Exercises on Compilers

Jordi Cortadella

February 7, 2022

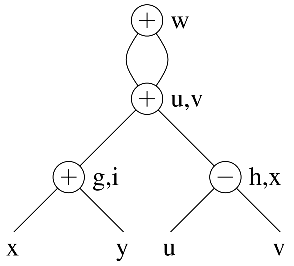
Code optimization

1. Consider the following basic block:

```
g = x + y
h = u - v
i = x + y
x = u - v
u = g + h
v = i + x
w = u + v
```

- Build the DAG representation.
- Generate code under the assumption that the set of live variables is $\{g, h, u, w\}$.
- Generate code under the assumption that the set of live variables is $\{u, h, x\}$.

Solution:



```
g = x + y
h = u + v
u = g + h
w = u + u

{live: g, h, u, w}
```

```
g = x + y
h = u + v
u = g + h
x = h

{live: u, h, x}
```

2. Consider the following code:

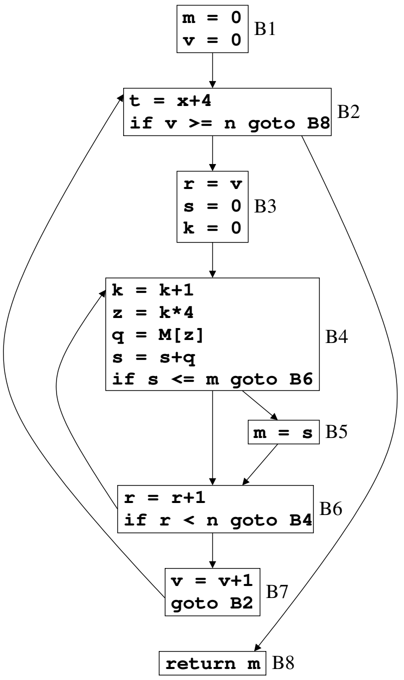
```
1:   m = 0
2:   v = 0
3:   t = x+4
4:   if v >= n goto 18
5:   r = v
6:   s = 0
7:   k = 0
8:   k = k+1
9:   z = k*4
10:  q = M[z]
11:  s = s+q
12:  if s <= m goto 14
13:  m = s
14:  r = r+1
15:  if r < n goto 8
16:  v = v+1
17:  goto 3
18:  return m
```

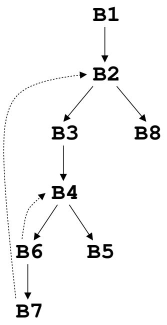
- Calculate the basic blocks and draw the associated control-flow graph.
- Calculate the tree of dominators.
- Identify the back edges and loops of the control-flow graph.
- Perform loop optimizations (assume that all variables are alive at the exit of the code).

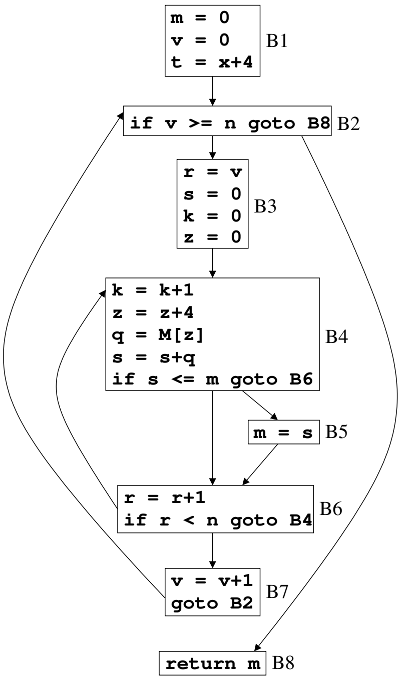
Solution:
The figure at the left shows the control-flow graph. The one in the middle shows the tree of dominators. The back edges are shown as dotted arcs in the tree of dominators. There are two loops determined by the sets of basic blocks {`B4, B6`} and {`B2, B3, B4, B6, B7`}.

Optimizations:

- The statement `t = x+4` in B2 is an invariant and can be moved out of the outermost loop.
- The variables `k` and `z` are induction variables. After extracting the initialization out of the loop, the multiplication can become an addition. The final code is the result of constant propagation after having moved `z = k+0` out of the loop (see the rules for treating induction variables). In this case, no auxiliary variable has been used as `B3` dominates `B4`. The same result could be obtained by using an auxiliary variable and eliminating copies.

