Lecture 22: Dataflow analysis

David Hovemeyer

November 20, 2024

601.428/628 Compilers and Interpreters

Today

- 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 `ForwardAnalysis` 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()`: model the effect of a single instruction on a dataflow fact
- `model_block()`: model the effect of a basic block as a whole on a dataflow fact
- `fact_to_string()`: returns a text representation of a fact (useful for debugging!)

`model_instruction()` vs. `model_block()`

Note that only one of `model_instruction()` and `model_block()` should have actual code in its body. (I.e., one or the other should be a no-op.)

`model_block()` is provided to allow for dataflow analyses that model the effects of blocks rather than individual instructions.

For most analyses, you'll want `model_instruction()`, since that is needed to compute dataflow facts before/after individual instructions.

So, if in doubt, implement `model_instruction()` and leave `model_block()` as a no-op.

Forwards or backwards?

For a forwards analysis, `model_instruction()` and `model_block()` model the effect of an instruction or block on a dataflow going *forwards*, i.e., in program order.

For a backwards analysis, `model_instruction()` and `model_block()` model the effect of an instruction or block on a dataflow going *backwards*, i.e., in the reverse of program order.

Example: LiveVregsAnalysis

Analysis to determine which virtual registers have live values

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

typedef Dataflow<LiveVregsAnalysis> LiveVregs;
```

LiveVregsAnalysis: fact type, top fact

```
// 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

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

LiveVregsAnalysis: modeling instructions

```
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());
    }
  }
}
```

LiveVregsAnalysis: stringifying a fact

```
// Convert a dataflow fact to a string (for printing the CFG annotated with
// dataflow facts)
std::string fact_to_string(const FactType &fact) const {
  return cpputil::stringify_bitset(fact);
}
```

Dataflow class

A Dataflow object parametized with the analysis class is responsible for both

- Running the analysis
- Allowing access to the results of the analysis

E.g.:

```
typedef Dataflow<LiveVregsAnalysis> LiveVregs;
```

LiveVregsAnalysis: running the analysis

```
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.

Using the analysis results

The `Dataflow` class has member functions which return the dataflow fact at a specific location:

- `get_fact_at_beginning_of_block()`: get the fact at the beginning of a specified basic block
- `get_fact_at_end_of_block()`: get the fact at the end of a specified basic block
- `get_fact_before_instruction()`: get the fact at the point just before given instruction in a basic block
- `get_fact_after_instruction()`: get the fact at the point just after given instruction in a basic block

See Lecture 19 slides for example.

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.

The `-D` option allows printing a control-flow graph annotated with dataflow facts.

You could add support for printing results of your own dataflow analysis; see the `print_dataflow_cfg()` function in `driver/main.cpp`

- You would also need to edit `driver/options.cpp` to add a new supported argument to the `-D` option for your dataflow analysis

```
$ $ASSIGN04_DIR/nearly_cc -h -D liveness 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...
```