Compiler Construction

Optimizations

Albert-Ludwigs-Universität Freiburg

Peter Thiemann

University of Freiburg

8. Juli 2025

Outline

Optimization

- Objective: Transform the code to improve its run time, memory use, energy efficiency, etc.
- The transformation must preserve the semantics!
- Each optimization has two aspects
  1. a condition under which the optimization is applicable
  2. the actual program transformation
- An optimization can happen at any level
- Two examples of optimization
  - peephole optimization
  - common subexpression elimination

Outline

Constant Folding

Folding expressions

If $c = c_1 \odot c_2$ for a binary operation $\odot$, then

$$l: \quad x \leftarrow c_1 \odot c_2 \;\} \longrightarrow \{\; l: \quad x \leftarrow c$$

Constant Folding

Folding expressions

If $c = c_1 \odot c_2$ for a binary operation $\odot$, then

$$l: \quad x \leftarrow c_1 \odot c_2 \;\} \longrightarrow \{\; l: \quad x \leftarrow c$$

Folding conditionals

Let $c_1 ? c_2$ be a comparison.

$$l: \quad \text{if } c_1 ? c_2 \text{ then } l_1 \text{ else } l_2 \;\} \longrightarrow \{\; l: \quad \text{goto } l_1 \qquad \text{if } c_1 ? c_2 \text{ is true}$$

$$l: \quad \text{if } c_1 ? c_2 \text{ then } l_1 \text{ else } l_2 \;\} \longrightarrow \{\; l: \quad \text{goto } l_2 \qquad \text{if } c_1 ? c_2 \text{ is false}$$

Constant Folding (2)

Folding across multiple instructions

Suppose $\oplus$ is associative and $c = c_1 \oplus c_2$

$$\left.\begin{array}{ll} l_1: & y \leftarrow x \oplus c_1 \\ l_2: & z \leftarrow y \oplus c_2 \end{array}\right\} \longrightarrow \left\{\begin{array}{ll} l_1: & y \leftarrow x \oplus c_1 \\ l_2: & z \leftarrow x \oplus c \end{array}\right.$$

- sometimes y becomes dead and l_1 can be eliminated

Constant Folding (2)

Folding across multiple instructions

Suppose $\oplus$ is associative and $c = c_1 \oplus c_2$

$$\left.\begin{array}{ll} l_1: & y \leftarrow x \oplus c_1 \\ l_2: & z \leftarrow y \oplus c_2 \end{array}\right\} \longrightarrow \left\{\begin{array}{ll} l_1: & y \leftarrow x \oplus c_1 \\ l_2: & z \leftarrow x \oplus c \end{array}\right.$$

- sometimes y becomes dead and l_1 can be eliminated

Folding summary

- Very simple
- Classical peephole optimization: can be performed locally

Strength Reduction

- Replace an expensive instruction by a cheaper one.
- Usually: exploit arithmetic laws

$$
\begin{aligned}
x + 0 &= x \\
x - 0 &= x \\
x * 0 &= 0 \\
x * 1 &= x \\
x * 2^n &= x{<<}n
\end{aligned}
$$

- (more interesting in connection with loops)

Useless instructions

These instruction sequences look unnatural, but they do arise after register allocation.

$$l: \quad x \leftarrow x \; \Big\} \longrightarrow \Big\{ \; l: \quad \text{nop}$$

$$\left.\begin{array}{ll} l_1: & x \leftarrow y \\ l_2: & y \leftarrow x \end{array}\right\} \longrightarrow \left\{\begin{array}{ll} l_1: & x \leftarrow y \\ l_2: & \text{nop} \end{array}\right.$$

Outline

Constant Propagation

Mission

- Explore the consequences of a constant assignment $x \leftarrow c$.
- Thus enable constant folding.

Transformation rule

Let a stand for an arbitrary argument. If it is known that $x = c$ at label l, then

$$l: \quad y \leftarrow x \odot a \; \} \longrightarrow \{ \; l: \quad y \leftarrow c \odot a$$

$$l: \quad y \leftarrow a \odot x \; \} \longrightarrow \{ \; l: \quad y \leftarrow a \odot c$$

Constant Propagation (2)

Applicability

- dataflow analysis (working on CFG)
- recall structure: program point $\to$ variable $\to$ domain
- domain for liveness: bool (ordered by false $<$ true)
- domain for CP: $V_\bot^\top$ where V is the set of constants

Constant Propagation (2)

Applicability

- dataflow analysis (working on CFG)
- recall structure: program point $\to$ variable $\to$ domain
- domain for liveness: bool (ordered by false $<$ true)
- domain for CP: $V_\bot^\top$ where V is the set of constants

Domain construction: CP lattice

Let $\uplus$ denote disjoint union.

$$V_\bot^\top \quad := \quad V \uplus \{\bot\} \uplus \{\top\}$$

Define a partial order on $V_\bot^\top$ by

- for all $\hat{v}$: $\bot \le \hat{v}$ and $\hat{v} \le \top$
- for all $v, w \in V$: $v \le w$ iff $v = w$

Constant Propagation (3)

Lattice

- $V_\bot^\top$ is a *complete lattice* because every subset of elements has a least upper bound $\sqcup$ and a greatest lower bound $\sqcap$.
- (Knaster Tarski Theorem)
  Every monotone function on $V_\bot^\top$ has a fixed point.

