Homework Sheet 7, Ex. 2

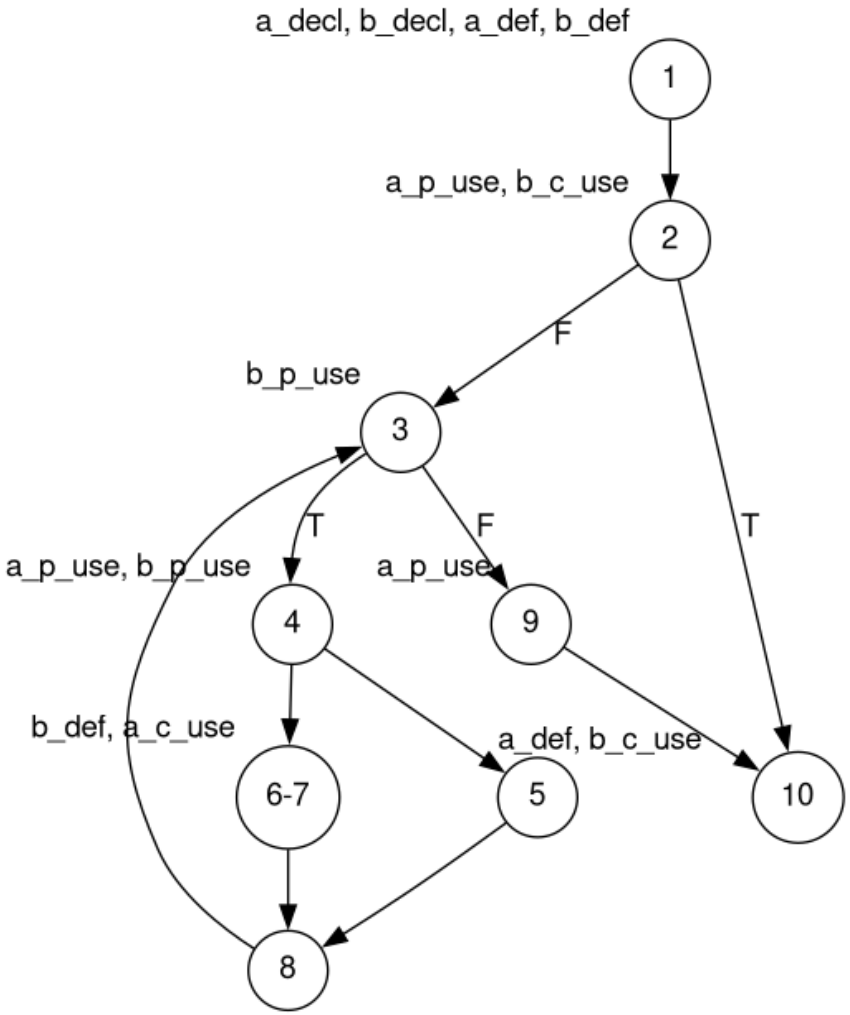
Büchler P., Lohmann N.J

Institute for Software Systems

June 26, 2023

Contents

Data Flow Graph



DU-Paths (a)

- 1-2, 1-2-3-9
- 1-2-3-4, 1-2-3-4-6-7, 1-2-3-4-6-7-9
- 5-8-3-4, 5-8-3-4-6-7
- 5-8-3-9

DU-Paths (b)

- 1-2, 1-2-3
- 1-2-3-4, 1-2-3-4-5
- 6-7-8-3, 6-7-8-3-4, 6-7-8-3-4-5

Test Suite

Test	#1
Test Path	1-2
Coverage	1-2
Input	a = 0, b = 0
Output	0

Test	#2
Test Path	1-2-3-9
Coverage	1-2, 1-2-3-9
Input	a = 1, b = 0
Output	1

Test Suite cont.

Test	#3
Test Path	1-2-3-4-6-7-8-3-9
Coverage	1-2, 1-2-3-4, 1-2-3-4-6-7, 1-2-3-4-6-7-8-9
Input	a = 1, b = 1
Output	1

Test	#4
Test Path	1-2-3-4-5-8-3-4-6-7-8-3-9
Coverage	1-2, 1-2-3-4, 5-8-3-4, 5-8-3-4-6-7, 5-8-3-4-9
Input	a = 2, b = 1
Output	1

Comparison Data Flow vs. Branch coverage

For 100% multiple-condition coverage, $c = 3 \implies 2^c = 8$
Twice the No. of Data-Flow cases, due to test cases of the form $a = 0, b = *$. Which never go further than the path 1-2-3-9.

