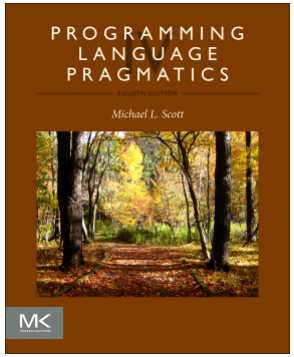
Code Optimization, Day 3: Dataflow Analysis

17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 17

Ben Titzer Prof. Jonathan Aldrich

Data Flow Analysis

- Dataflow analysis computes facts about a program at every point (e.g. basic block)
- A general framework can express many analyses:
  1. four sets for each basic block B, called In_B, Out_B, Gen_B, and $Kill_B$;
  2. values for the *Gen* and *Kill* sets;
  3. an equation relating the sets for any given block B;
  4. an equation relating the *Out* set of a given block to the *In* sets of its successors, or relating the *In* set of the block to the *Out* sets of its predecessors; and (often)
  5. certain initial conditions

Data Flow Analysis

- An example system of equations:

$$Out_B = Gen_B \cup (In_B \setminus Kill_B)$$

$$In_B = \bigcap_{\text{predecessors } A \text{ of } B} Out_A$$

- Dataflow analysis can be *forward* or *backward*
  - *forward:* Out sets are a function of the In sets
  - *backward*: In sets are a function of the Out sets
- The direction *suggests* (but does not require) a visitation order for the IR

Data Flow Analysis

- The goal of the analysis is to find a solution that represents a *fixed point* of the equations
- A fixed point is a set of *In* and *Out* sets that satisfy both the equations and the initial conditions
  - Some problems have a single fixed point
  - Others may have more than one
    - we usually want either the least or the greatest fixed point (smallest or largest sets)
  - Computing the fixed point solution efficiently requires careful datastructure design and the proper iteration order

Applying Dataflow analysis to Global Value Numbering

- SSA form construction used a forward dataflow analysis to rename variables and insert ϕ nodes
- With assignments to local variables eliminated, local value numbering can be extended to global value numbering through another dataflow analysis
  - As in local value numbering, the goal is to merge any virtual registers that are guaranteed to hold symbolically equivalent expressions
  - In the local case, we were able to perform a linear pass over the code
  - We kept a dictionary that mapped loaded and computed expressions to the names of virtual registers that contained them

Global Value Numbering

- A simple forward traversal does not suffice in the global case, because the code may have cycles and merges
  - We'll use a forward dataflow analysis to solve this
  - This algorithm works on SSA and non-SSA programs
  - It can also be obtained with a simpler algorithm that begins by unifying all expressions with the same top-level operator
    - In the end, repeatedly separates expressions whose operands are distinct
    - It is quite similar to the DFA minimization algorithm of Chapter 2
- We perform this analysis for our running example informally

Global Redundancy and Data Flow Analysis

- We will use a forward dataflow analysis where the sets In and Out contain expressions

$$Out_B = Gen_B \cup (In_B \setminus Kill_B)$$

$$In_B = \bigcap_{\text{predecessors } A \text{ of } B} Out_A$$

- Our initial condition is $In_1 = \varnothing$: no expressions are available at the beginning of execution

Global Redundancy and Data Flow Analysis

- For a block *B*
  - In_B is the set of expressions guaranteed to be available at the beginning of B
  - Out_B is the set of expressions guaranteed to be available at the end of *B*
  - $Kill_B$ is the set of expressions *killed* in *B*: invalidated by an instruction in *B*, and not subsequently recalculated
    - For non-SSA programs, an assignment to a local invalidates
    - For programs with load/store, stores to memory may invalidate
  - Gen_B is the set of expressions calculated in *B* and not subsequently killed in *B*

Global Redundancy and Data Flow Analysis

- Available expression analysis is known as a *forward* data flow problem, because information flows forward across branches: the *In* set of a block depends on the *Out* sets of its predecessors
  - We will see an example of a *backward* data flow problem later
- We calculate the desired fixed point of our equations in an inductive (iterative) fashion, much as we computed first and follow sets in Chapter 2
- Our equation for In_B uses intersection to insist that an expression be available on all paths into B
  - In our iterative algorithm, this means that In_B can only shrink with subsequent iterations

