Introduction to Compilers and Language Design

You can find the latest version of the PDF edition, and purchase inexpensive hardcover copies at this website:
`http://compilerbook.org`

Draft version: September 14, 2018

Chapter 8 – Intermediate Representations

8.1 Introduction

Most production compilers make use of an **intermediate representation (IR)** that lies somewhere between the abstract structure of the source language and the concrete structure of the target assembly language.

An IR is designed to have a simple and regular structure that facilitates optimization, analysis, and efficient code generation. A modular compiler will often implement each optimization or analysis tool as a separate module that consumes and produces the same IR, so that it is easy to select and compose optimizations in different orders.

It is common for an IR to have a defined **external format** that can be written out to a file in text form, so that it can be exchanged between unrelated tools. Although it may be visible to the determined programmer, it usually isn't meant to be easily readable. When loaded into memory, the IR is represented as a data structure, to facilitate algorithms that traverse its structure.

There are many different kinds of IR that can be used; some are very close to the AST we used up to this point, while others are only a very short distance from the target assembly language. Some compilers even use multiple IRs in decreasing layers of abstraction. In this chapter, we will examine different approaches to IRs and consider their strengths and weaknesses.

8.2 Abstract Syntax Tree

First, we will point out that the AST itself can be a usable IR, if the goal is simply to emit assembly language without a great deal of optimization or other transformations. Once typechecking is complete, simple optimizations like strength reduction and constant folding can be applied to the AST itself. Then, to generate assembly language, you can simply perform a post-order traversal of the AST and emit a few assembly instructions corresponding to each node. [1]

[1] This is the approach we use in a one-semester course to implement a project compiler, since there is a limited amount of time to get to the final goal of generating assembly language.

```
typedef enum {
     DAG_ASSIGN,
     DAG_DEREF,
     DAG_IADD,
     DAG_IMUL,
     ...
     DAG_NAME,
     DAG_FLOAT_VALUE,
     DAG_INTEGER_VALUE
} dag_kind_t;

struct dag_node {
     dag_kind_t kind;
     struct dag_node *left;
     struct dag_node *right;
     union {
          const char *name;
          double float_value;
          int integer_value;
     } u;
};
```

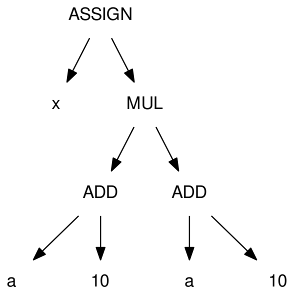
Figure 8.1: Sample DAG Data Structure Definition

However, in a production compiler, the AST isn't a great choice for an IR, mainly because the structure is *too* rich. Each node has a large number of different options and substructure: for example, an addition node could represent an integer addition, a floating point addition, a boolean-or, or a string concatenation, depending on the types of the values involved. This makes it difficult to perform robust transformations, as well as to generate an external representation. A more low-level representation is needed.

8.3 Directed Acyclic Graph

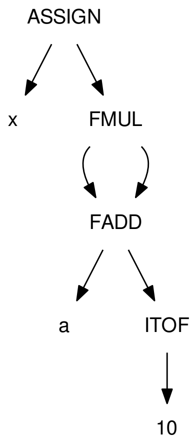
The **directed acyclic graph (DAG)** is one step simplified from the AST. A DAG is similar to the AST, except that it can have an arbitrary graph structure, and the individual nodes are greatly simplified, so that there is little or no auxiliary information beyond the type and value of each node. This requires that we have a greater number of node types, each one explicit about its purpose. For example, Figure 8.1 shows a definition of a DAG data structure that would be compatible with our project compiler.

Now suppose we compile a simple expression like `x=(a+10)*(a+10)`. The AST representation of this expression would directly capture the syntactic structure:

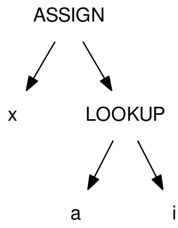


After performing typechecking, we may learn that `a` and `b` are floating point values, and therefore `10` must be converted into a float before performing floating point arithmetic. In addition, the computation `a+10` need only be performed once, and the resulting value used twice.

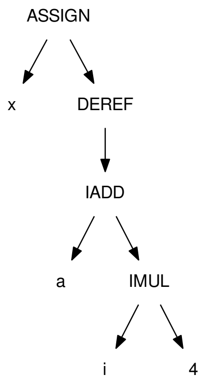
All of that can be represented with the following DAG, which introduces a new type of node `ITOF` to perform integer-to-float conversion, and nodes `FADD` and `FMUL` to perform floating point arithmetic:



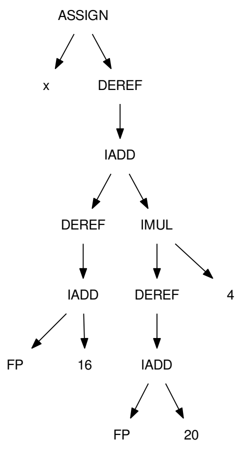
It is also common for a DAG to represent address computations related to pointers and arrays in greater detail, so that they can be shared and optimized, where possible. For example, the array lookup `x=a[i]` would have a very simple representation in the AST:



