COSE312: Compilers

Lecture 17 — Intermediate Representation (2)

Hakjoo Oh
2017 Spring

Common Intermediate Representations

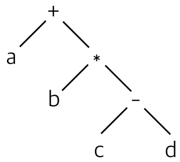
- Three-address code
- Static single assignment form
- Control-flow graph

Three-Address Code

- Instructions with at most one operator on the right side.
- Temporary variables are needed in translation, e.g., $x + y * z$:

$$\begin{array}{rcl} t_1 & = & y * z \\ t_2 & = & x + t_1 \end{array}$$

- A linearized representation of a syntax tree, where temporary variables correspond to the internal nodes of the tree: e.g.,



Static Single Assignment Form

- An intermediate representation suitable for many code optimizations.
- A program is in SSA iff
  1. each definition has a distinct name, and
  2. each use refers to a single definition.
- Example) Convert the following code into SSA form:

```
p = a + b
q = p - c
p = q * c
p = e - p
q = p + q
```

Static Single Assignment Form

The SSA form of the following:

```
if (flag) x = -1; else x = 1;
y = x * a;
```

needs a ϕ-function:

$$\texttt{if (flag) x}_1\texttt{ = -1; else x}_2\texttt{ = 1;}$$
$$\texttt{x}_3\texttt{ = }\phi\texttt{(x}_1\texttt{, x}_2\texttt{);}$$
$$\texttt{y = x}_3\texttt{ * a;}$$

Here, $\phi(x_1, x_2)$ has the value x_1 if the control flow passes through the true branch and the value x_2 otherwise.

Static Single Assignment Form

Exercise) Convert the following code into an SSA form:

```
i = 1
j = 1
k = 0
while (1) {
  if (k < 100) {
    if (j < 20)
      j = i
      k = k + 1
    else
      j = k
      k = k + 2
  }
  else return j
}
```

```
i1 = 1
j1 = 1
k1 = 0
while (1) {
  j2 = phi(j4, j1)
  k2 = phi(k4, k1)
  if (k2 < 100) {
    if (j2 < 20)
      j3 = i1
      k3 = k2 + 1
    else
      j5 = k2
      k5 = k2 + 2
  }
  else return j2
  j4 = phi(j3, j5)
  k4 = phi(k3, k5)
}
```

How to Convert a Program into SSA?

Efficiently Computing Static Single Assignment Form and the Control Dependence Graph

RON CYTRON, JEANNE FERRANTE, BARRY K. ROSEN, and
MARK N. WEGMAN
IBM Research Division
and
F. KENNETH ZADECK
Brown University

In optimizing compilers, data structure choices directly influence the power and efficiency of practical program optimization. A poor choice of data structure can inhibit optimization or slow compilation to the point that advanced optimization features become undesirable. Recently, static single assignment form and the control dependence graph have been proposed to represent data flow and control flow properties of programs. Each of these previously unrelated techniques lends efficiency and power to a useful class of program optimizations. Although both of these structures are attractive, the difficulty of their construction and their potential size have discouraged their use. We present new algorithms that efficiently compute these data structures for arbitrary control flow graphs. The algorithms use *dominance frontiers*, a new concept that may have other applications. We also give analytical and experimental evidence that all of these data structures are usually linear in the size of the original program. This paper thus presents strong evidence that these structures can be of practical use in optimization.

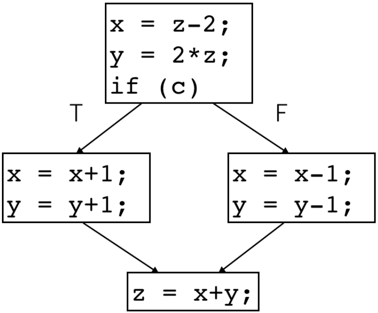
Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs—*control structures*; *data types and structures*; *procedures, functions and subroutines*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; I.1.2 [**Algebraic Manipulation**]: Algorithms—*analysis of algorithms*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*

