Compiler Construction: Practical Introduction

Lecture 8
Program Optimization
Bootstrapping Compilers

Eugene Zouev

Spring Semester 2023
Innopolis University

Program Optimization

Some general points

- Optimization can be performed on each stage of the program lifecycle: not only while compilation but while design, development and maintenance.
- Do we **really need** optimization?
  The best way to optimize a program – is **to design it correctly** (then perhaps we do not need to optimize it ☺)
- "Optimization-in-the-small" vs "optimization-in-the-large"

Program Optimization

- Finding places in programs which could be optimized (by some criteria) is very much empirical job; in the best case, there is just a set of techniques taken from experience.
- At the same time, there is a number of formal and/or constructive approaches for some kind of optimizations.

Today, we will be discussing what to optimize, but not how to do this…

Program Optimization

- While source code processing (lexical & syntax analysis)

  Big spectrum of optimization techniques.

- While semantic analysis (AST processing).

  Sequential AST traversing.
  Optimizations depend on the language semantics heavily.

- While target code generation (machine-dependent optimizations)

  Depend on the target architecture & on the instruction set.

- While linking: ***global*** code optimizations.

  Example: – C++ code bloat removing.

Common Subexpression Elimination (1)

```
long a = x*(1-sin(y));
long b = x + y/z;
long c = y/z + 1 - sin(y);
```

```
long tmp1 = 1-sin(y);
long tmp2 = z/y;

long a = x*tmp1;
long b = x + tmp2;
long c = tmp2 + tmp1;
```

The place:

While AST analysis.

Limitations:

1. Factorized functions cannot issue side effects.
2. Operands of factorized expressions cannot modify their values.

Common Subexpression Elimination (2)

```
static int x = 0;
long F(long y)
{
    x++;
    return <expression>;
}
```

Side effect

```
long a = x*(1-F(y));
long b = x + y/z;
...
z = <expression>;
...
long c = z/y + 1 - F(y);
```

Modifying value

Common Subexpression Elimination (3)

An expression may look different but still calculate the same value as some other expression => it can also get optimized.

```
long a = b*c - d;

long e = b;

b = b + 1 - b*c;

long f = b*c + c*e;
```

b*c: the second calculation of the same value.

The second calculation of b*c results in a different value.

The value of c*e is the same as b*c.

Operation Strength Reduction (1)

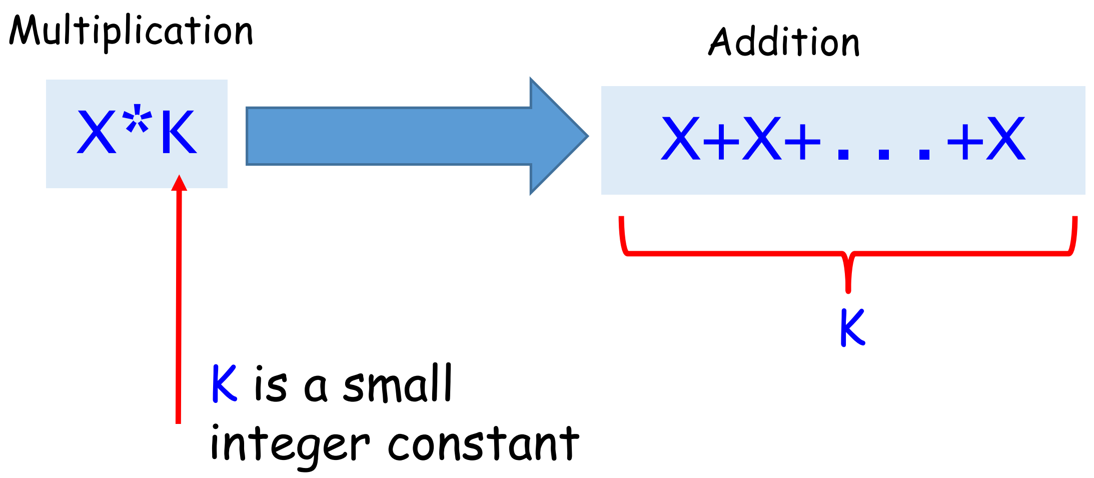
Actions: comparative performance

Multiplication/division on a power of two
Addition/subtraction
Multiplication
Division
Calculation of an integer power
Calculation of an arbitrary power

⇒ Replacing slower operations for faster ones (where possible)

For some target architectures it's **mandatory**: e.g., some RISCs just do not support multiplication!

