Compilers grab bag

•... this was intended to focus on whole-program analysis, but turned into a collection of all the topics I wish we'd had more time for ...

- whole program analysis (revisited)
- code generation (revisited)
- scalar optimizations
- instruction selection
- instruction scheduling
- register allocation

Whole-program analysis

- largely dependent on call-graph construction
- call-graph complicated by higher order functions:
  - determining which functions could be called by the higher order function requires knowledge of which functions could be passed as parameters
  - further complicated if functions/function references can also be stored in variables
  - not unlike the pointers/ambiguity issues discussed earlier, compiler may assume anything passable needs to be included in the call graph

Other whole-program concerns

- OO related issue: object initialization, cleanup, garbage collection, dynamic binding
- recognition/propagation of constant-valued function parameters and return values
- dataflow analysis techniques beyond the iterative ones
- faster/more accurate dominance computation

Code generation (revisit)

- mapping each SSA sequence into target language
- typically build/maintain tables with characteristics of target language/architecture
  - registers available, naming scheme, sizes
  - memory alignment sizes/rules,
  - procedure call conventions
  - specific instruction details
- need to pick the set of instructions and schedule an order for them, including allocation of which registers they use
- picking instructions, scheduling, and register allocation are all computationally hard problems

Scalar optimizations

- eliminating useless code: mark and sweep algorithm
  - mark all useless
  - sweep through, mark useful if i/o, affects return value, or affects storage accessible outside routine
  - trace all useful items back to their sources (and their sources and so on) marking them useful
  - remove anything still listed as uselss
- eliminating useless code II
  - unreachable code (e.g. after a jump but before a label)
  - empty blocks
  - redundant blocks

Instruction selection

- during tree-walk, map names to physical storage, apply loads/stores as needed, map SSA ops to target op(s)
- try pattern matching on subtrees: map specific subtree patterns to target language patterns, allows recognition of larger-scale replacements
- use tables of rewrite rules detailing how to take a subtree pattern and replace it with target language pattern
- searching for a “good” set of pattern matches (and hence rewrites) to cover entire tree
- follow with peephole optimization: sliding window looking at small sequence of instructions, seeking better rewrites

Instruction scheduling

- overlap instruction execution where possible, pipeline fetching of operands and instructions where possible
- sometimes must wait for completion of an operation, or to abandon pipelined operands/instructions due to result of another calculation or branch
- ideally, instruction scheduler picks an instruction sequence that minimizes the frequency of such delays
- work off partial ordering for instruction sequences (data dependencies), first applied strictly within the basic blocks, then to extended basic blocks (like with value numbering)

Register allocation

- registers are fastest location in memory
- often the only location accessible for some operations/operands
- effective register allocation crucial for performance
- take code that uses N registers and rewrite it to use M (presumably M being the number available on the target machine)
- decide which values are safe for registers (deal with ambiguity), pick which give greatest benefits
- might have to map to multiple pools of registers (e.g. general purpose vs floating point)

Allocation and assignment

- allocator might split task into two parts:
  - allocation: taking the N input names and mapping them into M new names (reducing total number of names used to the desired level)

    [this is the harder of the two problems: NPc vs poly]
  - assignment: mapping the specific M new names to actual registers in the target language

Local allocation/assignment

- applied within a basic block
  - M total registers available, k of those set aside for handling values being loaded/stored from/to memory
- top down approach
  - count uses of each variable in block
  - put most frequent (M-k) vars in registers, load/store others when needed
- bottom up approach
  - track sequence of upcoming value uses
  - fill available registers in sequence
  - on running out of registers, “spill” the value that won't be needed for the longest time in the future

Subroutine allocation/assignment

- trying to minimize overall impact of spills
- much more challenging than just within a simple block, trying to resolve cross-block issues
- actual costs/benefits depend on which blocks actually run during execution, in which order, and how often
- live range: set of definitions and their uses that are used together across a sequence of blocks, give a measure of which values may belong together in registers

Graph colouring and registers

- can k-colour a graph if we can colour the nodes on the graph so that no two like-coloured nodes share an edge
- interference graph models the values in a program as nodes, with edges between them if they “need” to be in registers at the same time
- if we can k-colour the interference graph then we can satisfy the constraints using k registers
- if we can't colour the graph with the number of registers actually available, then we'll need to spill some values to memory and try again ... looking for low cost spills