Lecture 19: Liveness analysis

David Hovemeyer

November 11, 2024

601.428/628 Compilers and Interpreters





- Dataflow analysis
- Liveness analysis
- InstructionSequence and ControlFlowGraph

◆□▶ ◆□▶ ◆ □▶ ◆ □▶ ● □ ● ● ●

Dataflow Analysis

◆□ > ◆□ > ◆ 三 > ◆ 三 > ● ○ ○ ○ ○

From last lecture:

Scope Can analyze/transform	Example
Local Single basic blocks	Local value numbering
Regional Multiple BBs w/o control	joins Super LVN
Global All basic blocks within a fi	unction Liveness analysis
Interprocedural All code in the program	Function inlining

Liveness analysis (our topic today) is a global analysis

It's an instance of *dataflow analysis*, an important general technique for global analysis

Goal of dataflow analysis: determine a fact at each location within a function

A dataflow fact is often a set

• E.g., set of registers currently containing a live value

A dataflow analysis is executed on a control-flow graph (graph of basic blocks)

The general idea is that the dataflow analysis models

- the effect of the basic block on a dataflow fact flowing through the basic block, taking into account the semantics of the instructions in the block
- the effect of control paths coming together, where it is possible for a location to be reached from multiple predecessor blocks

A dataflow analysis can be forwards or backwards

Liveness Analysis

◆□ > ◆□ > ◆ 三 > ◆ 三 > ● ○ ○ ○ ○

Which storage locations (e.g., virtual registers) contain a "live" value?

"Live" means "is used on some forward path"

Very useful!

- If we see a store to a location such that the value in that location is not live, we can eliminate the store (dead store elimination)
- If we're considering a transformation that eliminates a store to a location, it's only legal if the value being stored is dead (correctness requirement for peephole rewrites)
- If any locations other than function parameters contain live values at the entry to a function, then the function could use an uninitialized value

When a variable's storage is in memory, there could be one or more pointers pointing to it ("pointer aliasing").

That means that when a data value is stored to memory, it could potentially change the value of any memory location.

Conservative approach (always correct, but least precise):

 \ldots = *p; // make no assumptions about what value is loaded

*q = ...; // assume that any memory location might be written to

Because virtual registers aren't memory and don't have addresses, we don't need to worry about aliasing. Changing the value of one vreg can't change the value of any other vreg.

Each dataflow fact is a set of locations (i.e., vregs) containing live values.

LiveOut(n) is the set of locations containing live values at the end of block n

UEVar(n) is the set of locations containing a value that is definitely used in basic block n (i.e., it appears as a source operand prior to any instruction storing to it)

VarKill(n) is the set of locations for which an instruction in block *n* stores a value to the location

◆□▶ ◆□▶ ◆三▶ ◆三▶ 三三 のへぐ

Initially: for each block *n*, LiveOut(*n*) = \emptyset Iteratively:

 $LiveOut(n) = \bigcup_{m \in succ(n)} (UEVar(m) \cup (LiveOut(m) \cap VarKill(m)))$

The idea is that we start out by assuming that no vregs contain live values at the end of each block. However, for each successor m of block n

- any vreg that is definitely used in block *m* prior to being overwritten (UEVar(*m*)) is definitely live at the end of *n*
- any vreg that is live at the end of m and not in VarKill(m) is also live at the end of n

In general, we want to compute accurate liveness for block n's successors before we compute liveness for block n

However, the control flow graph can have cycles (e.g., if the function has a loop), so this isn't always possible

General strategy: repeatedly recompute liveness for all blocks until none of the facts change

► The LiveOut sets start out as empty; when they stop growing, we're done

▲□▶ ▲□▶ ▲ 三▶ ▲ 三▶ ▲ 三 ● ● ●

Algorithm

```
-- n is number of basic blocks
for (i = 0 \text{ to } n - 1)
     LiveOut(i) \leftarrow \emptyset
changed \leftarrow true
while changed
     changed \leftarrow false
     for (i = 0 \text{ to } n - 1)
           old \leftarrow LiveOut(i)
           recompute LiveOut(i)
           if old \neq LiveOut(i)
                 changed \leftarrow true
```

Note that the same fixed-point solution is reached regardless of the order in which blocks are recomputed, but some orders are better for efficiency (more about this when we fully cover dataflow analysis.)

```
// C code
int arrsum(int *a, int n) {
    int sum, i;
    sum = 0;
    for (i = 0; i < n; i = i + 1) {
        int elt;
        elt = a[i];
        if (elt % 2 == 0)
            sum = sum + a[i];
    }
    return sum;
}</pre>
```