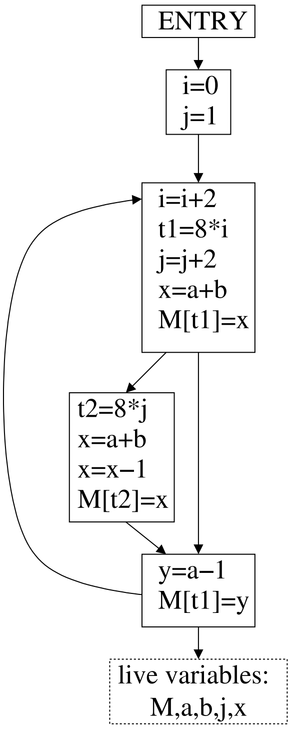




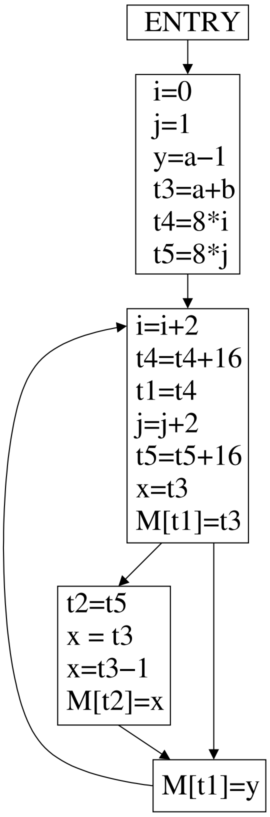


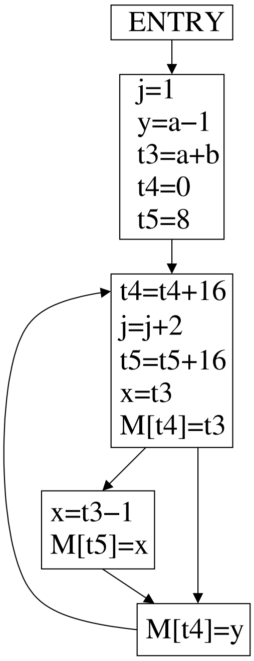
3. Consider the code in the figure:

   - Apply all possible optimizations considering the live variables specified at the exit of the code.
   - Show the final code after the optimizations and explain the transformations produced by each optimization.



Solution:





The optimizations are described in two steps (left and right figures).

Figure at the left:

- The invariant `a-1` is extracted out of the loop. There is no need for an auxiliary variable, therefore the instruction `y=a-1` is extracted.
- The invariant `a+b` is extracted out of the loop. We need an auxiliary variable (`t3`) since there are multiple assignments of variable `x`. The copies `x=t3` are generated and propagated i.e., `M[t1]=t3` and `x=t3-1`.
- `t1` and `t2` are induction variables. They auxiliary variables `t4` and `t5` are used for the induction. They are initialized outside the loop and the copies `t1=t4` and `t2=t5` are generated.

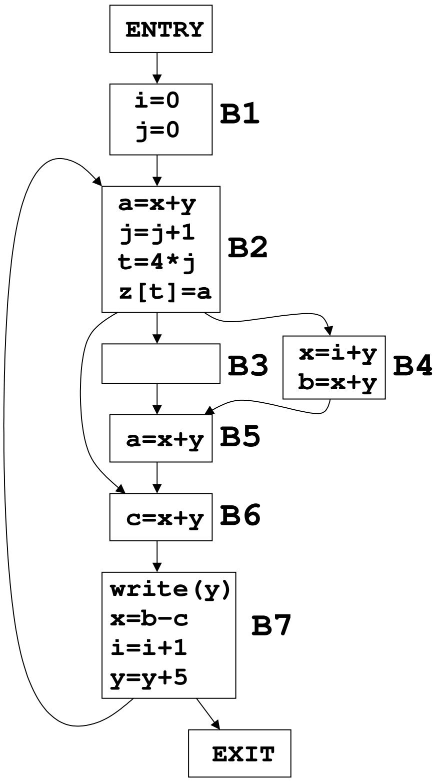
Figure at the right:

- Constants are propagated: `t4=0` and `t5=0`.
- Variable `i` becomes dead and the assignments can be eliminated.
- The copies `t1=t4` and `t2=t5` are propagated and eliminated since `t1` and `t2` become dead.
- Variable `x` is dead before the assignment `x=t3-1`. The assignment `x=t3` can be eliminated. **Observation:** the first assignment `x=t3` cannot be eliminated since `x` is alive after the assignment.

4.

Consider the code in the figure and assume that the array `z` is the only variable alive at the exit of the code.