Constant Propagation (3)

Lattice

- $V_\perp^\top$ is a *complete lattice* because every subset of elements has a least upper bound $\sqcup$ and a greatest lower bound $\sqcap$.
- (Knaster Tarski Theorem)
  Every monotone function on $V_\perp^\top$ has a fixed point.

Structure of the Analysis

- for each label, we have a preCP and a postCP : var $\to V_\perp^\top$.
- Initially, every variable is mapped to $\perp$ everywhere (unassigned).
- For each instruction l, we define a monotone *transfer function* that maps preCP(l) to postCP(l).
- Moreover, preCP(l) $= \bigsqcup_{p \in \mathrm{pred}(l)}$ postCP(p)

$\Rightarrow$ a *forward analysis*!

Constant Propagation (4)

Abstract evaluation

$\text{eval} : (\text{var} \to V_\bot^\top) \times \text{expression} \to V_\bot^\top$

$$
\begin{aligned}
\text{eval}(\rho, x) &= \rho(x) \\
\text{eval}(\rho, e_1 \oplus e_2) &= \text{eval}(\rho, e_1) \mathbin{\hat{\oplus}} \text{eval}(\rho, e_2)
\end{aligned}
$$

- If one argument of $\hat{\oplus}$ is $\bot$, then the result is $\bot$.
- Otherwise, if both arguments $v, w \in V$, $v \mathbin{\hat{\oplus}} w = v \oplus w$.
- Otherwise, if one argument is $\top$, then the result is $\top$.
- ($\oplus$ can be any binary operator including conditional)
- (unary operators are analogous)

Constant Propagation (5)

Transfer functions

Let $\rho = \text{preCP}(l)$ and $\rho' = \text{postCP}(l)$.

- $l : x \leftarrow e$, then $\rho' = \rho[x := \text{eval}(\rho, e)]$
- $l :$ **if** $x = e$ **then** l_1 **else** l_2, then let $\hat{e} = \text{eval}(\rho, e)$ and
  - $\rho'_1 = \rho[x := \hat{e} \sqcap \rho(x)]$ and
  - $\rho'_2 = \rho$ if $\hat{e} \stackrel{\wedge}{=} \rho(x) \sqsupseteq$ **false**
  - $\rho'_2 = \bot$ otherwise
- $l :$ **if** e **then** l_1 **else** l_2, then let $\hat{e} = \text{eval}(\rho, e)$
  - $\rho'_1 = \rho$ if $\hat{e} \sqsupseteq$ **true**; otherwise $\bot$
  - $\rho'_2 = \rho$ if $\hat{e} \sqsupseteq$ **false**; otherwise $\bot$

Constant Propagation (6)

```
z = 3
x = 1
while (x > 0) {
  if (x = 1) then
    y = 7
  else
    y = z + 4
  x = 3
  print y
}
```

Outline

Motivation

Avoid recomputation of the same expression

Transformation

$$\left.\begin{array}{ll} l_1: & y \leftarrow a_1 \oplus a_2 \\ & \dots \\ l_2: & z \leftarrow a_1 \oplus a_2 \end{array}\right\} \longrightarrow \left\{\begin{array}{ll} l_1: & y \leftarrow a_1 \oplus a_2 \\ & \dots \\ l_2: & z \leftarrow y \end{array}\right.$$

Conditions

- y should not be updated on any path from l_1 to l_2
- No variable occurring in $a_1 \oplus a_2$ should be changed on any path from l_1 to l_2
- Implemented with domain *available expressions* (AE)
- Enabled by $(y, a_1 \oplus a_2) \in AE(l_2)$

CSE (1)

Domain construction: AE lattice

$\mathbf{AE} = \{(y, e) \mid y \in \text{var}, e \in \text{expression}\}$

- powerset lattice (a complete lattice)
- finite for every program instance because each program contains finitely many variables and finitely many expressions

$\Rightarrow$ effective computation of the least fixed point

CSE(2)

Transfer Functions

Let $\alpha = \text{preAE}(l)$ and $\alpha' = \text{postAE}(l)$.

- $l : x \leftarrow e$, then
  $\alpha' = (\alpha \setminus \{(y, e') \mid y = x \vee x \in e'\}) \cup \{(x, e)\}$
  - remove prior assignments to x
  - remove expressions that (may have) changed due to assignment to x

CSE(2)

Transfer Functions

Let $\alpha = \text{preAE}(l)$ and $\alpha' = \text{postAE}(l)$.

- $l : x \leftarrow e$, then
  $\alpha' = (\alpha \setminus \{(y, e') \mid y = x \vee x \in e'\}) \cup \{(x, e)\}$
  - remove prior assignments to x
  - remove expressions that (may have) changed due to assignment to x

Style of analysis

- Forward analysis
- At joins of the control flow we only keep expressions available in **all** predecessors

$\Rightarrow$ $\text{preAE}(l) = \bigcap_{p \in \text{pred}(l)} \text{postAE}(p)$

Copy Propagation

Special case

If $(x, y) \in \text{preAE}(l)$, then we could replace uses of x by uses of y in instruction l.

- Advantage: might be able to eliminate x and thus the assignment(s) $x \leftarrow y$
- Disadvantage: the life range of y gets extended $\Rightarrow$ increased register pressure