Global Redundancy and Data Flow Analysis

- We turn our attention to *live variable analysis* -very important in any subroutine in which global common subexpression analysis has eliminated load instructions
- Live variable analysis is a *backward* flow problem
- It determines which instructions produce values that will be needed in the future, allowing us to eliminate *dead* (useless) instructions
  - in our example we consider only values written to memory and with the elimination of dead stores
  - applied to values in virtual registers as well, live variable analysis can help to identify other dead instructions

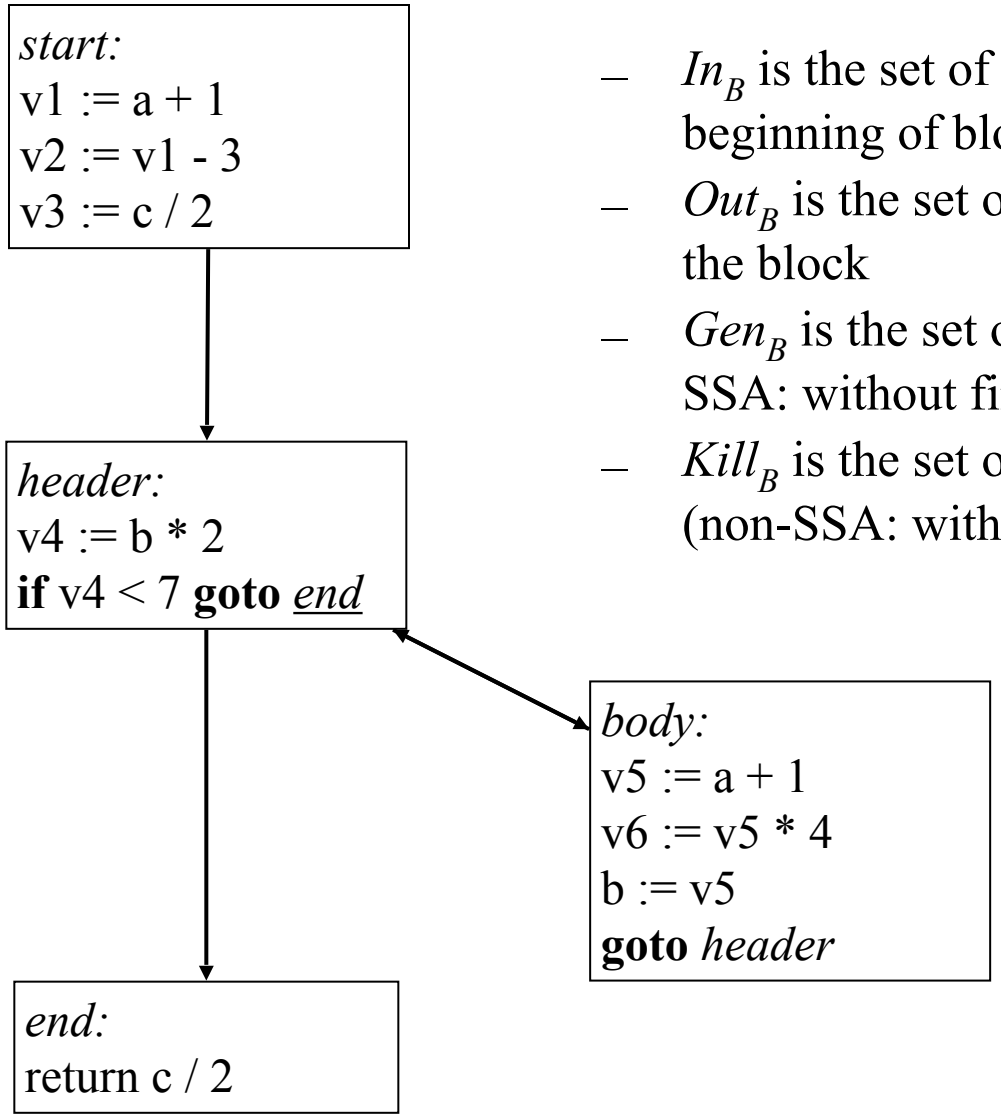
Global Redundancy and Data Flow Analysis

