CMSC 430
Introduction to Compilers
Fall 2018

Data Flow Analysis
Applications and Implementations
Data Flow Analysis

• A framework for proving facts about programs

• Reasons about lots of little facts

• Little or no interaction between facts
  ▪ Works best on properties about how program computes

• Based on all paths through program
  ▪ Including infeasible paths

• Operates on control-flow graphs, typically
Most data flow analyses can be classified this way

- A few don’t fit: bidirectional analysis

Lots of literature on data flow analysis
Applications: Reaching Defs.

- **Constant propagation**: if all definitions of a given variable’s use are the same constant value, just assign the constant directly.

- **Loop invariant code motion**: if an expression is computed in a loop, but all of the components are defined outside the loop, the code can move.
Applications: Liveness

• **Register allocation**: variables that are not live in a given basic block (or subgraph) do not need to be in registers. More on this later.

• **Dead code elimination**: variables that are assigned but not live after the assignment don’t need to be computed at all.
Applications: Available Exprs.

- **Common sub-expression elimination**: create a new variable containing the result of an expression. Replace subsequent uses of the expression with a read from the variable.
Applications: Very Busy Exprs.

- **Code motion**, e.g., move the computation of an expression to before a loop or branch.

- If the same expression will be computed on every branch of a conditional, or every time through the loop, it can be pre-computed.

```c
if (a < b) {
    x = a - b
} else {
    y = a - b
}
```

```c
t = a - b
if (a < b) {
    x = t
} else {
    y = t
}
```
Implementations

• Optimizing compilers implement data-flow analysis

• GCC:
  - https://github.com/gcc-mirror/gcc/blob/master/gcc/df-core.c
  - https://github.com/gcc-mirror/gcc/blob/master/gcc/df-problems.c

• Clang:
  - https://clang.llvm.org/doxygen/LiveVariables_8cpp_source.html
  - https://github.com/llvm-mirror/clang/blob/master/lib/Analysis/LiveVariables.cpp
Implementations (cont.)

- Static analysis and bug-finding tools also use DFA

- Haskell package for LLVM: http://hackage.haskell.org/package/llvm-analysis-0.3.0/docs/LLVM-Analysis-Dataflow.html

- C Intermediate Language (CIL)
  - https://github.com/cil-project/cil
  - Written in OCaml!
  - Stable but no longer directly maintained
  - Used in Frama-C: http://frama-c.com/
Using CIL on Grace

```bash
$ ssh grace.umd.edu
$ source /afs/glue.umd.edu/class/fall2018/cmsc/430/0201/public/.opam/opam-init/init.csh
$ git clone https://github.com/cil-project/cil
$ cd cil
$ ./configure && make
$ ./bin/cilly --help | less
$ ./bin/cilly \
  --save-temps \
  --doLiveness \
  --live_func=main \
  --live_debug \
  /afs/glue.umd.edu/class/fall2018/cmsc/430/0201/public/src/ex1/ex1.c
```
int main(int argc, char *argv[]) {
    int x, y, z, w, a;

    x = 10;
w = 20;
a = 100;

    y = x + 3;
z = y + w;
w = 42;

    while (z < a) {
        z = z + y;
a = a - 1;
x = x + 1;
        if (z > 5) {
            y = x + 3;
        }
    }

    return x;
}
```c
main()

1: x = 10
   w = 20
   a = 100
   y = x + 3
   z = y + w
   w = 42

2: loop

3: z < a

5: z = z + y
   a = a - 1
   x = x + 1

6: z > 5

7: y = x + 3

4: break

8: return x

1: x(int ), y(int ), z(int ), a(int ),
2: x(int ), y(int ), z(int ), a(int ),
3: x(int ), y(int ), z(int ), a(int ),
4: x(int ),
5: x(int ), y(int ), z(int ), a(int ),
6: x(int ), y(int ), z(int ), a(int ),
7: x(int ), z(int ), a(int ),
8: x(int ),
```