Operation Strength Reduction (2)



In general case it's impossible...

- At least one operand must be an integer constant.
- The constant should be relatively small; otherwise rounding errors will accumulate.

Operation Strength Reduction (3)

Division

X/K

K - constant

Multiplication

X*(1/K)

```
double x = c/b;
double y = (e+f)/b + d;
double z = b;
b = b+1;
...
z = sin(x)/z + e/b;
```

```
double tmp = (double)1/b;
double x = c*tmp;
double y = (e+f)*tmp + d;
double z = b;
b = b+1;
...
z = sin(x)*tmp + e/b;
```

Dead Code Elimination

```
double a;
...
a = (x+y)*sin(z);
...
a = x/y;
```

```
double a;
...
a = x/y;
```

If the value of a ***does not change*** between two assignments, then the first assignment can be removed.

Limitation: the action being removed cannot make side effects.

Constant propagation

If the value of a variable is known
then the variable reference can be replaced
for the value itself.

```
long a = 2;
long b = 3;
...
long c = a*b;
...
long t = (b+c)*a+x;
```

```
long a = 2;
long b = 3;
...
long c = 6; // a*b
...
long t = 18 + x; // (b+c)*a
```

Conditional Constant propagation

If the value in a loop condition is known in advance then the loop could be simplified.

```
while ( <Expression> )
{
    <Statements>
}
```

```
Loop:
  if ( !<Expression> ) goto Exit;
  <Statements>
  goto Loop;
Exit:
  ;
```

```
while ( 1 )
{
    <Statements>
}
```

Loop:
~~if (!<Expression>) goto Exit;~~
<Statements>
goto Loop;
~~Exit:~~
~~;~~

Type conversion optimizations

Type conversion is a potentially costly operation; therefore it's a good candidate for optimizations.

```
double a, b;
long i, j;
...
double c = a + i + b - j;
```

```
... a + (double)i + b - (double)j ...
```

```
double a, b;
long i, j;
...
double c = (a+b) + (i-j);
```

```
... a+b + (double)(i-j) ...
```

Code Hoisting

Access to array elements and function calls are also good candidates for optimizations...

```
for i:integer range 1..100 loop
    x(i) = y(i)+1/y(i);
    z(i) = y(i)**2;
end loop;
```

Address of `y(i)` gets calculated **300 times**

```
for i:integer range 1..100 loop
    declare
        tmp : real := y(i);
    begin
        x(i) = tmp+1/tmp;
        z(i) = tmp**2;
    end;
end loop;
```

Address of `y(i)` gets calculated **100 times**

Loop Fusion (1)

Loops are main consumers of CPU time!

```
for i:integer range 1..100 loop
    x(i) = 0;
end loop;
```

```
for i:integer range 1..100 loop
    z(i) = y(i)**2;
end loop;
```

```
for i:integer range 1..100 loop
    x(i) = 0;
    z(i) = y(i)**2;
end loop;
```

Costs for loop organization are reduced

Loop Fusion (2)

(More general case)

```
for i:integer range 1..100 loop
    x(i) = 0;
end loop;
```

```
for i:integer range 1..200 loop
    z(i) = y(i)**2;
end loop;
```

Overall amount of iterations: **300**

```
for i:integer range 1..100 loop
    x(i) = 0;
    z(i) = y(i)**2;
end loop;
```

```
for i:integer range 101..200 loop
    z(i) = y(i)**2;
end loop;
```

Overall amount of iterations: **200**

Loop Unrolling (1)

```
for (int i=0; i<100; i++)
{
    x[i] = y[i]*z[i];
}
```

Loop step = **1**
Overall amount
of iterations: **100**

```
for (int i=0; i<100; i+=2)
{
    x[i] = y[i]*z[i];
    x[i+1] = y[i+1]*z[i+1];
}
```