Practice

But how does this actually look like? After all our Data Flow graphs are fortunately made for us humans and hence subjective!, but a compiler needs to be precise. DU-paths (commonly known as Def-Use chains) are too complex to generate but go in the right direction. The next step are Static Single-Assignment form (SSA) derived from the CFG.

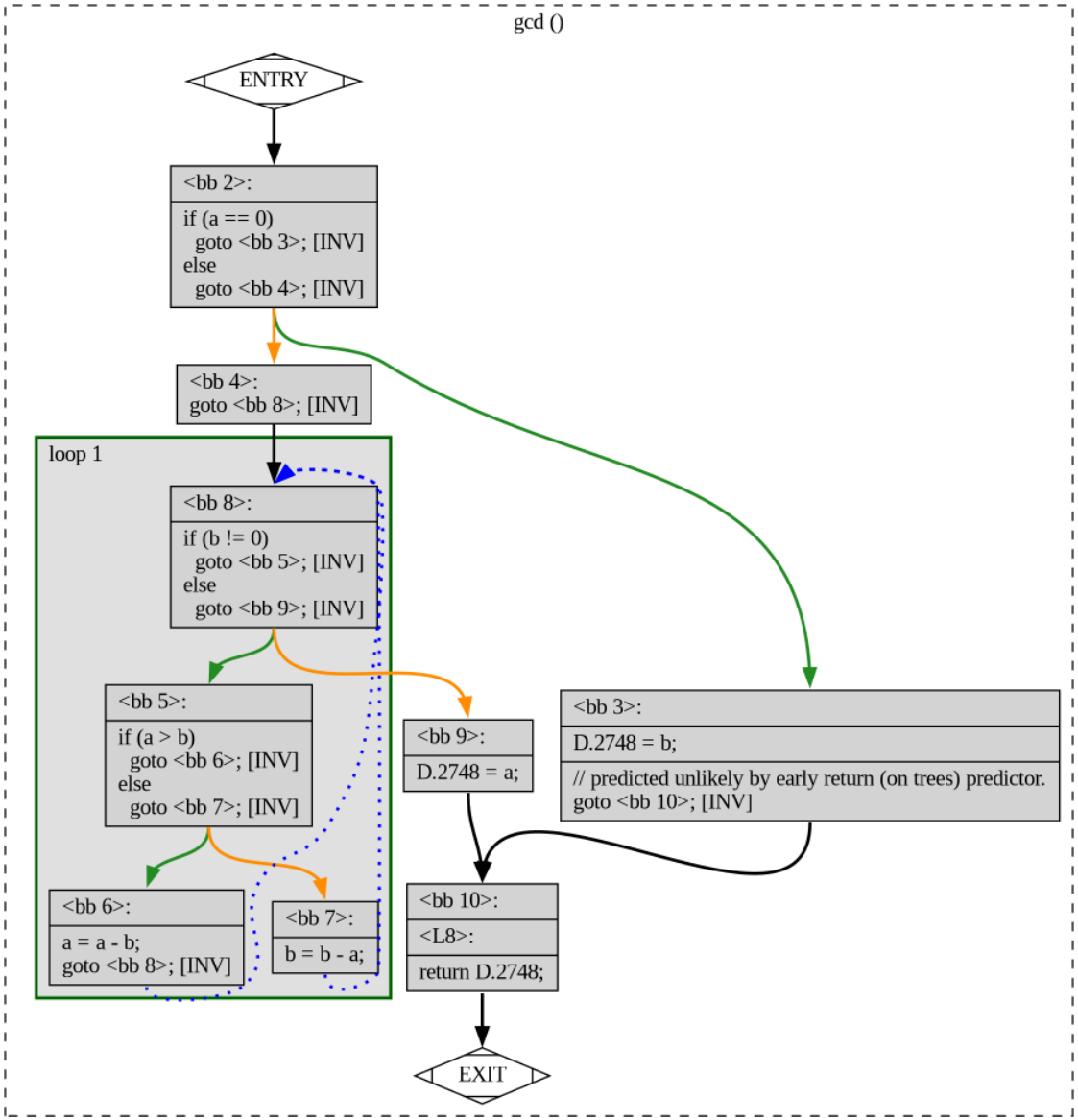
One can generate quiet helpful information that the compiler *gcc* $\geq$ 4.0 uses via

