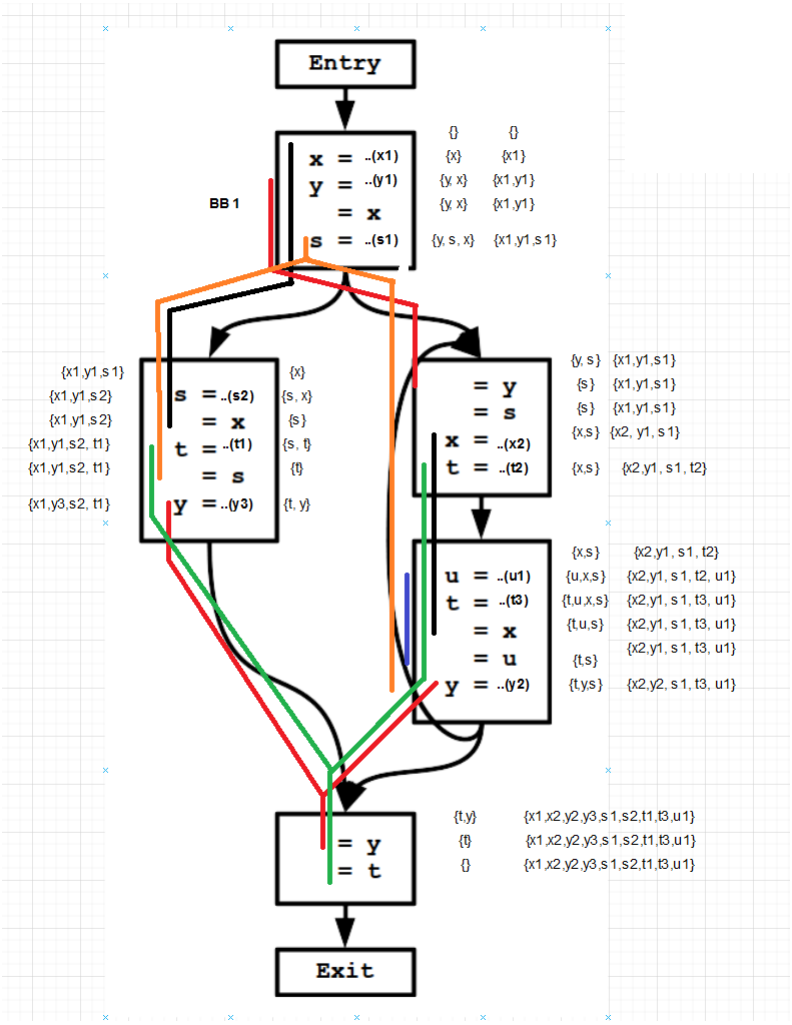
1. Live Variables



2. Reaching Definition

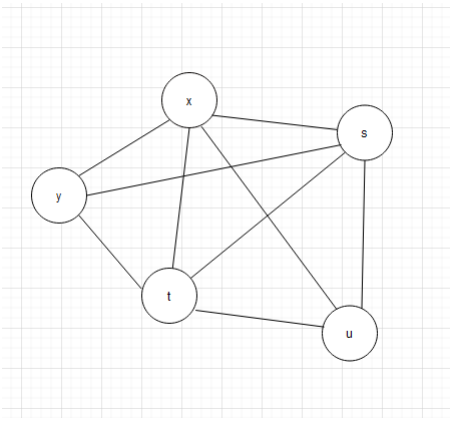


3. Live Ranges



Black = x, Red = y, Orange = s, Green = t, Blue = u

4. Interference Graph



5. Final Color Graph