NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4205 Grand Summary, pt. 2

Process images

- Detailed image is in lecture note on assembly (*x86_64 Assembly language*)
- Most of the detail level is relevant first at the system level, but high IR can well be planned with the aim of being simple to translate into it.

Stack machine exec. model

- In order to lay out the process image, some model of execution must be assumed
- Most common is some variation on a stack machine
  - Parameters spill on stack
  - Locally scoped variables go on stack
- Other things are handled by address/dereference
  - Addresses of functions
  - Addresses of globals and heap

Stack frames

- This introduces the need for *activation records*, which are the layouts of stack frames
- Contain (at least)
  - Reference to caller (return pointer)
  - Reference to stack frame of caller
  - Local variables
- The exact choice of contents and their use are subject to convention
  - Ours: caller saves registers, callee takes care of frame, returns result in register
  - This responsibility can be divided differently, with different implications for the generated code

High IR

- Closely corresponds to source program
- Typically contains annotated syntax tree (+ any other information needed for particular language constructs)
- Still discards parts of the source which are only of syntactic value, reducing the syntax tree to an *abstract* one
- The abstract part just means we've thrown away all the now-redundant information which was an artifact of the grammar and scanning

IR lowering

- Semantics-preserving transformations on the high IR take out redundant information / admit high-level optimizations
  - Different loop constructs can all be translated to one format
  - “unless” is just “if” in a different guise
  - Many other forms of *syntactic sugar* exist purely for the benefit of source program prettiness, and vanish in translation
- The constructs of the resulting representation each need to be associated with a translation scheme from high to low IR
- Choice of low-level IR is open to what is convenient
  - *Three Address Code (TAC)* is what we used in lectures
  - Practical work didn't really use a low-level IR, went straight to assembly

TAC

- TAC represents instructions as
  - Basic operations
  - At most two operands
  - At most one target
- Encodable as quadruples
- Essentially, a high-level assembly
  - Ifs, but label/jump for loops
  - Function calls become sequence of parameters + actual call
  - No concrete memory layout concerns yet
  - All values are variables, as many as needed

Lowering scheme

- Each type of construct left in the high-IR needs a rule
- Expressions work by recursive translation of two-operand operations, creating the overall expression as string of three-adr. Ops
- Loops transform to conditionals and jumps
- Output marked by effects of the translation scheme
  - Lots of temporary variables
  - Parts of a single high-level construct may now be scattered throughout the sequence of operations

Optimizations

- Optimization in the back end applies to all front ends attachable to it
  - May benefit multiple source languages
  - Harder to detect what can be done
- Some optimizations apply to high IR, some to low, some to both, some open possibilities for each other
- We went through a long list just to have a context for further discussion

The long list

- Function inlining
- Function cloning
- Constant folding
- Constant propagation
- Dead code elimination
- Loop-invariant code motion
- Common sub-expression elimination
- Strength reduction
- Loop unrolling

Control flow graphs

- Divide programs in basic blocks
- Edges follow from conditionals, jumps
- Basic block has no diverging paths
- Separates effects of
  - Basic blocks themselves
  - Control paths through the program

Data flow analysis

- Analyses are instances of general framework which works in terms of
  - Partially ordered set
  - Meet operator

  where the set and order are chosen to reflect the information at *program points* (found before/after instructions, basic blocks)
- General worklist algorithm guarantees Maximal Fixed Point (MFP) solution if transfer function is monotone
- *Monotone (monotonic)* means it will only take a program point to a less precise state in the *lattice* of possible states

Data flow analysis

- *Meet-over-paths* (MOP) solution is solution given by following every possible path, applying the transfer function to the chosen blocks separately'
- MFP is as precise as MOP when transfer function is distributive
  - i.e. applying it to the union of two paths is the same as applying it to both paths and unifying the result

Data flow analysis instances

- Live Variables
- Reaching Definitions
- Available Expressions
- Constant Folding
- Dominator relation

(...and copy propagation...)

Live Variables

- Determines whether a variable can possibly used later
- Least precise: “all vars can be used later”
- Partial order is set inclusion, empty set is top
- Transfer function takes
  - All variables live at the beginning of a block must be live at the end of all its predecessors
  - All variables used are live at input
  - All variables defined are not live before
- Starts at end of program with no live variables at program points, works backwards adding them
- Distributive meet/transfer

Available Expressions

- Works on sets of numbered expressions
- More expressions available = more precise
- Starts from top, adding expressions
- At meeting points, works with intersection, so that expressions which are available have been made available by every path to here
- Blocks add expressions they evaluate, remove expressions which include variables changed in the block
- Works forward, distributive meet/transfer

Reaching Definitions

- Works on sets of numbered assignments, determines which of them can be in effect at a later point
- Meet is union, so that all assignments which *may* have been made (on one path or the other) are included
- More precise when fewer definitions are possible
- Blocks remove definitions at (re)assignments, add own assignments instead
- Works forward, distributive meet/transfer

Constant Folding

- Lattice is product of variables and natural numbers
- Top is “unknown constant-ness” (not very informative, but every variable which gets a value at some point will come down from there)
- Middle is “known to be constant, value is *n*”
- Bottom is “known not to be constant”
- Forward analysis
- Meet/transfer not distributive

Dominator relation

- Works on set of basic blocks in control graph
  - Really speaks about control flow, just does it in terms of data
- Forward, control flow is only interesting part
- Meet is intersection: what dominates all paths here dominate this block also
- Distributive meet/transfer
- Dominators organize into a tree by attaching children to *immediate dominator*
- Use is to detect back edges (loops) in control flow graphs:
  - Jumping to a block which dominates this one is a loop

Instruction selection by tiling

- Instructions can be chosen from fixed-cost *tiles*, which map sequences of instructions to a tree representation of code
- Tree representation can be compressed into *directed acyclic graph*, saving work on repeated expressions
- Idea is that having a choice of tiles permits selecting the least expensive overall combination
- Most relevant to CISC architectures, RISC has less choice in tile construction
- On modern hardware, tile cost is *very* approximate

Register allocation

- Graph coloring problem, each register is a color
- Construct interference graph of variables that are simultaneously live
- Infeasible to get optimal solution
- Approximation
  - Reduce interference graph to nothing
  - Reintroduce nodes and color optimistically

Moral from the back end

- Automatically detecting potential for optimization is
  - Tricky (because it is)
  - Conservative (because it has to be)
- Part of the point of going through the low level programming is so that
  - You can identify when (maybe, why) the compiler fails to help you with it
  - You can also do it yourself

Moral of the story

- As previously mentioned, few people go on to write a lot of compilers

  (Those who do find that it's rather more complicated still)

- As a compiler *user*, you have hopefully become a little bit more familiar with the instrument you operate
  - and where the precise meaning of the source language comes from

That's it from me

- At least in the auditorium, I'll still post things on It's Learning, answer emails, *etc.*
  - Do check in from time to time

- Thank you for your patience, effort, and participation