Optimising Compilers

Computer Science Tripos Part II

Timothy Jones
Lecture 1
Introduction
A non-optimising compiler

- character stream
  - lexing
  - token stream
  - parsing
  - parse tree
  - translation
  - intermediate code
  - code generation
  - target code
An optimising compiler

- Character stream
  - Lexing
  - Token stream
  - Parsing
    - Parse tree
      - Translation
        - Intermediate code
          - Code generation
            - Target code
              - Optimisation

- Decompilation
Optimisation
(really “amelioration”!)

Good humans write simple, maintainable, general code.

Compilers should then remove unused generality, and hence hopefully make the code:

• Smaller
• Faster
• Cheaper (e.g. lower power consumption)
Optimisation = Analysis + Transformation
Analysis + Transformation

• Transformation does something dangerous.
• Analysis determines whether it’s safe.
Analysis + Transformation

• An analysis shows that your program has some property...
• ...and the transformation is designed to be safe for all programs with that property...
• ...so it’s safe to do the transformation.
Analysis + Transformation

```c
int main(void)
{
    return 42;
}

int f(int x)
{
    return x * 2;
}
```
int main(void)
{
    return 42;
}

int f(int x)
{
    return x * 2;
}
Analysis + Transformation

```c
int main(void)
{
    return f(21);
}

int f(int x)
{
    return x * 2;
}
```
Analysis + Transformation

```c
int main(void)
{
    return f(21);
}
```
while (i <= k*2) {
    j = j * i;
    i = i + 1;
}

Analysis + Transformation
int t = k * 2;
while (i <= t) {
    j = j * i;
    i = i + 1;
}

✓
Analysis + Transformation

```c
while (i <= k*2) {
    k = k - i;
    i = i + 1;
}
```
int t = k * 2;
while (i <= t) {
    k = k - i;
    i = i + 1;
}

✗
Stack-oriented code

iload 0
iload 1
iadd
iload 2
iload 3
iadd
imul
ireturn
3-address code

MOV t32, arg1
MOV t33, arg2
ADD t34, t32, t33
MOV t35, arg3
MOV t36, arg4
ADD t37, t35, t36
MUL res1, t34, t37
EXIT
C into 3-address code

```c
int fact (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n-1);
    }
}
```
C into 3-address code

ENTRY fact
MOV t32, arg1
CMPEQ t32, #0, lab1
SUB arg1, t32, #1
CALL fact
MUL res1, t32, res1
EXIT

lab1: MOV res1, #1
EXIT
Flowgraphs

- A graph representation of a program
- Each node stores 3-address instruction(s)
- Each edge represents (potential) control flow:

\[\text{pred}(n) = \{n' \mid (n', n) \in \text{edges}(G)\}\]
\[\text{succ}(n) = \{n' \mid (n, n') \in \text{edges}(G)\}\]
Flowgraphs

ENTRY fact

MOV t32, arg1

CMPEQ t32, #0

MOV res1, #1

SUB arg1, t32, #1

CALL fact

MUL res1, t32, res1

EXIT

EXIT
Basic blocks

A maximal sequence of instructions $n_1, ..., n_k$ which have

• exactly one predecessor (except possibly for $n_1$)

• exactly one successor (except possibly for $n_k$)
Basic blocks

ENTRY fact

MOV t32, arg1

CMPEQ t32, #0

SUB arg1, t32, #1

CALL fact

MUL res1, t32, res1

EXIT

MOV res1, #1

EXIT
Basic blocks

ENTRY fact
MOV t32, arg1
CMPEQ t32, #0

MOV res1, #1
EXIT

SUB arg1, t32, #1
CALL fact
MUL res1, t32, res1
EXIT
Basic blocks

ENTRY fact

MOV t32, arg1
CMPEQ t32, #0

MOV res1, #1
CALL fact
MUL res1, t32, res1

SUB arg1, t32, #1
EXIT
Basic blocks

A basic block doesn’t contain any interesting control flow.
Basic blocks

Reduce time and space requirements for analysis algorithms by calculating and storing data flow information once per block (and recomputing within a block if required) instead of once per instruction.
Basic blocks

MOV t32, arg1
MOV t33, arg2
ADD t34, t32, t33
MOV t35, arg3
MOV t36, arg4
ADD t37, t35, t36
MUL res1, t34, t37
Types of analysis
(and hence optimisation)

Scope:

• Within basic blocks (“local” / “peephole”)
• Between basic blocks (“global” / “intra-procedural”)
  • e.g. live variable analysis, available expressions
• Whole program (“inter-procedural”)
  • e.g. unreachable-procedure elimination
Peephole optimisation

ADD t32, arg1, #1
MOV r0, r1
MOV r1, r0
MUL t33, r0, t32

ADD t32, arg1, #1
MOV r0, r1
MOV r1, r0
MUL t33, r0, t32

matches
replace
MOV x, y
MOV y, x
with
MOV x, y
Types of analysis
(and hence optimisation)

Type of information:

• Control flow
  • Discovering control structure (basic blocks, loops, calls between procedures)

• Data flow
  • Discovering data flow structure (variable uses, expression evaluation)
Finding basic blocks

1. Find all the instructions which are leaders:
   - the first instruction is a leader;
   - the target of any branch is a leader; and
   - any instruction immediately following a branch is a leader.

2. For each leader, its basic block consists of itself and all instructions up to the next leader.
ENTRY fact
MOV t32, arg1
CMPEQ t32, #0, lab1
SUB arg1, t32, #1
CALL fact
MUL res1, t32, res1
EXIT
lab1:
MOV res1, #1
EXIT
Summary

• Structure of an optimising compiler
• Why optimise?
• Optimisation = Analysis + Transformation
• 3-address code
• Flowgraphs
• Basic blocks
• Types of analysis
• Locating basic blocks