- Which variables are alive at the end of block B7?
- Which blocks dominate B7?
- Assume that `x` and `y` are parameters of the function initialized before the function call. Is there any variable that could be potentially used without having been initialized before?
- Apply all possible optimizations you know.



Solution:

The variables alive at the end of B7 are: `b`, `i`, `j`, `x`, `y` and `z`.

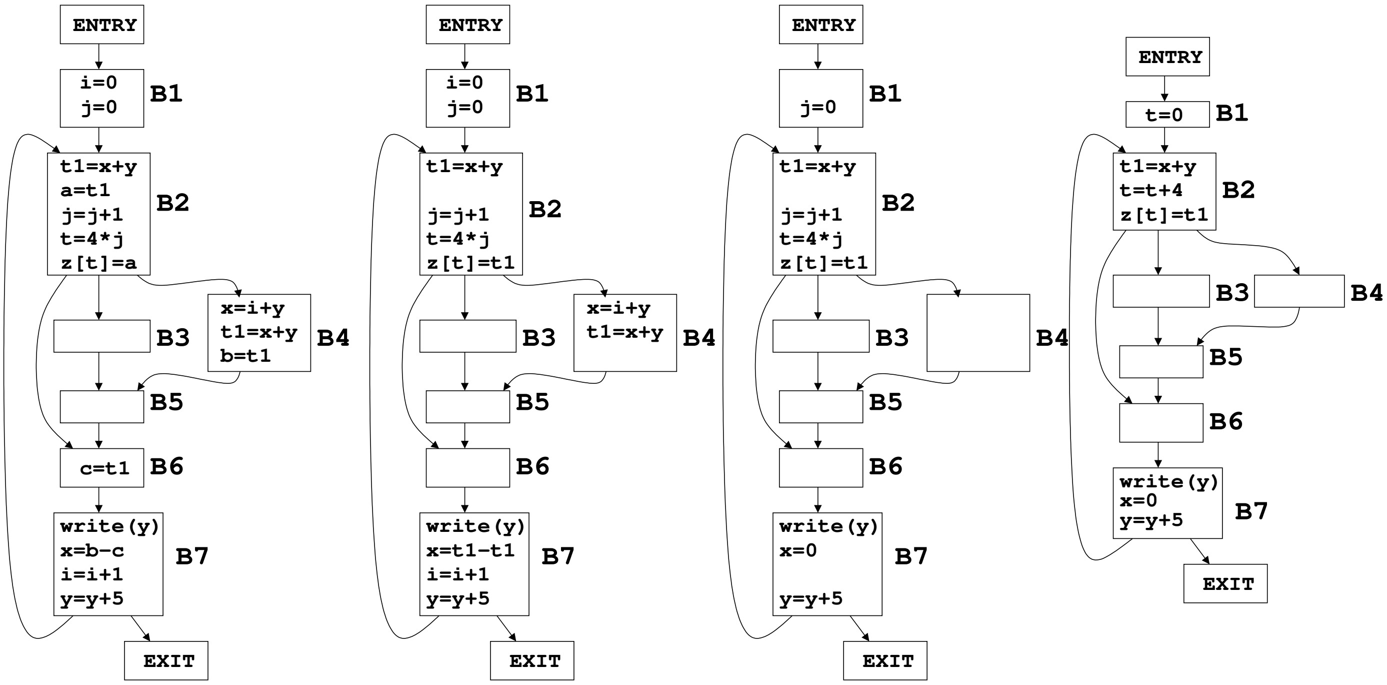
The blocks that dominate B7 are B1, B2, B6 and B7.

Variable `b` is alive at the entry of the the function and could be potentially used without having been initialized.

Optimizations (see next figure):

There is a bug in the following solution:
In step (b) the copy `b=t1` cannot be propagated to the assignment `x=b-c`. `b` could be an initialized local variable, or a function parameter. As a consequence `b` does not become dead, and the copy cannot be removed.

