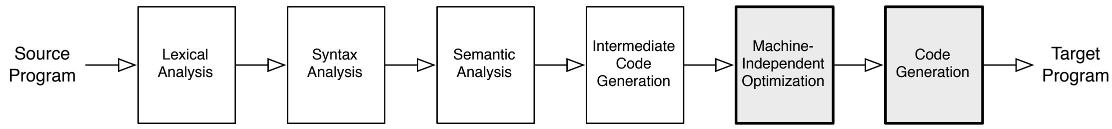
CS 301 Spring 2019
Tutorial Assignment
23 April

More Optimization, Runtime Systems

1 Plan



This week, we wrap up discussion of data-flow analysis by designing some data-flow analyses and sketching the implementation of your optimizer. The remaining readings and exercises consider: other strategies for optimization and non-trivial code generation; compiling and optimizing code at run time using information from real program executions; and implementing garbage collection. Aim for high-level insights – not detailed mechanics – in these parts.

2 Readings

- From last week, review Dragon 9.3–9.4: data-flow framework foundations and example
- Skim these overviews for a tasted of additional optimization and code generation topics:
  - EC 9.2.3: limitations of data-flow analysis
  - EC 5.4.2: single static assignment (SSA) form
  - EC 8.7–8.7.1: inlining
  - EC 11.1, 12.1: instruction selection and scheduling
  - EC 13.1–13.2.2: register allocation
- Dynamic Optimization:
  - *A Survey of Adaptive Optimization in Virtual Machines.* Matthew Arnold, et al. In Proceedings of the IEEE Vol. 93 Issue 2, February 2005.
    `http://www.ittc.ku.edu/~kulkarni/teaching/archieve/EECS800-Spring-2008/survey_adaptive_optimization.pdf`
- Garbage Collection:
  - *Uniprocessor Garbage Collection Techniques.* Paul Wilson.
    `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.2438&rep=rep1&type=pdf`
    Read sections 1–2, 3–3.2, 4.

More fun:

- *21 compilers and 3 orders of magnitude in 60 minutes: a wander through a weird landscape to the heart of compilation.* Graydon Hoare. 2019.
  `http://venge.net/graydon/talks/CompilerTalk-2019.pdf`
  Slides from a talk – will not take 60 minutes to read. Fun, fascinating, and meandering tour of *real* compilers and history.
- *A Catalogue of Optimizing Transformations.* Frances E. Allen and John Cocke. 1971.
  `https://www.clear.rice.edu/comp512/Lectures/Papers/1971-allen-catalog.pdf`

  Frances Allen won the Turing Award in 2006 (`https://amturing.acm.org/award_winners/allen_1012327.cfm`) for inventing the bedrock of modern compiler optimizations. These are still the most important ones, decades after this 1971 paper.

3 Exercises

1. Start designing for your optimizer implementation:

   (a) How will you represent data-flow facts for each analysis in your compiler?

   (b) How will you use data-flow to perform each optimization on a TAC list?

   Consider:

   - Live Variable Analysis / Dead Code Elimination
   - Constant Folding Analysis / Constant Folding
   - Available Expressions Analysis / Common Subexpression Elimination
   - Reaching Copies Analysis (a refinement of Reaching Definitions) / Copy Propagation

2. Exercises 5 (null check elimination) and 6 (arrays bounds check elimination) from last week if you did not cover them in tutorial.

3. Other static optimization strategies:

   (a) Does *single static assignment* (SSA) form make any code properties clearer than TAC? How? How might this affect the ease of optimizations?

   (b) Give some examples of when/how *inlining* would be beneficial and when/how it could be detrimental to code speed and size.

   (c) How are instruction selection, instruction scheduling, or register allocation inter-related?

4. Dynamic optimizations:

   (a) What are the upsides and downside of optimizing code at run time? Think about this at some length. Can run-time optimization clear some of the limitations of compile-time optimization? Is it worth the cost? Consider the ideas of profile-guided optimization, feedback-guided optimization, adaptive optimization, etc. (All closely related.)

   (b) Identify several opportunites for run-time optimization for a language like Java or Roost. (First, think back to a paper we read before spring break on Polymorphic Inline Caches.)

   (c) How do optimization opportunities differ based on properties of the source language in use? Pick a few languages familiar to you, e.g., C, Java, Scala, SML, Javascript, Python, Racket, Roost.

5. Garbage collection:

   (a) What is a limitation that applies to reference-counting, mark-sweep, *and* copying garbage collection?

   (b) What is a problem in both reference-counting and mark-sweep garbage collection that is addressed by copying collection?

   (c) What is one limitation of reference-counting that is not a problem for mark-sweep garbage collection?

   (d) What is the point of incremental garbage collection?

   (e) What is the key expected behavior of programs for which generational collection is optimized? (This is also called the *generational hypothesis*.) How does generational collection optimize for this behavior?