- For this instance of data flow analysis
  - In_B is the set of variables live at the beginning of block B
  - Out_B is the set of variables live at the end of the block
  - Gen_B is the set of variables used in B (non-SSA: without first being defined in B)
  - $Kill_B$ is the set of variables defined in B (non-SSA: without having been used first)
- The data flow equations are:

$$In_B = Gen_B \cup (Out_B \setminus Kill_B)$$

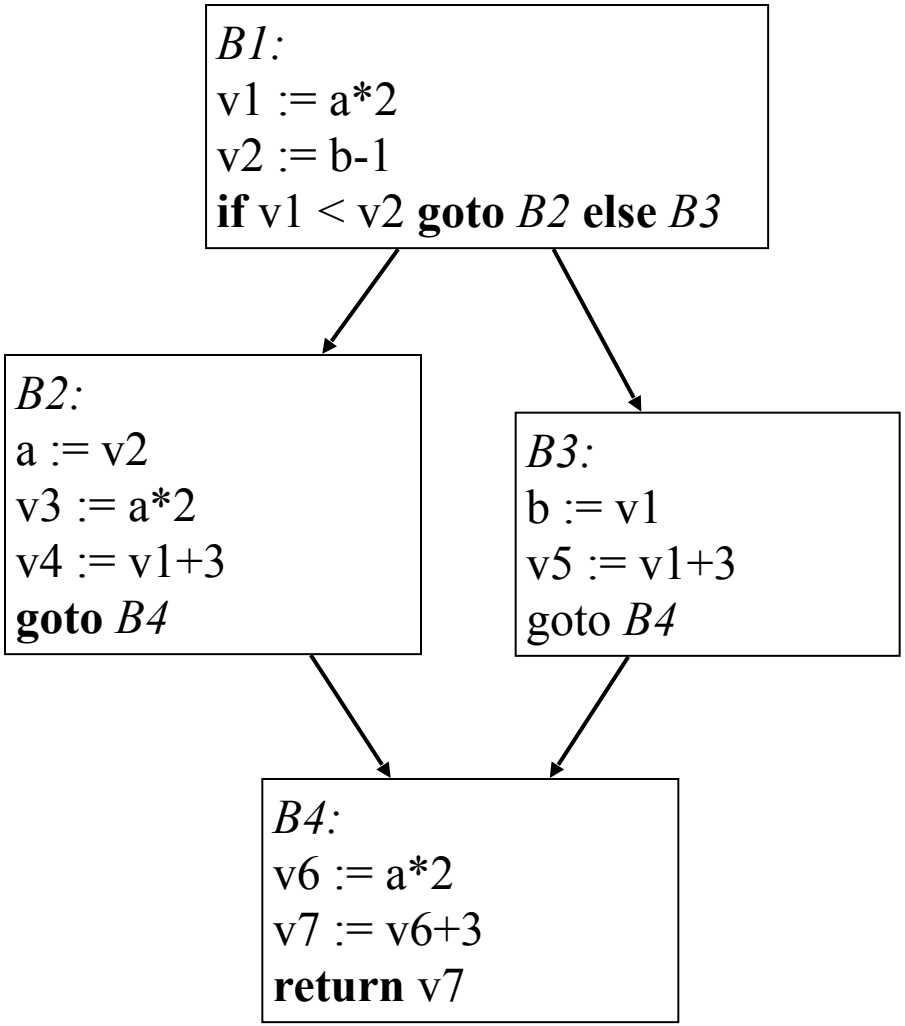
$$Out_B = \bigcup_{\text{successors } C \text{ of } B} In_C$$

