CS 536 Announcements for Thursday, May 5, 2022

Course evaluation – log into aefis.wisc.edu using your NetID

Last Time
- optimization overview
- peephole optimization
- loop optimizations
- copy propagation

Today
- wrap up optimization
  - copy propagation
- wrap up course / review

Optimization Review

Goal: Produce "better" code that does the "same thing" as the original code.
- better = faster code, fewer instructions

When?
- before code generation (i.e., on intermediate representation)
- after code generation (i.e., on generated machine code)

Important considerations
- **performance/profitability** – want to be sure optimization is "worth it"
- **safety** – original source code, non-optimized target code, and optimized target code all do the "same thing" / have the same "meaning"

Look at optimizations that
- are **sound** transformations
- recognize a behavior in a program & replace it with a "better" version
Copy propagation

**Idea:** Suppose we are at use U of x and a definition D of x (of the form \( x = y \)) reaches U
- If
  1) no other definition of x reaches U and
  2) y does not change between D and U
- then we can replace the use of x at U with y

**Optimization opportunities**
- can create useless code (which can then be removed)
- can create improved code
- constant folding
- if done before other optimizations, can improve results

To do copy propagation, we must make sure two properties hold:

**Property 1)** No other definition of x reaches U
- How? Do a reaching-definitions analysis

  one way: data flow analysis
  another way: create CFG (control flow graph)
  - do "backwards" search starting at U
  - stop exploring a branch of the search when find a def of x (but continue overall search)
Example

\[
\begin{align*}
x &= 3; \\
y &= 5; \\
p &= x; \\
\text{if } (w \times x > 9) \{
    &\quad x = 4; \\
    &\quad z = x + w \times y;
\} \\
\text{else } \{
    &\quad z = 2 \times y + x;
\} \\
q &= 5 \times p; \\
s &= z + x; \\
t &= s + y;
\end{align*}
\]

After one pass

\[
\begin{align*}
p &= 3 \\
q &= 5 \times p \\
&\quad (p(1) \text{ reads } p(1), \text{ can copy propagate}) \\
q &= 5 \times 3
\end{align*}
\]
Copy Propagation (cont.)

**Property 2)** $y$ does not change between D and U
- If $y$ is a constant, then this is trivially true.

- If on any path through CFG from D to U there is a def of $y$, then $y$ might change.
- If $y$ and $z$ alias (refer to same memory location) and there is a def of $z$ between D & U, then $y$ might change.

\[ x = y \]
\[ // code to make $y$ & $z$ aliases \]
\[ z = 5; \]
\[ w = x + y; \]
\[ // can't change $x + y$ \]

**In C/C++**
\[ x = y; \]
\[ int *z = &y; \]
\[ *z = 5; \]
\[ w = x + y; \]

$x$ and $y$ are same loc in memory.

Optimization Wrap-up

- **From end**
  - Scanner
  - IR Optimizer
  - Static Semantics Analyzer

- **Back end**
  - Code generator
  - IR Optimizer

- Copy propagation
- Peephole optimizations
Where have we been?

CS 536: Introduction to Programming Languages and Compilers

What does a programming language consist of?
- tokens
- grammar
- static semantic analysis

What else? What choices are made?
- scoping rules
  - how do we match variable decks to variable uses?
    - what is allowed? Nested functions, nested var decks
- types
  - what types are there?
  - how do the types relate to each other?
- parameter passing
  - what ways are there to get info to a called procedure?
  - what impacts are there on the calling procedure?
- when do we check for things?
  - at compile time → statically
  - at run time → dynamically
  - or both?
Where have we been?
CS 536: Introduction to Programming Languages and Compilers

How do we translate a PL into something a computer can run? i.e., compilers

- recognizing tokens
  - regexps & FSMS
  - tools for translating

- recognizing languages
  - context-free grammars, parse
  - what can be parsed
  - how? top-down vs bottom-up parsing

- enforcing scoping and typing rules
- developing data structures that assist our translation/representation/translation
  - AST, parse tree, symbol tables
- how do we organize and manage memory
  - variables - where stored, how accessed
    - local vs global vs non-local
  - using registers & stack
- handling control flow within a program
  - interprocedural - how function calls & returns are implemented
  - intraprocedural - how are loops & selection statements implemented

How can we make our translation better?

- intermediate representations
- IR optimizations
  - copy propagation, L1CM & other loop optimizations
- MC optimizations
  - peephole optimizations
Course wrap-up

Covered a broad range of topics
- some formal concepts
- some practical concepts

What we skipped
- object-oriented language features
- dynamically-allocated memory management
- linking and loading
- interpreters
- register allocation
- dataflow analysis
- performance analysis
- proofs
Final Exam, Sunday, May 8, 12:25 pm
1800 Engineering Hall

Bring your UW Student ID

Reference material provided along with exam:
- copy of the minim grammar
- compiler class reference with selected class, methods, fields

Topic overview

Basic ideas of scanning & parsing
Symbol-table management / name analysis
- static scoping
- dynamic scoping

Type checking
Runtime storage management
- general storage layout
- activation records
- access to variables at runtime (parameters, locals, globals, non-locals)

Parameter-passing modes

Code generation
- numeric and control-flow approaches

Optimization
- goals
- optimization techniques (e.g., peephole optimization, copy propagation)

Extending
- grammar
- AST
- name analysis,
- type checking
- code generation
to handle new constructs