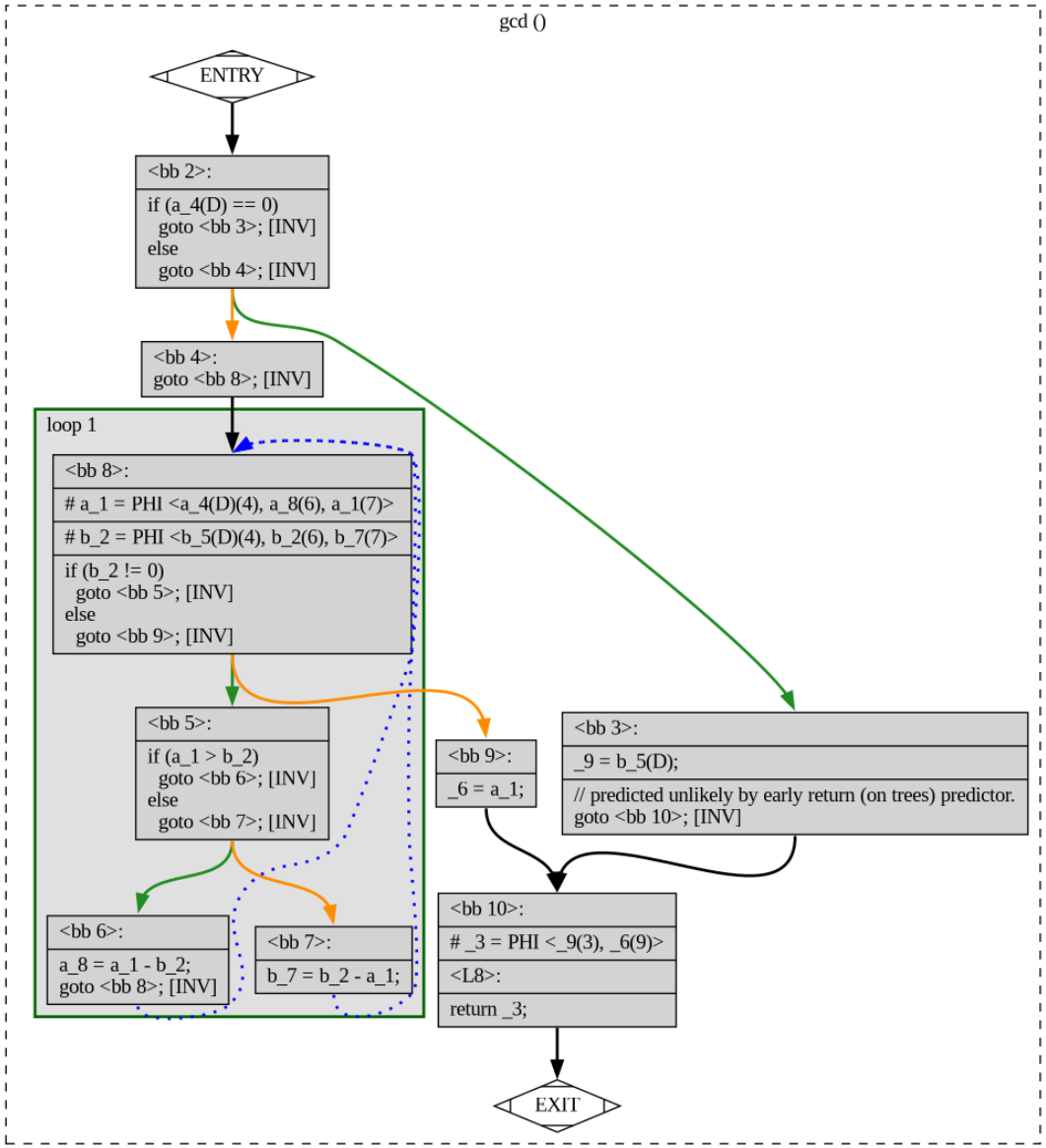
```
gcc gcd_data_flow.c -fdump-tree-all-graph
```