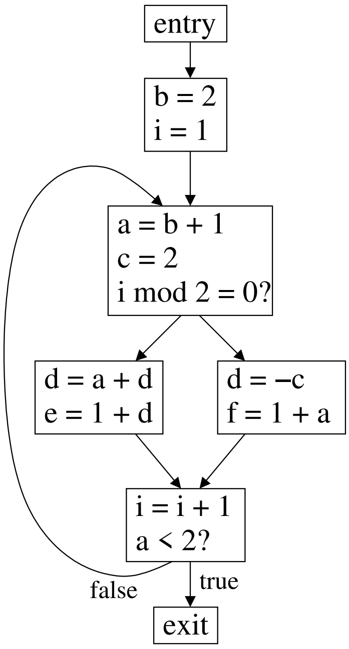
- Figure (a):
  - Variable `a` is dead at the end of `B5`, thus the assignment at `B5` can be removed.
  - The common expression `x+y` at `B6` is substituted by a new variable `t1`.
- Figure (b):
  - The copy `a=t1` is propagated in `B2`. Variable `a` becomes dead and the assignment can be eliminated.
  - The copies `b=t1` and `c=t1` are the only definitions of `b` and `c` that reach the assignment `x=b-c`. The copies can be propagated. After that, `b` and `c` become dead.
- Figure (c):
  - After the optimization `x=0`, variable `t1` becomes dead at `B4`.
  - After removing the assignment of `t1` at `B4`, variable `x` becomes dead at `B4` and the assignment `x=i+y` can be removed.
  - Variable `i` becomes dead at the end of `B7` and the assignment `i=i+1` can be removed. Similarly for `i=0`.
- Figure (d):
  - The induction variable `t` is optimized. The assignment `t=4*j` is moved to `B1` and substituted by `t=t+4` in `B2`. After constant propagation, the variable `j` becomes dead and can be eliminated.



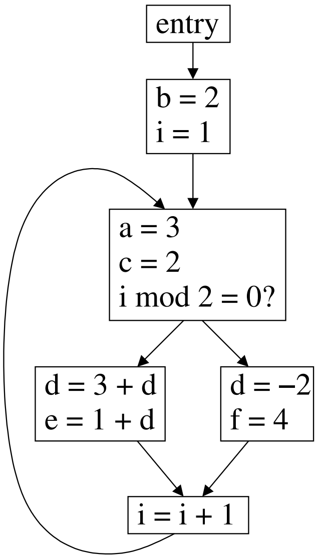
(a) (b) (c) (d)

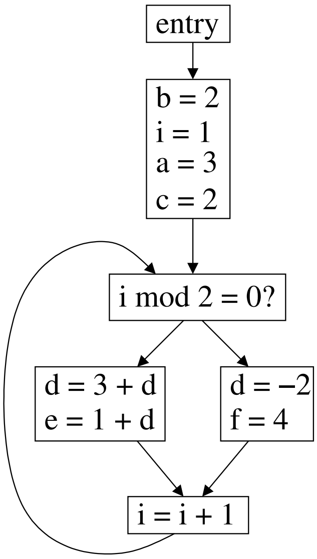
5. Consider the code in the figure:

   - Apply all possible optimizations.
   - Discuss how much information can be extracted at compile time about the execution of the code.



Solution:





The figure at the left shows the code after constant propagation. The most relevant optimization is the elimination of the condition `a<2`, which is always false.

The figure at the right shows the code after moving invariants out of the loop. Since no information is provided about the liveness of the variables, all of them are assumed to be alive.

The optimization reveals that this is an infinite loop. In general, infinite loops may not be incorrect since the core of many reactive systems consist of an infinite loop that read events and react to them.

In general, detecting whether a loop is infinite or not is an undecidable problem (the Halting Problem!), however in some particular cases it is possible to infer this information.

6.

```
1: a = y + c
2: d = a + x
3: b = x + y
4: a = b * d
5: e = x - a
6: y = b + x
7: a = e * b
```

Let us assume that the set of live variables at the end of the code at the left is $\{a, y\}$.

- Indicate which variables are alive at each point of the code.
- Rewrite the code using a minimum number of registers. Name the registers as `R1`, `R2`, `R3`, `R4`, etc.

Solution:

```
{c, x, y}
1: a = y + c
{a, x, y}
2: d = a + x
{d, x, y}
3: b = x + y
{b, d, x}
4: a = b * d
{a, b, x}
5: e = x - a
{b, e, x}
6: y = b + x
{b, e, y}
7: a = e * b
{a, y}
```

⇒

The code can be rewritten with only 3 registers:

```
{R1:c, R2:x, R3:y}
1: R1 = R3 + R1
{R1:a, R2:x, R3:y}
2: R1 = R1 + R2
{R1:d, R2:x, R3:y}
3: R3 = R2 + R3
{R1:d, R2:x, R3:b}
4: R1 = R3 * R1
{R1:a, R2:x, R3:b}
5: R1 = R2 - R1
{R1:e, R2:x, R3:b}
6: R2 = R3 + R2
{R1:e, R2:y R3:b}
7: R1 = R1 * R3
{R1:a, R2:y}
```