Cytron et al.
Efficiently Computing Static Single Assignment Form and the Control Dependence Graph.
ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 13 Issue 4, Pages 451-490

Control-Flow Graph

- Control-Flow Graph (CFG): graph representation of the program
  - A commonly used form for static analysis and optimization
  - Nodes are basic blocks
  - Edges represent control flows

```
x = z-2;
y = 2*z;
if (c) {
  x = x+1;
  y = y+1;
} else {
  x = x-1;
  y = y-1;
}
z = x+y;
```



Basic Blocks

- Maximal sequences of consecutive, branch-free instructions.

```
   x = 1
   y = 1
   z = x + y
L: t1 = z + 1
   t1 = t1 + 1
   z = t1
   goto L
```

- Properties:
  - Instructions in a basic block are always executed together.
  - No jumps to the middle of a basic block.
  - No jumps out of a basic block, except for the last instruction.

Partitioning Instructions into Basic Blocks

Given a sequence of instructions:

- Determine *leaders*, the first instructions in some basic block.
  1. The first instruction 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.
- For each leader, its basic block consists of itself and all instruction up to but not including the next leader or the end of the program.

Example

```
    i = 1
L1: j = 1
L2: t1 = 10 * i
    t2 = t1 + j
    t3 = 8 * t2
    t4 = t3 - 88
    a[t4] = 0
    j = j + 1
    if j <= 10 goto L2
    i = i + 1
    if i <= 10 goto L1
    i = 1
L3: t5 = i - 1
    t6 = 88 * t5
    a[t6] = 1
    i = i + 1
    if i <= 10 goto L3
```

Control-Flow Graph

A graph representation of intermediate code:

- A directed graph $G = (N, \hookrightarrow)$, where each node $n \in N$ is a basic block and an edge $(n_1, n_2) \in (\hookrightarrow)$ indicates a possible control flow of the program.
- $n_1 \hookrightarrow n_2$ iff
  - there is a conditional or unconditional jump from the end of n_1 to the beginning of n_2, or
  - n_2 immediately follows n_1 in the original program, and n_1 does not end in an unconditional jump.
- Often, control-flow graphs have unique *entry* and *exit* nodes.

Example

```
    i = 1
L1: j = 1
L2: t1 = 10 * i
    t2 = t1 + j
    t3 = 8 * t2
    t4 = t3 - 88
    a[t4] = 0
    j = j + 1
    if j <= 10 goto L2
    i = i + 1
    if i <= 10 goto L1
    i = 1
L3: t5 = i - 1
    t6 = 88 * t5
    a[t6] = 1
    i = i + 1
    if i <= 10 goto L3
```

CFG Construction for High-level Languages

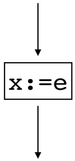
- High-level statements:

$$S \rightarrow x := e \mid S_1; S_2 \mid \textit{if } e \; S_1 \; S_2 \mid \textit{while } e \; S$$

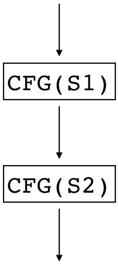
- $CFG(S)$: control-flow graph of S
- $CFG(S)$ is recursively defined

CFG Construction for High-level Languages

- $x := e$

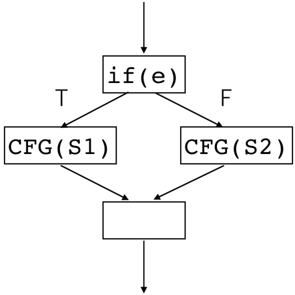


- $S_1; S_2$

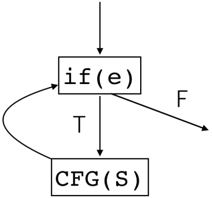


CFG Construction for High-level Languages

- $if\ e\ S_1\ S_2$



- $while\ e\ S$



Example

```
while (c) {
  x = y;
  y = 2;
  if(d) x = y;
  else y = x;
  z = 1;
}
z = x
```

Summary

Intermediate Representations:

- Three-address code
- Static single assignment form
- Control-flow graph