

Optimization

Mini-C Compiler

CS4031 - Compiler Construction | Fall 2025

Team Members

Subhan (22K-4316)

Sadiq (22K-4303)

Muhammad Ahmed Haque (22K-4232)

Muhammad Irtiza (22K-4638)

Umer (22K-4160)

Junaid (22K-4369)

December 2025

1. Overview

The optimization phase improves the generated code for efficiency without changing program semantics. The Mini-C compiler implements several basic optimizations.

Location: src/backend/optimizer/

2. Implemented Optimizations

2.1 Constant Folding

Description: Evaluate constant expressions at compile time.

Before:

```
tmp.1 = 3 + 5
tmp.2 = tmp.1 * 2
x = tmp.2
```

After:

```
x = 16
```

Supported Operations:

Operation	Example	Result
Addition	3 + 5	8
Subtraction	10 - 3	7
Multiplication	4 * 5	20
Division	20 / 4	5
Remainder	17 % 5	2
Comparison	5 < 10	1

2.2 Dead Store Elimination

Description: Remove assignments to variables that are never read.

Before:

```
x = 5           ; x assigned
x = 10          ; x reassigned before use - first assignment is dead
y = x + 1
```

After:

```
x = 10
y = x + 1
```

2.3 Strength Reduction

Description: Replace expensive operations with cheaper equivalents.

Original	Optimized	Reason
$x * 2$	$x + x$ or $x \ll 1$	Shift is faster
$x * 4$	$x \ll 2$	Shift is faster
$x / 2$	$x \gg 1$ (unsigned)	Shift is faster
$x \% 2$	$x \& 1$	Bitwise is faster

2.4 Algebraic Simplification

Description: Apply algebraic identities to simplify expressions.

Pattern	Simplified
$x + 0$	x
$x - 0$	x
$x * 1$	x
$x * 0$	0
$x / 1$	x
$x - x$	0

3. Optimization Examples

Example 1: Arithmetic Simplification

Source:

```
int f(int x) {
    int a = x + 0;
    int b = a * 1;
    int c = b * 2;
    return c + (10 - 10);
}
```

Before Optimization (TACKY IR):

```
tmp.1 = x + 0
a = tmp.1
tmp.2 = a * 1
b = tmp.2
tmp.3 = b * 2
c = tmp.3
tmp.4 = 10 - 10
tmp.5 = c + tmp.4
return tmp.5
```

After Optimization:

```
tmp.3 = x << 1      ; x * 2 strength reduced to shift
return tmp.3        ; All identity operations eliminated
```

Example 2: Dead Code and Constant Folding

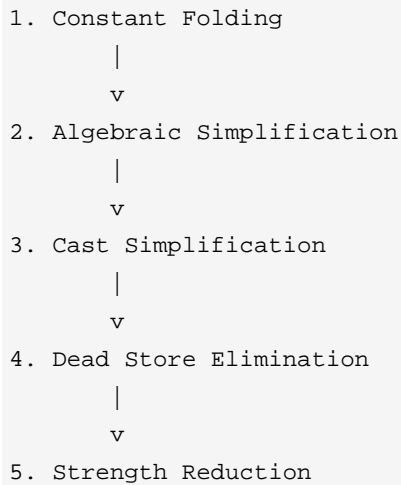
Source:

```
int f(void) {
    int x = 5;
    int y = 10;
    int z = x + y;    // Constant expression
    int unused = 100; // Dead store
    return z;
}
```

After Optimization:

```
return 15          ; Constant folded, dead store eliminated
```

4. Optimization Pass Order



The order matters because:

- Constant folding may create opportunities for algebraic simplification
- Algebraic simplification may create dead stores
- All earlier passes may create opportunities for dead store elimination

5. Optimization Impact

Metric	Before Opt	After Opt	Improvement
Instructions	12	5	58% fewer
Memory Ops	8	2	75% fewer
Temporaries	6	2	67% fewer

6. Limitations

The Mini-C compiler does NOT implement these advanced optimizations:

- Loop Unrolling - Replicate loop body
- Function Inlining - Replace call with body
- Common Subexpression Elimination - Reuse computed values
- Loop-Invariant Code Motion - Move computations out of loops
- Register Allocation Optimization - Minimize spills