COMPILER CONSTRUCTION (Code Generation)

Prof. K R Chowdhary
Email: kr.chowdhary@jietjodhpur.ac.in

Campus Director, JIET, Jodhpur

Tuesday 16th October, 2018

Introduction

It is final phase in compiler, takes as input the intermediate representation along with relevant symbol table information, and produces as output a semantically equivalent target program (see Fig. 1).

Figure 1: Position of code generator.

The target program (machine code) must preserve the semantics of the source program and be of high quality; it must make effective use of the available resources.
A code generator has three primary tasks: instruction selection, register allocation and assignment, and instruction ordering.

Basic Blocks and Flow Graph

It is helpful for discussing code generation. We can do a better job of register allocation if we know how values are defined and used. We can select the instruction in a better way, etc.

Partition the intermediate code into basic blocks, with following properties

1. The flow of control can only enter the basic block through the first instruction in the block.
2. Control will leave the block at the last instruction in the block.

The basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow the other blocks.

Basic Blocks Algorithm

Algorithm-1: Partitioning three-address instructions set into basic blocks.
INPUT: A sequence of three-address instructions.
OUTPUT: A list of the basic blocks, such that each instruction is assigned to exactly one basic block.
METHOD: First, we determine those instructions in the intermediate code that are *leader instructions*, that is, the first instructions in some basic block. The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Example

Intermediate code to set a 10×10 matrix to an identity matrix.

```
for i from 1 to 10 do
  for j from 1 to 10 do
     a[i,j] = 0.0;
for i from 1 to 10 do
   a[i,i] = 1.0;
```

Intermediate code to set a 10×10 matrix to identity matrix

1) $i = 1$
2) $j = 1$
3) $t1 = 10 * i$
4) $t2 = t1 + j$
5) $t3 = 4 * t2$
6) $t4 = t3 - 44$
7) $a[t4] = 0.0$
8) $j = j + 1$
9) $if\ \ j <= 10\ goto\ (3)$
10) $i = i + 1$
11) $if\ \ i < 10\ goto\ (2)$
12) $i = 1$
13) $t5 = i - 1$
14) $t6 = 44 * t5$
15) $a[t6] = 1.0$
16) $i = i + 1$
17) $if\ \ i <= 10\ goto\ (13)$

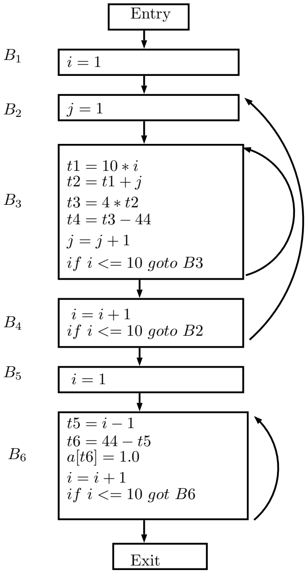
Analysis of 10×10 matrix

1) $i = 1$
2) $j = 1$
3) $t1 = 10 * i$
4) $t2 = t1 + j$
5) $t3 = 4 * t2$
6) $t4 = t3 - 44$
7) $a[t4] = 0.0$
8) $j = j + 1$
9) $if\ \ j <= 10\ goto\ (3)$
10) $i = i + 1$
11) $if\ \ i < 10\ goto\ (2)$
12) $i = 1$
13) $t5 = i - 1$
14) $t6 = 44 * t5$
15) $a[t6] = 1.0$
16) $i = i + 1$
17) $if\ \ i <= 10\ goto\ (13)$

Flow Graph

The nodes of the flow graph are the basic blocks.
There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B.
We say that B is a predecessor of C.
Often we add two nodes, called the entry and exit, that do not correspond to executable intermediate instructions.
There is an edge from the entry to the first executable node of the flow graph, i.e., to the basic block that comes from the first instruction.

flow-graph from intermediate code



Optimization of Basic Block code by DAG

We can perform several code-improving transformations on the code represented by the block.

1. Elimination of local common subexpressions
2. We can eliminate dead code
3. We can reorder statements that do not depend on one another;
4. We can apply algebraic laws to reorder operands of three-address instructions, and sometimes it simplify the computation.

Finding Local Common Subexpressions

Common subexpressions can be detected by noticing, as a new node M is about to be added, whether there is an existing node N with the same children, in the same order, and with the same operator. If so, N computes the same value as M and may be used in its place.

$$\begin{aligned} a &= b + c \\ b &= a - d \\ c &= b + c \\ d &= a - d \end{aligned} \tag{1}$$

Finding Local Common Subexpressions

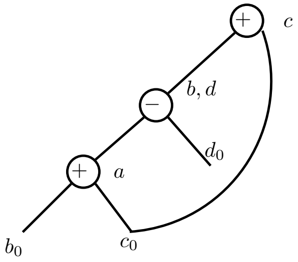


Figure 2: DAG for basic block.

$$\begin{aligned} a &= b + c \\ d &= a - d \\ c &= d + c \end{aligned} \tag{2}$$

Other Example & Dead Code Elimination.

$$
\begin{aligned}
a &= b + c \\
b &= b - d \\
c &= c + d \\
e &= b + c
\end{aligned}
\tag{3}
$$

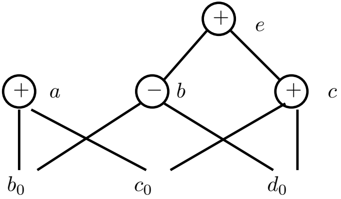


Figure 3: DAG for basic block in equation 3.

Other Example & Dead Code Elimination.

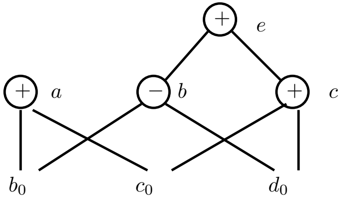


Figure 4: DAG for basic block in equation 3.

We delete from a DAG any root (node with no ancestors) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

If, in Fig. 4, *a* and *b* are live but *c* and *e* are not, we can immediately remove the root labeled *e*. Then, the node labeled *c* becomes a root and can be removed. The roots labeled *a* and *b* remain, since they each have live variables attached.