However, an array lookup is actually accomplished by adding the starting address of the array `a` with the index of the item `i` multiplied by the size of objects in the array, determined by consulting the symbol table. This could be expressed in a DAG like this:



As a final step before code generation, the DAG might be expanded to include the address computations for local variables. For example, if `a` and `i` are stored on the stack at sixteen and twenty bytes past the frame pointer `FP`, respectively, the DAG could be expanded like this:



The **value-number method** can be used to construct a DAG from an AST. The idea is to build an array where each entry consists of a DAG node type, and the array index of the child nodes. Every time we wish to add a new node to the DAG, we search the array for a matching node and re-use it to avoid duplication. The DAG is constructed by performing a post-order traversal of the AST and adding each element to the array.

The DAG above could be represented by this value-number array:

#	Type	Left	Right	Value
0	`NAME`			x
1	`NAME`			a
2	`INT`			10
3	`ITOF`	2		
4	`FADD`	1	3	
5	`FMUL`	4	4	
6	`ASSIGN`	0	5	

Obviously, searching the table for equivalent nodes every time a new node gets added is going to have polynomial complexity. However, the absolute sizes stay relatively small, as long as each individual expression has its own DAG.

By designing the DAG representation such that all necessary information is encoded into the node type, it becomes easy to write a portable external representation. For example, we could represent each node as a symbol, followed by its children in parentheses:

```
ASSIGN(x,DEREF(IADD(DEREF(IADD(FP,16)),
                     IMUL(DEREF(IADD(FP,16)),4)))))
```

Clearly, this sort of code would not be easy for a human to read and write manually, but it is trivial to print and trivial to parse, making it easy to pass between compiler stages for analysis and optimization.

Now, what sort of optimizations might you do with the DAG? One easy optimization is **constant folding**. This is the process of reducing an expression consisting of only constants into a single value. [2] This capability is handy, because the programmer may wish to leave some expressions in an explicit form for the sake of readability or maintainability while still having the performance advantage of a single constant in the executable code.

DAG Constant Folding Algorithm
Examine a DAG recursively and collapse all operators on two constants into a single constant.

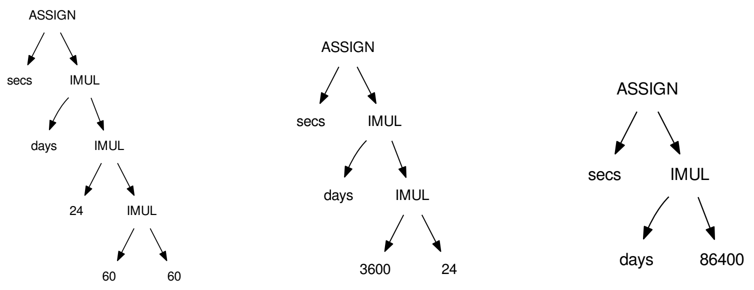
```
ConstantFold( DagNode n ):

If n is a leaf:
    return;
Else:
    n.left = ConstantFold(n.left);
    n.right = ConstantFold(n.right);

    If n.left and n.right are constants:
        n.value = n.operator(n.left,n.right);
        n.kind = constant;
        delete n.left and n.right
```

[2]Constant folding is a narrow example of the more general technique of **partial execution** in which some parts of the program are executed at compile time, while the rest is left for runtime.

Figure 8.2: Example of Constant Folding



Suppose you have an expression that computes the number of seconds present in the number of days. The programmer expresses this as `secs=days*24*60*60` to make it clear that there are 24 hours in a day, 60 minutes in an hour, and 60 seconds in a minute. Figure 8.2 shows how `ConstantFold` would reduce the DAG. The algorithm descends through the tree and combines `IMUL(60,60)` into `3600`, and then `IMUL(3600,24)` into 86400.

8.4 Control Flow Graph

It is important to note that a DAG by itself is suitable for encoding expressions, but it isn't as effective for control flow or other ordered program structures. Common sub-expressions are combined under the assumption that they can be evaluated in any order (consistent with operator precedence) and the values already in the DAG do not change. This assumption does not hold when we consider multiple statements that modify values, or control flow structures that repeat or skip statements.

To reflect this, we can use a **control flow graph** to represent the higher-level structure of the program. The control flow graph is a directed graph (possibly cyclic) where each node of the graph consists of a **basic block** of sequential statements. The edges of the graph represent the possible flows of control between basic blocks. A conditional construct (like `if` or `switch`) results in branches in the graph, while a loop construct (like `for` or `while`) results in reverse edges.