Most importantly

```
gcd_data_flow.c.cfg
```

```
gcd_data_flow.c.ssa
```


gcd ()
ENTRY
<bb 2>:
if (a == 0)
goto <bb 3>; [INV]
else
goto <bb 4>; [INV]
<bb 4>:
goto <bb 8>; [INV]
loop 1
<bb 8>:
if (b != 0)
goto <bb 5>; [INV]
else
goto <bb 9>; [INV]
<bb 5>:
if (a > b)
goto <bb 6>; [INV]
else
goto <bb 7>; [INV]
<bb 6>:
a = a - b;
goto <bb 8>; [INV]
<bb 7>:
b = b - a;
<bb 9>:
D.2748 = a;
<bb 3>:
D.2748 = b;
// predicted unlikely by early return (on trees) predictor.
goto <bb 10>; [INV]
<bb 10>:
<L8>:
return D.2748;
EXIT


gcd ()
ENTRY
<bb 2>:
if (a_4(D) == 0)
goto <bb 3>; [INV]
else
goto <bb 4>; [INV]
<bb 4>:
goto <bb 8>; [INV]
loop 1
<bb 8>:
# a_1 = PHI <a_4(D)(4), a_8(6), a_1(7)>
# b_2 = PHI <b_5(D)(4), b_2(6), b_7(7)>
if (b_2 != 0)
goto <bb 5>; [INV]
else
goto <bb 9>; [INV]
<bb 5>:
if (a_1 > b_2)
goto <bb 6>; [INV]
else
goto <bb 7>; [INV]
<bb 9>:
_6 = a_1;
<bb 3>:
_9 = b_5(D);
// predicted unlikely by early return (on trees) predictor.
goto <bb 10>; [INV]
<bb 6>:
a_8 = a_1 - b_2;
goto <bb 8>; [INV]
<bb 7>:
b_7 = b_2 - a_1;
<bb 10>:
_3 = PHI <_9(3), _6(9)>
<L8>:
return _3;
EXIT

Coverage

How about the statement/decision coverage? Really dependent on the languages ecosystem and most are happy with statement coverage. For c there is gcov available. Running

```
#include <assert.h>
unsigned gcd(unsigned a, unsigned b) {
    if (a == 0) return b;
    while (b != 0) {
        if (a > b) {
            a -= b;
        } else {
            b -= a;
    } }
    return a;
}

int main() {
    assert(gcd(0, 0) == 0);
    assert(gcd(1, 0) == 1);
    assert(gcd(1, 1) == 1);
    assert(gcd(2, 1) == 1);
    return 0;
}
```

```
$ gcc -Wall -fprofile-arcs -ftest-coverage gcd_data_flow_test.
$ ./a.out
$ gcov -b -c gcd_data_flow_test.c
```

```
    File 'gcd_data_flow_test.c'
    Lines executed:100.00% of 13
    Branches executed:100.00% of 14
    Taken at least once:71.43% of 14
    Calls executed:50.00% of 8
    Creating 'gcd_data_flow_test.c.gcov'

    function gcd called 4 returned 100% blocks executed 100%
        4:    3:unsigned gcd(unsigned a, unsigned b) {
        4:    4:    if (a == 0) return b;
branch  0 taken 25% (fallthrough)
branch  1 taken 75%
        6:    5:    while (b != 0) {
branch  0 taken 50%
branch  1 taken 50% (fallthrough)
        3:    6:        if (a > b) {
branch  0 taken 33% (fallthrough)
branch  1 taken 67%
        1:    7:            a -= b;
        -:    8:        } else {
        2:    9:            b -= a;
        -:   10:    } }
        3:   11:    return a;
        -:   12:}
        -:   13:
```

Removing the last test gives instead

```
    File 'gcd_data_flow_test.c'
    Lines executed:91.67% of 12
    Branches executed:100.00% of 12
    Taken at least once:66.67% of 12
    Calls executed:50.00% of 6
    Creating 'gcd_data_flow_test.c.gcov'
```

```
        function gcd called 3 returned 100% blocks executed 90%
        3:    3:unsigned gcd(unsigned a, unsigned b) {
        3:    4:    if (a == 0) return b;
branch  0 taken 33% (fallthrough)
branch  1 taken 67%
        3:    5:    while (b != 0) {
branch  0 taken 33%
branch  1 taken 67% (fallthrough)
        1:    6:        if (a > b) {
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
    #####:    7:            a -= b;
        -:    8:        } else {
        1:    9:            b -= a;
        -:   10:    } }
        2:   11:    return a;
        -:   12:}
```

The End

Thank you for your attention!