CSE 501:
Implementation of Programming Languages

Main focus: **program analysis and transformation**
- how to represent programs?
- how to analyze programs? what to analyze?
- how to transform programs? what transformations to apply?
Study imperative, functional, and object-oriented languages

Prerequisites:
- CSE 401 or equivalent
- CSE 505 or equivalent

Reading:
- Appel’s “Modern Compiler Implementation”
  + ~20 papers from literature
- “Compilers: Principles, Techniques, & Tools”, a.k.a. the Dragon Book, as a reference

Coursework:
- periodic homework assignments
- major course project
- midterm + final

---

Course outline

Models of compilation/analysis

Standard optimizing transformations

Basic representations and analyses

Fancier representations and analyses

Interprocedural representations, analyses, and transformations
  - for imperative, functional, and OO languages

Compiler back-end issues
  - register allocation
  - instruction scheduling

Run-time system issues
  - garbage collection
  - compiling dynamic dispatch, first-class functions, ...

Dynamic (JIT) compilation

---

Why study compilers?

Meeting area of programming languages, architectures
  - capabilities of compilers greatly influences design of these others

Program representation, analysis, and transformation is widely useful beyond this “traditional” task
  - software engineering tools
  - DB query optimizers
  - programmable graphics renderers
  - safety checking of code, e.g., in programmable/extensible systems, networks, databases

Cool theoretical aspects, too
  - lattice domains, graph algorithms, computability/complexity

Goals for language implementation

Correctness

Efficiency
  - of: time, data space, code space
  - at: compile-time, run-time

Support expressive, safe language features
  - first-class, higher-order functions
  - method dispatching
  - exceptions, continuations
  - reflection, dynamic code loading
  - bounds-checked arrays, ...
  - garbage collection
  - ...

Support desirable programming environment features
  - fast turnaround
  - separate compilation, shared libraries
  - source-level debugging
  - profiling
  - ...

Idea: compile to a portable intermediate language

Define “portable” intermediate language
(e.g. Java bytecode, MSIL, SUIF, WIL, C, ...)

 Compile multiple languages into it
• each such compiler may not be much more than a front-end

 Compile to multiple targets from it
• may not be much more than back-end

 Maybe interpret/execute directly

 Advantages:
• reuse of front-ends and back-end
• portable “compiled” code

 BUT: design of portable intermediate language is hard
• how universal?
  across input language models? target machine models?
• fast interpretation and simple compilation at odds

Key questions

How are programs represented in the compiler?

How are analyses organized/structured?
Over what region of the program are analyses performed?
What analysis algorithms are used?

What kinds of optimizations can be performed?
Which are profitable in practice?
How should analyses/optimizations be sequenced/combined?

How best to compile in face of:
• pointers, arrays
• first-class functions
• inheritance & message passing
• parallel target machines

Other issues:
• speeding compilation
• making compilers portable, table-driven
• supporting tools like debuggers, profilers, garbage collect’rs

Overview of optimizations

First analyze program to learn things about it
Then transform the program based on info
Repeat...

Requirement: don’t change the semantics!
• transform input program into
  semantically equivalent but better output program

Analysis determines when transformations are:
• legal
• profitable

Caveat: “optimize” a misnomer
• result is almost never optimal
• sometimes slow down some programs on some inputs (although hope to speed up most programs on most inputs)
Semantics

Exactly what are the semantics that are to be preserved?

Subtleties:
- evaluation order
- arithmetic properties like associativity, commutativity
- behavior in “error” cases

Some languages very precise
- programmers always know what they’re getting

Others weaker
- allow better performance (but how much?)

Semantics selected by compiler option?

Scope of analysis

Peephole: across a small number of “adjacent” instructions
- [adjacent in space or time]
- trivial analysis

Local: within a basic block
- simple, fast analysis

Intraprocedural (a.k.a. global):
- across basic blocks, within a procedure
- analysis more complex:
  - branches, merges, loops

Interprocedural:
- across procedures, within a whole program
- analysis even more complex:
  - calls, returns
- sometimes useful
- more useful for higher-level languages
- hard with separate compilation

Whole-program:
- analysis examines whole program in order to prove safety

A tour of common optimizations/transformations

arithmetic simplifications:
- constant folding
  \[ x := 3 + 4 \Rightarrow x := 7 \]
- strength reduction
  \[ x := y + 4 \Rightarrow x := y \ll 2 \]

constant propagation
\[ x := 5 \Rightarrow x := 5 \Rightarrow x := 5 \]
\[ y := x + 2 \quad y := 5 + 2 \quad y := 7 \]

integer range analysis
- fold comparisons based on range analysis
- eliminate unreachable code

```java
for(index = 0; index < 10; index ++) {
    if index >= 10 goto _error
    a[index] := 0
}
```
- more generally, symbolic assertion analysis

copy propagation
\[ x := y \Rightarrow x := y \]
\[ w := w + x \quad w := w + y \]