▲□▶ ▲□▶ ▲ 三▶ ▲ 三▶ 三三 - のへぐ

// High-		10				sconv la	wr 25	wr13	
// IIIgii	Tevel COC	16				SCOIIV_IQ	VI 20,	VIIJ	
// (cont	inues on	right))			mul_q	vr26,	vr25,	\$4
	.globl ar	rrsum				add_q	vr27,	vr10,	vr26
arrsum:						mov_l	vr29,	(vr27))
	enter	\$0				add_l	vr28,	vr12,	vr29
	mov_q	vr10,	vr1			mov_l	vr12,	vr28	
	mov_l	vr11,	vr2		.L2:				
	mov_l	vr12,	\$0			add_l	vr31,	vr13,	\$1
	mov_l	vr13,	\$0			mov_l	vr13,	vr31	
	jmp	.L1			.L1:				
.L0:						cmplt_l	vr32,	vr13,	vr11
	sconv_lq	vr17,	vr13			cjmp_t	vr32,	.LO	
	mul_q	vr18,	vr17,	\$4		mov_l	vr0, v	r12	
	add_q	vr19,	vr10,	vr18		jmp	.Larrs	sum_ret	urn
	mov_l	vr20,	(vr19)		.Larrsur	_return:			
	mod_l	vr22,	vr20,	\$2		leave	\$0		
	cmpeq_1	vr24,	vr22,	\$0		ret			
	cjmp_f	vr24,	.L2						

◆□ ▶ ◆□ ▶ ◆ 臣 ▶ ◆ 臣 ▶ ○ 臣 ○ のへぐ



▲□▶ ▲□▶ ▲ 三▶ ▲ 三▶ 三三 - のへぐ



◆□▶ ◆□▶ ◆三▶ ◆三▶ 三三 - のへで



◆□ ▶ ◆□ ▶ ◆ □ ▶ ◆ □ ▶ ○ □ ○ ○ ○ ○



◆□ > ◆□ > ◆豆 > ◆豆 > ̄豆 = のへで



▲□▶ ▲□▶ ▲□▶ ▲□▶ ▲□ ● ● ●



▲□▶ ▲□▶ ▲□▶ ▲□▶ ▲□ ● ● ●



・ロト・日本・日本・日本・日本・日本





◆□ > ◆□ > ◆豆 > ◆豆 > ̄豆 = のへで



◆□ > ◆□ > ◆豆 > ◆豆 > ̄豆 = のへで



◆□▶ ◆□▶ ◆三▶ ◆三▶ 三三 - 釣�?



・ロト・日本・日本・日本・日本・日本



・ロト・日本・日本・日本・日本・日本



▲□▶ ▲□▶ ▲□▶ ▲□▶ ▲□ ● ● ●



・ロト・日本・日本・日本・日本・日本



・ロト・日本・日本・日本・日本・日本



▲□▶ ▲□▶ ▲□▶ ▲□▶ ▲□ ● ● ●



・ロト・日本・日本・日本・日本・日本



▲□▶ ▲圖▶ ▲≣▶ ▲≣▶ = 差 = 釣�?



▲□▶ ▲□▶ ▲□▶ ▲□▶ ▲□ ● ● ●



◆□ > ◆□ > ◆豆 > ◆豆 > ̄豆 = のへで



・ロト・日本・日本・日本・日本・日本



InstructionSequence and ControlFlowGraph

▲□▶ ▲□▶ ▲ 三▶ ▲ 三▶ 三三 - のへぐ

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

ControlFlowGraph: a graph of basic blocks

- A basic block 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

// High-level InstructionSequence to high-level CFG std::shared_ptr<InstructionSequence> hl_iseq = /* ... */ auto hl_cfg_builder = ::make_highlevel_cfg_builder(hl_iseq);

std::shared_ptr<ControlFlowGraph> hl_cfg = hl_cfg_builder.build();

// Low-level InstructionSequence to low-level CFG
std::shared_ptr<InstructionSequence> ll_iseq = /* ... */
auto ll_cfg_builder = ::make_lowlevel_cfg_builder(ll_iseq);
std::shared_ptr<ControlFlowGraph> ll_cfg = ll_cfg_builder.build();

▲□▶ ▲□▶ ▲□▶ ▲□▶ ▲□ ● ● ●

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

◆□▶ ◆□▶ ◆ 臣▶ ◆ 臣▶ 三臣 - ∽ � � �

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

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

```
class MyTransform : public ControlFlowGraphTransform {
  private:
    LiveVregs m_live_vregs;
    // ...other analyses if needed...
  public:
    MyTransform(std::shared_ptr<ControlFlowGraph> cfg);
```

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

MyTransform::MyTransform(std::shared_ptr<ControlFlowGraph cfg)</pre>

▲□▶ ▲□▶ ▲□▶ ▲□▶ ▲□ ● ● ●

: 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(std::shared_ptr<InstructionSequence> orig_bl
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;
}
```

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.)

▲□▶ ▲□▶ ▲□▶ ▲□▶ ▲□ ● ● ●

See Assignment 5 for more details