Loop step = **2**
Overall amount
of iterations: **50**

Loop Unrolling (2)

```
-- Skip past blanks, loop is opened up for speed
while Source (Scan_Ptr) = ' ' loop
    if Source (Scan_Ptr + 1) /= ' ' then
        Scan_Ptr := Scan_Ptr + 1; exit;
    end if;
    if Source (Scan_Ptr + 2) /= ' ' then
        Scan_Ptr := Scan_Ptr + 2; exit;
    end if;
    if Source (Scan_Ptr + 3) /= ' ' then
        Scan_Ptr := Scan_Ptr + 3;  exit;
    end if;
    if Source (Scan_Ptr + 4) /= ' ' then
        Scan_Ptr := Scan_Ptr + 4; exit;
    end if;
    if Source (Scan_Ptr + 5) /= ' ' then
        Scan_Ptr := Scan_Ptr + 5; exit;
    end if;
    if Source (Scan_Ptr + 6) /= ' ' then
        Scan_Ptr := Scan_Ptr + 6; exit;
    end if;
    if Source (Scan_Ptr + 7) /= ' ' then
        Scan_Ptr := Scan_Ptr + 7; exit;
    end if;
    Scan_Ptr := Scan_Ptr + 8;
end loop;
```

"Manual" loop unrolling

A real example:
The scanner of the Ada GNAT compiler

Tail Recursion Elimination (1)

```
void f(int x)
{
    if ( x == 0 ) return;
    ...Some actions...
    f(x-1);
}
```

```
void f(int x)
{
    if ( x == 0 ) return;
    ... Some actions...
    if ( Some_condition )
        f(x-1);
    else
        f(x-2);
}
```

The idea:
If the recursive call is the very last operation in the function body then it can be replaced for the direct jump to the beginning of the body – perhaps with argument (re)initialization.

See more details in:
http://en.wikipedia.org/wiki/Tail_call

Tail Recursion Elimination (2)

```
void f(int x)
{
    if ( x == 0 ) return;
    ...Some actions...
    f(x-1);
}
...
f(3);
```

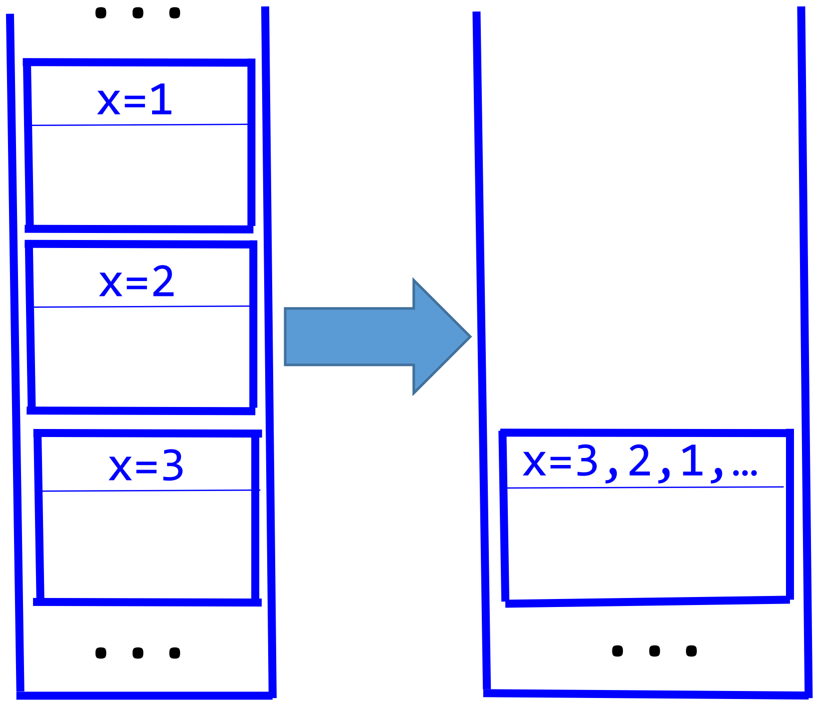
```
void f(int x)
{
  Repeat:
    if ( x == 0 ) return;
    ...Some actions...
    f(x-1);
    x = x-1;
    goto Repeat;
}
...
f(3);
```

Execution Stack

Stackframe for f's call with x = 1

Stackframe for f's call with x = 2

Stackframe for f's call with x = 3



The idea:
If the recursive call is the very last operation in the function body then it can be replaced for the direct jump to the beginning of the body – perhaps with argument (re)initialization.

