Applied Static Analysis

Three Address Code

Software Technology Group
Department of Computer Science
Technische Universität Darmstadt
Dr. Michael Eichberg

Intermediate Representations

Goal: Facilitate Static Analyses

How:
- Nested Control-flow and complex expressions are unraveled.
- Intermediate values are given explicit names.
- The data-flow is made (more) explicit.
- The instruction set is limited (more orthogonal).
- ...

Examples:
- 3-Address Code (TAC)
- Static Single Assignment (Form) (SSA)

Typically, an intermediate representation tries to limit the number of alternatives how a specific effect can be reached. E.g., instead of offering four different ways of *loading* a constant int value as in Java bytecode (`iconst_X` , `bipush` , `sipush` and `ldc(_w)`) only one instruction is offered.

Basically every compiler or static analysis framework offers one or more intermediate representations. Because there isn't a single representation that facilitates all kinds of optimizations/transformations/analyses equally well. Soot – a Java Bytecode based analysis framework – for example, offers the three representations called: `BAF` , `Jimple` and `Shimple` .

Three-address Code

Three-address code is a sequence of statements (linearized representation of a syntax tree) with the general form:

```
x = y op z
```

where x,y and z are (local variable) names, constants (in case of y and z) or compiler-generated temporaries.

The name was chosen, because *most* statements use three addresses: two for the operators and one to store the result.

Two fundamental implementation techniques can be distinguished:

- quadruples: in this case result is made explicit.
- triplets: in this representation temporary names are avoided by directly referring to the definition sites in expression.

In static analysis frameworks quadruples based representations are more common and the result variable is also used to bind def-use information.

General Types of Three-Address Statements

- Assignment statements: `x = y op z` or `x = op z`
- Copy statements `x = y`
- Unconditional jumps: `goto l`
- Conditional jumps: `if (x rel_op y) goto l` (else fall through), `switch`
- Method call and return: `invoke(m, params)`, `return x`
- Array access: `a[i]` or `a[i] = x`
- *IR specific types.*

In three-address code it is often customary that a jump is performed if the condition of an `if` statement evaluates to true.

Unconditional jump statements that are specific to Java Bytecode are `jsr l` and `ret`.

Converting Java Bytecode to Three-Address Code

(Syntax-directed Translation)

- Compute for each instruction the current stack layout by following the control flow; i.e., compute the types of values found on the stack before the instruction is evaluated.
  (This is required to correctly handle generic stack-manipulation instructions.)
- Assign each local variable to a variable where the name is based on the local variable index.
- Assign each variable on the operand stack to a corresponding local variable with an index based on the position on the stack.

The JVM specification guarantees that the operand stack always has the same layout independent of the taken path.

E.g., an `iinc(local variable index=1, increment=2)` instruction is transformed into the three address code: `r_1 = r_1 + 2`.

E.g., if the operand stack is empty and we push the constant 1, then the three address code would be: op_0 = 1; if we would then push another value 2 then the code would be: `op_1 = 2` and an addition of the two values would be: `op_0 = op_0 + op_1`.

A more detailed discussion can be found in the dragon book[1].

Converting Java Bytecode to three-address code

```
static int numberOfDigits(int i) {
    return ((int) Math.floor(Math.log10(i))) + 1;
}
```

PC	Code	Stack Layout	TAC
-	-	-	`r_0 = i // init parameters`
0	iload_0		`op_0 = r_0`
1	i2d	0: Int Value, →	`op_0 = (double) op_0`
2	invokestatic log10 (double):double	0: Double Value , →	`op_0 = log10(op_0)`
5	invokestatic floor(double):double	0: Double Value, →	`op_0 = floor(op_0)`
8	d2i	0: Double Value, →	`op_0 = (int) op_0`
9	iconst_1	0: Int Value, →	`op_1 = 1`
10	iadd	0: Int Value, 1: Int Value, →	`op_0 = op_0 + op_1`
11	ireturn	0: Int Value, →	`return op_0;`

Optimizations to get “reasonable” three-address code

1. Peephole optimizations which use a *sliding window* over the cfg’s basic blocks to perform, e.g., the following optimizations:
   - copy propagation
   - elimination of redundant loads and stores
   - constant folding
   - constant propagation
   - common subexpression elimination
   - strength reduction (`x * 2` ⇒ `x + x` ; `x / 2` ⇒ `x >> 1`)
   - elimination of useless instructions (`y = x * 0` ⇒ `y = 0`)
2. Intra-procedural analyses:
   - to type the reference variables
   - *standard optimizations to further minimize the code*

E.g., in Soot many of the steps described above are performed sequentially. OPAL, however, uses a different approach inspired by graph-free data-flow analysis[2] to compute the three-address code representation in two steps: (1) performing the data-flow analysis, (2) generation of the final three address code. This enables OPAL to be faster and more precise.

Static Single Assignment Form

Many analyses require def-use information. I.e., the information where a used local variable is defined or vice versa.

When an intermediate (three-address code based) representation is in SSA (Form) then:

- each variable is assigned exactly once (i.e., it has only one static definition-site in the program text), and
- every variable is defined before it is used.

When two control-flow paths merge, a selector function ϕ is used that initializes the variable based on the control flow that was taken.

Example plain(naive) three-address code

```
static int max(int i, int j) {
  int max;
  if (i > j) max = i; else  max = j;
  return max;
}
```

```
0: if(i <= j) goto 3;
1: r_0 = i;
2: goto 4;
3: r_0 = j;
4: return r_0;
```

In the naive three-address code the def-use/use-def chain is not explicit. To get the respective information a *reaching-definitions* analysis needs to be performed. For example, to determine that `r_0` contains either `i` or `j` a data-flow analysis of the entire method is required.

Example SSA three-address code

```
static int max(int i, int j) {
  int max;
  if (i > j) max = i; else  max = j;
  return max;
}
```

```
0: if(i <= j) goto 3;
1: t_1 = i;
2: goto 4;
3: t_2 = j;
4: t_3 = Φ(t_1,t_2); // <= Control-flow join
   return t_3;
```

In the above code every local variable has a single definition site and it is only initialized at that site. The Φ function is required since the returned value is either (exclusively) `t_1` or `t_2` and this depends on the past control-flow. The information that `t_3` is either `i` or `j` can be deduced by following the use-def chain.

Soot's SSA representation is called Shimple. OPAL offers an SSA-like representation which is called TACAI.

References

1. A. Aho, R. Sethi and J. D. Ullman; Compilers - Principles, Techniques and Tools; Addison Wesley 1988↩

2. Mohnen, M.; A Graph—Free Approach to Data—Flow Analysis. In Compiler Construction (Vol. 2304, pp. 46–61). 2002; DOI↩