common subexpression elimination (CSE)
\[ x := a + b \Rightarrow x := a + b \]
\[ y := a + b \quad y := x \]
- can also eliminate redundant memory references, branch tests

partial redundancy elimination (PRE)
- like CSE, but with earlier expression only available along subset of possible paths

```java
if ... then \Rightarrow if ... then
... ...
  x := a + b \quad t := a + b; x := t
end \quad else t := a + b end
... ...
  y := a + b \quad y := t
```
pointer/alias analysis

\[ p := \&x \Rightarrow p := \&x \Rightarrow p := \&x \]
\[ *p := 5 \quad *p := 5 \quad *p := 5 \]
\[ y := x + 1 \quad y := 5 + 1 \quad y := 6 \]
\[ x := 5 \]
\[ *p := 3 \]
\[ y := x + 1 \Rightarrow \cdots \]

dead (unused) assignment elimination

\[ x := y \ast \ast \ast \]
\[ \ldots \quad // \quad no \ use \ of \ x \]
\[ x := 6 \]

- a common clean-up after other optimizations:
\[ x := y \Rightarrow x := y \Rightarrow x := y \]
\[ w := w + x \quad w := w + y \Rightarrow w := w + y \]
\[ \ldots \quad // \quad no \ use \ of \ x \]

partial dead assignment elimination
- like DAE, except assignment only used on some later paths

dead (unreachable) code elimination

- another common clean-up after other optimizations

parallelization

\[ \text{for } i := 1 \text{ to } 1000 \Rightarrow \text{for all } i := 1 \text{ to } 1000 \]
\[ a[i] := a[i] + 1 \quad a[i] := a[i] + 1 \]

loop interchange, skewing, reversal, ...

blocking/tiling
- restructuring loops for better data cache locality

loop-invariant code motion

\[ \text{for } j := 1 \text{ to } 10 \Rightarrow \text{for } j := 1 \text{ to } 10 \]
\[ \text{for } i := 1 \text{ to } 10 \quad t := b[j] \quad \text{for } i := 1 \text{ to } 10 \]
\[ a[i] := a[i] + b[j] \quad a[i] := a[i] + t \]

induction variable elimination

\[ \text{for } i := 1 \text{ to } 10 \Rightarrow \text{for } p := \&a[1] \text{ to } \&a[10] \]
\[ a[i] := a[i] + 1 \quad *p := *p + 1 \]
- \( a[i] \) is several instructions, *p is one

loop unrolling

\[ \text{for } i := 1 \text{ to } N \Rightarrow \text{for } i := 1 \text{ to } N \text{ by } 4 \]
Optimization themes

Don’t compute it if you don’t have to
  • dead assignment elimination

Compute it at compile-time if you can
  • constant folding, loop unrolling, inlining

Compute it as few times as possible
  • CSE, PRE, PDE, loop-invariant code motion

Compute it as cheaply as possible
  • strength reduction, induction var. elimination, parallelization, register allocation, scheduling

Enable other optimizations
  • constant & copy propagation, pointer analysis

Compute it with as little code space as possible
  • dead code elimination

The phase ordering problem

Typically, want to perform a number of optimizations; in what order should the transformations be performed?

some optimizations create opportunities for other optimizations
  ⇒ order optimizations using this dependence
  • some optimizations simplified
    if can assume another opt will run later & “clean up”

but what about cyclic dependences?
  • e.g. constant folding ⇒ constant propagation

what about adverse interactions?
  • e.g.
    common subexpression elimination ⇒ register allocation
  • e.g.
    register allocation ⇒ instruction scheduling
Compilation models

Separate compilation
• compile source files independently
• trivial link, load, run stages
  + quick recompilation after program changes
    – poor interprocedural optimization

Link-time compilation
• delay bulk of compilation until link-time
• then perform whole-program optimizations
  + allow interprocedural & whole-program optimizations
  – quick recompilation? shared precompiled libraries?
Examples: Vortex, some research optimizers/parallelizers, ...

Run-time compilation (a.k.a. dynamic, just-in-time compilation)
• delay bulk of compilation until run-time
• can perform whole-program optimizations + optimizations
  based on run-time program state, execution environment
  + best optimization potential
  + can handle run-time changes/extensions to the program
  – severe pressure to limit run-time compilation overhead
Examples: Java JITs, Dynamo, FX-32, Transmeta

Selective run-time compilation
• choose what part of compilation to delay to run-time
  + can balance compile-time/benefit trade-offs
Example: DyC

Hybrids of all the above
• spread compilation arbitrarily across stages
  + all the advantages, and none of the disadvantages!!
Example: Whirlwind

Engineering

Building a compiler is an engineering activity
• balance
  complexity of implementation,
  speed-up of “typical” programs,
  compilation speed,
  ...

Near infinite number of special cases for optimization
  can be identified
• can’t implement them all

Good compiler design, like good language design, seeks
small set of powerful, general analyses and transformations,
to minimize implementation complexity while
maximizing effectiveness
• reality isn’t always this pure...