Lecture 22: Dataflow analysis

David Hovemeyer

November 29, 2023

601.428/628 Compilers and Interpreters

- Dataflow analysis in practice

Implementing dataflow analysis

Dataflow analysis is an algorithm in which the behavior varies in specific ways depending on the application:

- ▶ Type of dataflow fact
- ▶ The top fact (the one that combines nondestructively with other facts)
- ▶ Forward or backward
- ▶ How dataflow facts are combined (i.e., "must" or "may")
- ▶ Effect of instructions on dataflow facts

The starter code has a general framework for dataflow analysis which can be customized to do whatever analysis you need to do

- ▶ ...if you want to implement your own dataflow analysis

Analysis class

The "customized" details are encapsulated into an *analysis class*, which derives from either FordwardAnalysis or BackwardAnalysis

The analysis class must provide:
- ▶ FactType: data type of the dataflow facts
- ▶ get_top_fact(): return the top fact
- ▶ combine_facts(): returns the result of combining dataflow facts
- ▶ model_instruction(): updates the dataflow fact that "logically" precedes an instruction to model the effect of executing that instruction
  - ▶ For backward analyses, this is modeling the execution of the instruction in reverse
- ▶ fact_to_string(): returns a text representation of a fact (useful for debugging!)

Example: `LiveVregsAnalysis`

Analysis to determine which virtual registers have live values

```
class LiveVregsAnalysis : public BackwardAnalysis {
public:
  ...details...
};

typedef Dataflow<LiveVregsAnalysis> LiveVregs;
```

```cpp
// We assume that there are never more than this many vregs used
static const unsigned MAX_VREGS = 256;

// Fact type is a bitset of live virtual register numbers
typedef std::bitset<MAX_VREGS> FactType;

// The "top" fact is an unknown value that combines nondestructively
// with known facts. For this analysis, it's the empty set.
FactType get_top_fact() const { return FactType(); }
```

LiveVregsAnalysis: combining facts

```cpp
// Combine live sets. For this analysis, we use union.
FactType combine_facts(const FactType &left, const FactType &right) const {
  return left | right;
}
```

```
void model_instruction(Instruction *ins, FactType &fact) const {
  // Model an instruction (backwards). If the instruction is a def,
  // the assigned-to vreg is killed. Every vreg used in the instruction,
  // the vreg becomes alive (or is kept alive.)

  if (HighLevel::is_def(ins)) {
    Operand operand = ins->get_operand(0);
    fact.reset(operand.get_base_reg());
  }

  for (unsigned i = 0; i < ins->get_num_operands(); i++) {
    if (HighLevel::is_use(ins, i)) {
      Operand operand = ins->get_operand(i);
      fact.set(operand.get_base_reg());
      if (operand.has_index_reg())
        fact.set(operand.get_index_reg());
    }
  }
}
```

```cpp
// Convert a dataflow fact to a string (for printing the CFG annotated with
// dataflow facts)
std::string fact_to_string(const FactType &fact) const {
  std::string s("{");
  for (unsigned i = 0; i < MAX_VREGS; i++) {
    if (fact.test(i)) {
      if (s != "{") { s += ","; }
      s += std::to_string(i);
    }
  }
  s += "}";
  return s;
}
```

```
std::shared_ptr<ControlFlowGraph> hl_cfg = /* ... */;
LiveVregs live_vregs(hl_cfg);
live_vregs.execute();
```

See Dataflow::execute (in dataflow.h) for implementation of dataflow algorithm.

Printing control-flow graph with dataflow facts

For testing and debugging a dataflow analysis, and for making use of dataflow facts in analysis and optimization, it's very helpful to be able to see the exact facts at each location in a function.

See `print_cfg.h`, `print_cfg.cpp` for how printing CFGs with liveness information is implemented

▶ You could add support for printing your own dataflow analysis results

The `-L` command line option enables printing CFG with live vregs analysis results

▶ You could add a command line option for your analysis

```
$ $ASSIGN04_DIR/nearly_cc -L input/example02.c
...output...
        .section .text
        .globl main
main:
BASIC BLOCK 0 [entry]                 /* {} */
  fall-through EDGE to BASIC BLOCK 2
                  At end of block: /* {} */


BASIC BLOCK 1 [exit]                  /* {} */
                  At end of block: /* {} */


BASIC BLOCK 2                         /* {} */
        enter   $0                    /* {} */
        mov_l   vr13, $11             /* {} */
        mov_l   vr10, vr13            /* {13} */
        mov_l   vr14, $1              /* {10} */
        mov_l   vr11, vr14            /* {10,14} */
        mov_l   vr15, $0              /* {10,11} */
        mov_l   vr12, vr15            /* {10,11,15} */
        jmp     .L1                   /* {10,11,12} */
  branch EDGE to BASIC BLOCK 3
                  At end of block: /* {10,11,12} */
...more output...
```