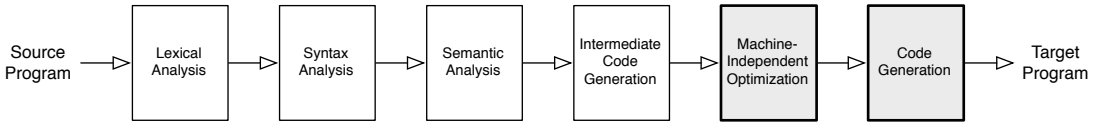
CS 301 Spring 2016
Meetings April 19

More Optimization and Analysis

Plan



This week, we wrap up discussion of data-flow analysis and begin work on the IC compiler's optimizer. Our chief goal is to prepare for implementing the IC data-flow engine and analyses by practicing with data-flow frameworks. On Friday, we reviewed and extend the theory of lattices and data-flow that we started to explore last week. In meetings on Tuesday (Monday Schedule), we will work through the remaining problems from last week, focusing on framing interesting analysis questions as data-flow frameworks and designing your IC analysis and optimization code. We will use any remaining time to consider alternative techniques for optimization and code generation (briefly, in the abstract).

Please prioritize data-flow (both the theory and the more concrete design for your IC data-flow code) and skim the other material as second priority. The core reading for this (Dragon 9.3 – 9.4) repeats from last week and is hopefully more accessible after our work on foundations this week. Read for reference while building your data-flow frameworks in the continued exercises from last week. Following the constant propagation example will be helpful when thinking about last week's later exercises.

The remaining readings (and the remainder of our meetings this semester) will take a higher-level view of several interesting issues. Read for big ideas as time permits. There is one final study on mechanics of basic register allocation (Exercise 5). From here on, we will not focus on details or mechanics as we will not implement these techniques in our planned project phases, though some make good final project extensions.

Readings

- From last week, Dragon 9.3 – 9.4: data-flow framework foundations and example (20 pages).
- Skim EC 9.2.3: limitations of data-flow analysis (2 pages).
- Skim EC 5.4.2: single static assignment (SSA) form (3 pages).
- EC 13.1 - 13.4.0: intro to register allocation (10 pages).
- Skim EC 11.1, 12.1: light overviews of smarter code generation schemes (2 pages, 4 pages).

Exercises

1. Start designing for the IC Optimizer:
   - How will you represent the data-flow facts for each analysis in your compiler?
   - How will you use data-flow to perform each optimization on a TAC list?
2. Exercises 3 – 6 from last week.
3. Think about *single static assignment* form. Does it make any code properties clearer than TAC?
4. Discuss project ideas.
5. (Adapted from EC 13 exercise 1.) Consider the following pseudo-x86 code. Unlike true x86, instructions may refer to two source registers and a distinct third destination register. Sources appear to the left of `=>`, destination to the right.

```
mov -8(%rbp)  => %v1       # load from Memory[%rbp -  8] and store in %v1
mov -16(%rbp) => %v2       # load from Memory[%rbp - 16] and store in %v2
add %i + %v1  => %v3       # add %i and %v1 and store in %v3
sub %v2 - %i  => %v4       # subtract %i from %v2 and store in %v4
mul %v3 * %v4 => %v5       # multiply %v3 and %v4 and store in %v5
mul %v2 * 2   => %v6       # multiply %v2 and 2 and store in %v6
add %v5 + %v6 => %v7       # add %v5 and %v6 and store in %v7
mov %v7       => -24(%rbp) # load from %v7 and store in Memory[%rbp - 24]
```

Registers `%v1`–`%v7` are virtual registers holding temporary values. Register `%i` is a virtual register holding the value of declared local variable `i`. Assume that the physical pseudo-x86 machine provides only five physical registers: `%r1`, `%r2`, `%r3`, `%r4`, and `%rbp`, which is dedicated to holding the base pointer for the current stack frame. All virtual registers must be replaced by physical registers. If there are too few registers, spill extra values into unused stack slots (starting at `-32(%rbp)`) as needed.

(a) Perform local top-down register allocation (see EC 13.3.1).

(b) Perform local bottom-up register allocation (see EC 13.3.2).

(c) How do these results change if we eliminate the physical `%r4`?

(d) Most instructions in the true x86 ISA use one source and one destination operand, combining the contents of source and destination with some operator and storing the result in destination. For example, `addq %rax, %rcx` adds the contents of `%rax` and `%rcx` and overwrites `%rcx` with the result. How does this"two-address code" style complicate register allocation vs. the "three-address" pseudo-x86 above?