Tail Recursion Elimination (3)

```
long factorial(long n)
{
   if (n == 0) return 1;
   else return n*factorial(n-1);
}
```

This is **not** tail recursion. **Why?**

```
int fac_times(int n, int acc)
{
    if (n == 0) return acc;
    else return fac_times(n-1,acc*n);
}

int factorial(int n)
{
    return fac_times(n,1);
}
```

Equivalent program **with** tail recursion.

Bootstrapping the Compiler

In many cases, a compiler for a higher-level language "High" is written in a <u>lower</u> language "Low".

For example, the first compiler for the Fortran language was initially written in an assembly language…

The Interstron C++ compiler was initially written in C

However:

- The GNAT **Ada** compiler is written in Ada.
- The **Eiffel** compiler is written in Eiffel.
- The **Scala** compiler is written in Scala.
- Most **C++** compilers are written in C++
- …

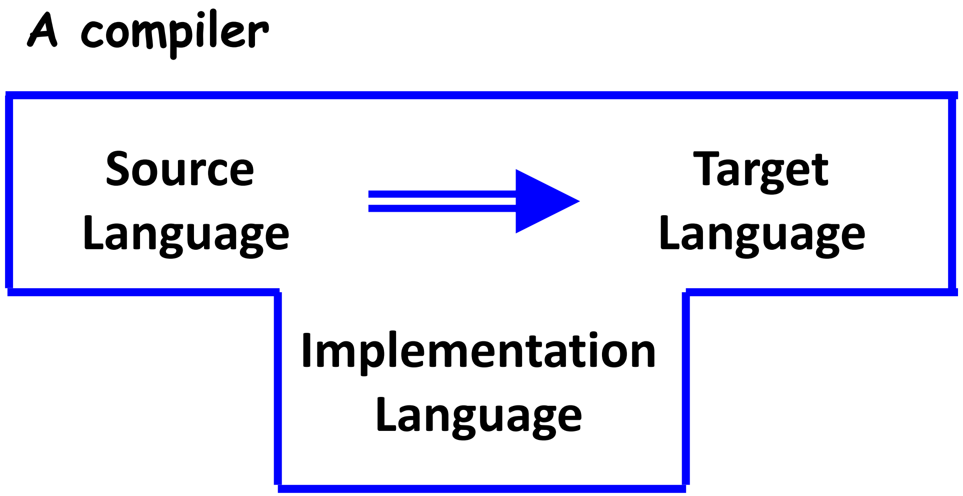
The Interstron C++ compiler was later rewritten in C++

Bootstrapping Technology

- The technology applies when the implementation & source languages are **the same**.
- **Advantages**:
  - More stable technology;
  - Supports graduate language & compiler improvement;
  - No dependency on any third-party tools;
  - The code of the compiler is **an excellent test** for both language and compiler itself.
- **Disadvantages**:
  - A bit awkward technology; requires non-trivial management & powerful management tools (e.g., ***ant***).

Bootstrapping Technology

- Reference: Terence Parr.
- Graphical notation ("T Notation"):



