Lecture 19: Liveness analysis

David Hovemeyer

November 13, 2023

601.428/628 Compilers and Interpreters

Today

- `InstructionSequence` and `ControlFlowGraph`
- Determining liveness (high-level and low-level)
- Using liveness information

InstructionSequence and ControlFlowGraph

InstructionSequence and ControlFlowGraph

`InstructionSequence`: represents a linear sequence of `Instructions` (high-level or low-level)

`ControlFlowGraph`: a graph of `BasicBlocks`

- A `BasicBlock` is just an `InstructionSequence` with a little bit of additional information
- A branch or function call can only be the last instruction in the basic block
- Instructions that are a control flow target must always be the first instruction in a basic block
- Edges of graph represent control flow possibilities

Converting InstructionSequence to ControlFlowGraph

```
// High-level InstructionSequence to high-level CFG
std::shared_ptr<InstructionSequence> hl_iseq = /* ... */
HighLevelControlFlowGraphBuilder hl_builder(hl_iseq);
std::shared_ptr<ControlFlowGraph> hl_cfg = hl_builder.build();

// Low-level InstructionSequence to low-level CFG
std::shared_ptr<InstructionSequence> ll_iseq = /* ... */
LowLevelControlFlowGraphBuilder ll_builder(ll_iseq);
std::shared_ptr<ControlFlowGraph> ll_cfg = ll_builder.build();
```

Converting ControlFlowGraph to InstructionSequence

```
// Works for either high-level or low-level CFG
std::shared_ptr<ControlFlowGraph> cfg = /* ... */
std::shared_ptr<InstructionSequence> iseq =
  cfg->create_instruction_sequence();
```

Computing liveness information for high-level IR

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

// live_vregs now has liveness information for virtual registers
// used in basic blocks of the control flow graph
```

Making use of liveness information

The best way to make use of liveness analysis results (or results from any other dataflow analysis) is to derive a class from `ControlFlowGraphTransform`:

- Your derived class's constructor executes the liveness analysis (and potentially other dataflow analyses) on the `ControlFlowGraph`
- Override the `transform_basic_block` member function to implement a local (basic-block scope) code transformation
- Within `transform_basic_block`, you can use the analysis's `get_fact_before_instruction` and/or `get_fact_after_instruction` functions to get the dataflow fact at the location immediately before or after a specified instruction in the basic block
- For liveness analysis, the dataflow fact is a `std::bitset` containing the register numbers of live virtual or machine registers

A control flow graph transformation

```
class MyTransform : public ControlFlowGraphTransform {
private:
  LiveVregs m_live_vregs;
  // ...other analyses if needed...

public:
  MyTransform(const std::shared_ptr<ControlFlowGraph> &cfg);

  virtual std::shared_ptr<InstructionSequence>
    transform_basic_block(const InstructionSequence *orig_bb);
};
```

CFG transform: constructor

```
MyTransform::MyTransform(const std::shared_ptr<ControlFlowGraph &cfg)
  : ControlFlowGraphTransform(cfg)
  , m_live_vregs(cfg) {
  m_live_vregs.execute(); // compute vreg liveness
}
```

CFG transform: basic block transform

```
std::shared_ptr<InstructionSequence>
MyTransform::transform_basic_block(const InstructionSequence *orig_bb) {
  std::shared_ptr<InstructionSequence> result_iseq(new InstructionSequence());

  for (auto i = orig_bb->cbegin(); i != orig_bb->cend(); ++i) {
    Instruction *orig_ins = *i;

    // ...
    // Determine live vregs after instruction executes
    LiveVregs::FactType fact
      m_live_vregs.get_fact_after_instruction(orig_ins);
    // ...

    result_iseq->append(/* transformed instruction */);
  }

  return result_iseq;
}
```

For liveness on low-level code

`LiveMregs` computes liveness for machine registers. It works the same way as `LiveVregs`, except that the dataflow facts are bitsets of machine registers containing live values.

- The bitset values are the ordinal values of members of the `MachineReg` enumeration (i.e., `MREG_RAX`, etc.)

For further details...

See Assignment 5 for more details