Running live variable analysis and dead code elimination

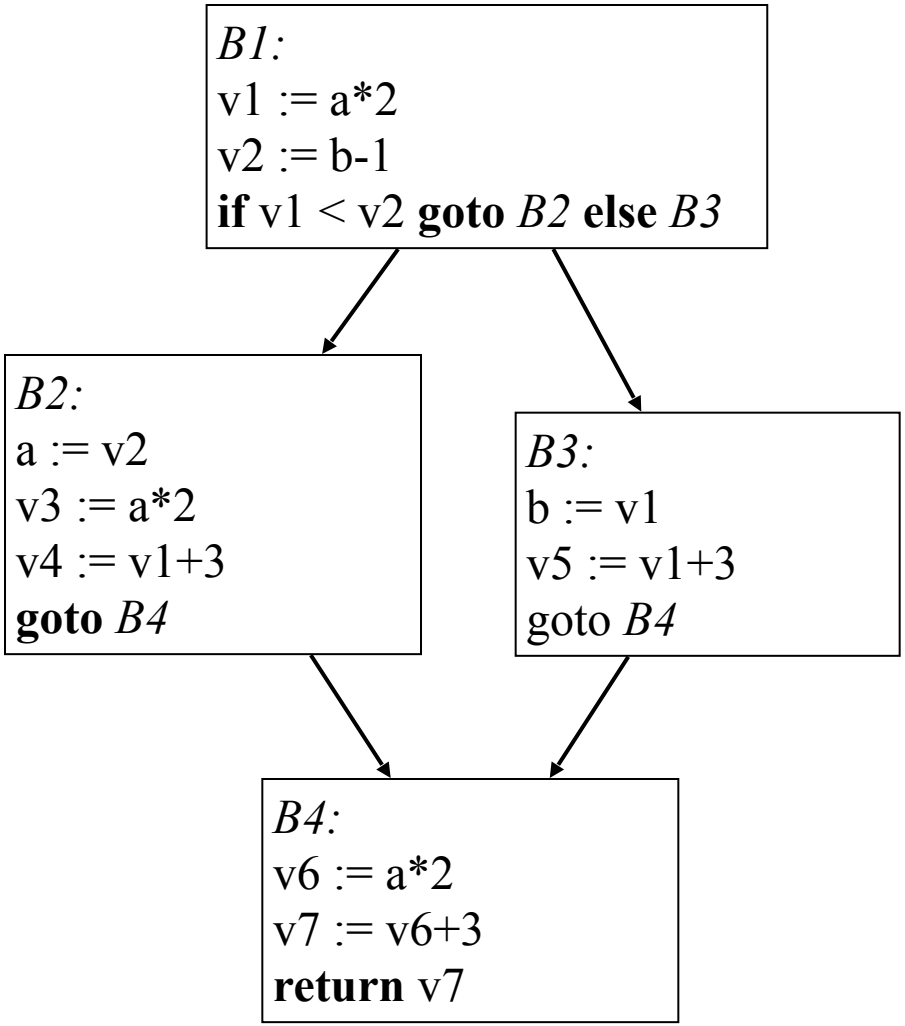


- In_B is the set of variables live at the beginning of block B
- Out_B is the set of variables live at the end of the block
- Gen_B is the set of variables used in B (non-SSA: without first being defined in B)
- $Kill_B$ is the set of variables defined in B (non-SSA: without having been used first)

Exercise: Apply live variable analysis and dead code elimination to this program



Exercise: Apply live variable analysis and dead code elimination to this program



	IN	GEN	KILL	OUT
B1	a b	a b	v1 v2	v1 v2 a
B2	v1 v2	v1 v2	v3 v4 a	a
B3	v1 a	v1	v5 b	a
B4	a	a	v6 v7	

ELSEVIER

Register Allocation

- Program IR has an unlimited number of variable names, often called “virtual registers”
- Real machines have a fixed set, often 8, 16, or 32 of *physical* or *architectural* registers
- May be different sets for float and integer
- Registers are the fastest storage of the entire computer system => important to use effectively
- How do we map the unbounded set of virtual registers onto a finite set?
  - Answer: cannot, in general, must use some stack
  - When to use stack, and which register to use?
    - Program analysis!

Applying liveness analysis to Register Allocation

- Observation: liveness analysis computes the set of live variables at the beginning and end of every basic block
- What we need: set of live variables at *every* program point
- Variables that are not live at any of the same program points can use the same register
- Otherwise we say they *interfere*
- Computing the set of interferences allows us to allocate registers to variables efficiently

Approaches to Register Allocation

- Local: only use registers within a block
- Linear scan: simplify control flow graph to a line, obtain a set of intervals that are relatively easy to assign to registers (almost same as local)
- Graph coloring: use *interference graph* directly to compute a more efficient solution
  - Interference graph represents all conflicts between variables that cannot use the same register
  - Graph-coloring is an NP-hard problem
  - Nearly all compilers use a *coloring heuristic* for the graph
- In any case, if allocation fails, resort to *spilling*

Approaches to Register Allocation

- Graph coloring: use *interference graph* directly to compute a more efficient solution
  - Build interference graph from liveness analysis
  - Nodes represent variables
  - (Undirected edges represent vertices)
  - Assign K (number of registers) or fewer colors to vertices
  - Simplification-based heuristic: remove nodes < K degree onto a stack
  - Pop nodes in reverse order, rebuilding the graph and giving each a color that differs from its neighbors
  - Spill if not enough colors