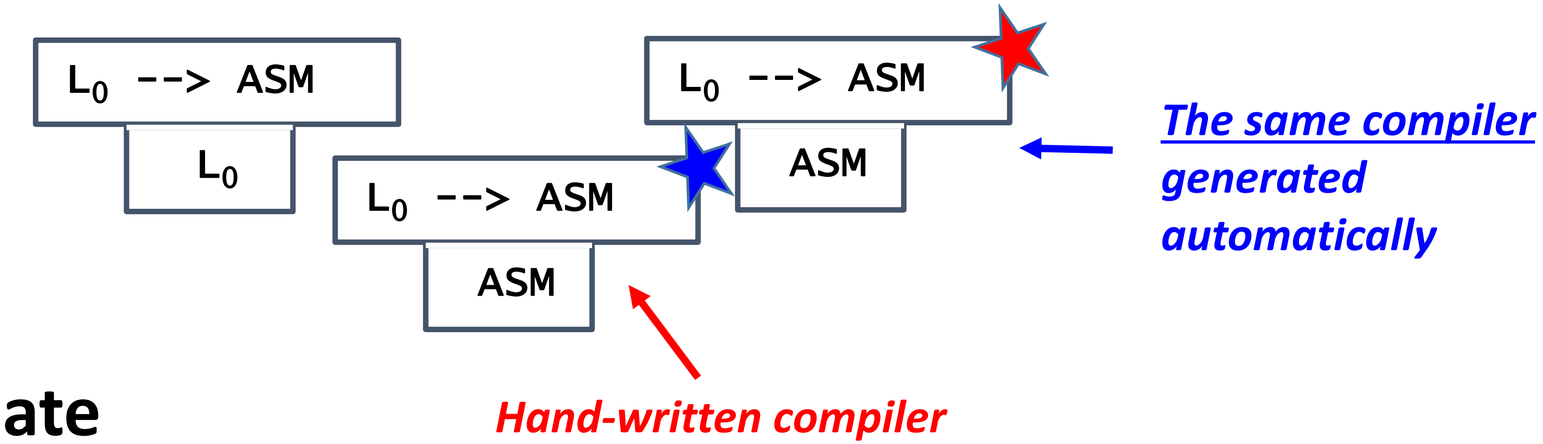
Bootstrapping the Compiler

0. Initial development step

"ASM" can be any assembly language:
- Intel x64
- LLVM
- .NET MSIL
- JVM bytecode
- etc.

- Define a very simple <u>subset</u> of the target L language: L_0
- Write the prototype compiler for L_0 <u>in the same L_0 language</u>
- <u>Manually</u> <u>rewrite</u> this prototype compiler in an assembly language

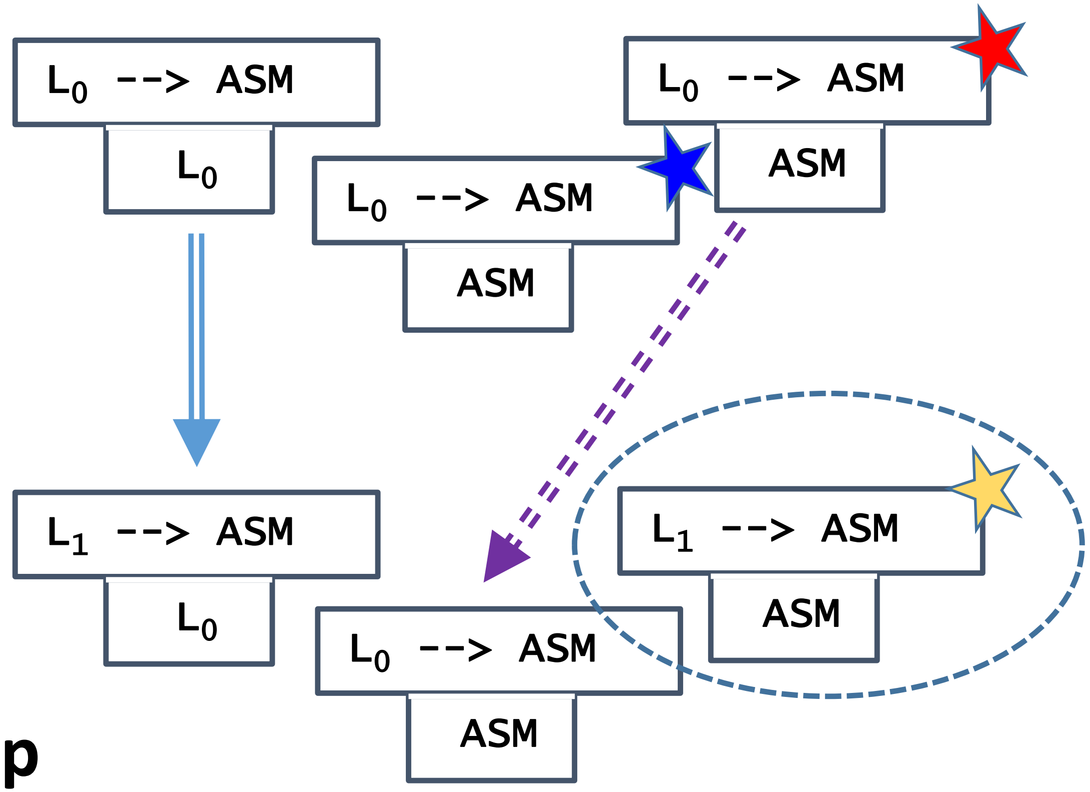
Bootstrapping the Compiler



0a. Intermediate development step

- **Apply the hand-written compiler to the initially written one**

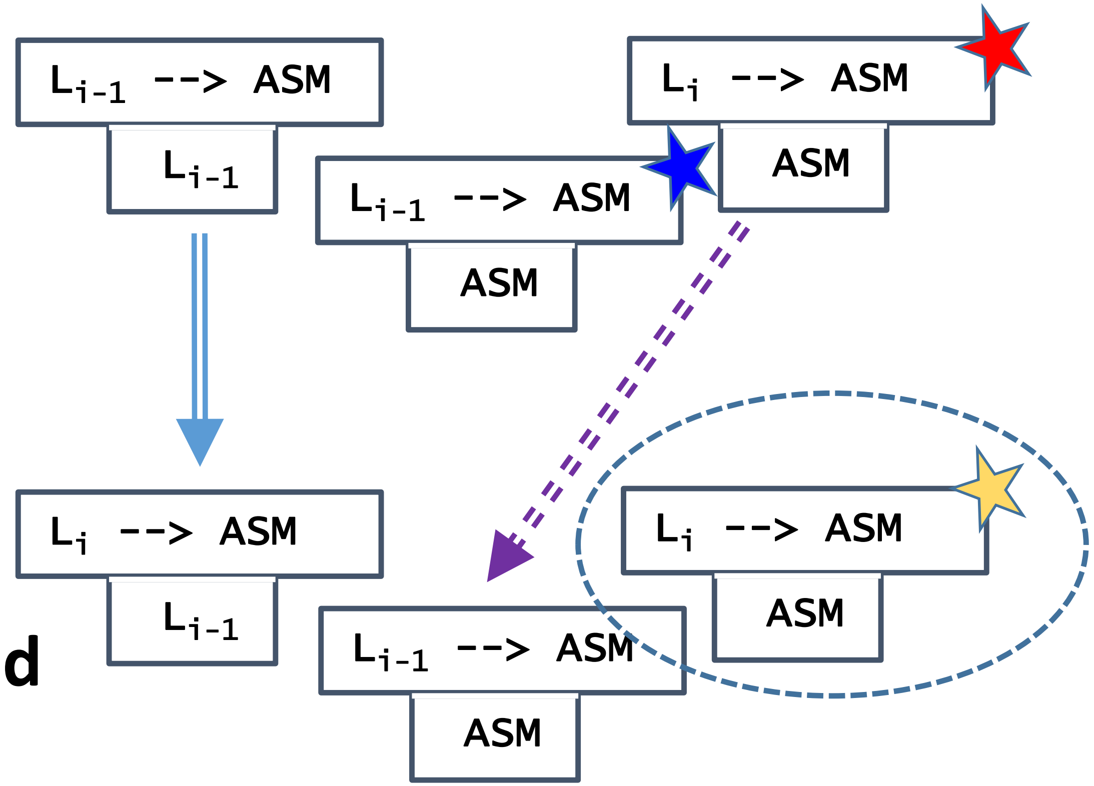
Bootstrapping the Compiler



1. The first development step

- Extend the initial L_0 compiler adding new features: get the L_1 compiler
- Apply the previous version of the compiler to the newer one
- The result: the compiler for the next version of the language.

Bootstrapping the Compiler



2. The second and the following development steps

- Extend the current L_{i-1} compiler adding new features: get the L_i compiler
- Apply the $i\text{-}1_{th}$ version of the compiler to the i_{th} one
- The result: the compiler for the next version of the language.