For example, this bit of code:

```
for(i=0;i<10;i++) {
    if(i%2==0) {
        print "even";
    } else {
        print "odd";
    }
    print "\n";
}
return;
```

Would result in this control flow graph:

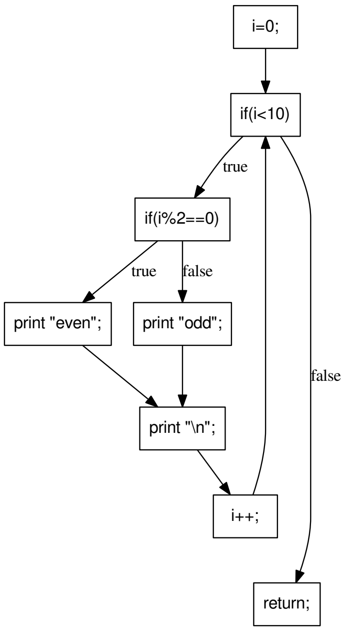


Figure 8.3: Example Control Flow Graph

Note that the control flow graph has a different structure than the AST. The AST for a for-loop would have each of the control expressions as an immediate child of the loop node, whereas the control flow graph places each one in the order it would be executed in practice. Likewise, the `if` statement has edges from each branch of the conditional to the following node, so that one can easily trace the flow of execution from one component to the next.

8.5 Static Single Assignment Form

The **static single assignment (SSA)** [1] form is a commonly-used representation for complex optimizations. SSA uses the information in the control flow and updates each basic block with a new restriction: *variables cannot change their values*. Instead, whenever a variable is assigned a new value, it is given a new version number.

For example, suppose that we have this bit of code:

```
int x = 1;
int a = x;
int b = a + 10;
x = 20 * b;
x = x + 30;
```

We could re-write it in SSA form like this:

```
int x_1 = 1;
int a_1 = x_1;
int b_1 = a_1 + 10;
x_2 = 20 * b_1;
x_3 = x_2 + 30;
```

A peculiarity comes when a variable is given a different value in two branches of a conditional. Following the conditional, the variable could have either value, but we don't know which one. To express this, we introduce a new function $\phi(x,y)$ which indicates that either value x or y could be selected at runtime. The ϕ function may not necessarily translate to an instruction in the assembly output, but serves to link the new value to its possible old values, reflecting the control flow graph.

For example, this code fragment:

```
if(y<10) {
   x=a;
} else {
   x=b;
}
```

Becomes this:

```
if(y_1<10) {
   x_2=a;
} else {
   x_3=b;
}
x_4 = phi(x_2,x_3);
```

8.6 Linear IR

A linear IR is an ordered sequence of instructions that is closer to the final goal of an assembly language. It loses some of the flexibility of a DAG (which does not commit to a specific ordering) but can capture expressions, statements, and control flow all within one data structure. This enables some optimization techniques that span multiple expressions.

There is no universal standard for a linear IR, but it typically looks like an idealized assembly language, with a large (or infinite) number of registers, and the typical arithmetic and control flow operations. Here, let us assume an IR where `LOAD` and `STOR` are used to move values between memory and registers, and three-address arithmetic operations read two registers and write to a third, from right to left. Our example expression would look like this:

```
1. LOAD a        -> %r1
2. LOAD $10      -> %r2
3. ITOF %r2      -> %r3
4. FADD %r1, %r3 -> %r4
5. FMUL %r4, %r4 -> %r5
6. STOR %r5      -> x
```

This IR is easy to store efficiently, because each instruction can be a fixed size 4-tuple representing the operation and (max) of three arguments. The external representation is also straightforward.

As the example suggests, it is most convenient to pretend that there are an infinite number of **virtual registers**, such that every new value computed writes to a new register. In this form, we can easily identify the **lifetime** of a value by observing the first point where a register is written, and the last point where a register is used. Between those two points, the value of register one must be preserved. For example, the lifetime of `%r1` is from instruction 1 to instruction 4.

At any given instruction, we can also observe the set of virtual registers that are live:

```
1. LOAD a        -> %r1      live: %r1
2. LOAD $10      -> %r2      live: %r1 %r2
3. ITOF %r2      -> %r3      live: %r1 %r2 %r3
4. FADD %r1, %r3 -> %r4      live: %r1 %r3 %r4
5. FMUL %r4, %r4 -> %r5      live: %r4 %r5
6. STOR %r5      -> x        live: %r5
```

This observation makes it easy to perform operations related to instruction ordering. Any instruction may be moved to an earlier position (within one basic block) as long as the values it reads are not moved above their definitions. Likewise, any instruction may be moved to a later position as long as the values it writes are not moved below their uses. Moving instructions can reduce the number of physical registers needed in code generation, as well as reduce execution time in a pipelined architecture.

8.7 Stack Machine IR

An even more compact intermediate representation is a **stack machine** IR. Such a representation is designed to execute on a **virtual stack machine** that has no traditional registers, but only a stack to hold intermediate registers. A stack machine IR typically has a `PUSH` instruction which pushes a variable or literal value on to the stack and a `POP` instruction which removes an item and stores it in memory. Binary arithmetic operators (like `FADD` or `FMUL`) implicitly pop two values off the stack and push the result on the stack, while unary operators (`ITOF`) pop one value and push one value. A few utility instructions are needed to manipulate the stack, like a `COPY` instruction which pushes a duplicate value on to the stack.

To emit a stack machine IR from a DAG, we simply perform a post-order traversal of the AST and emit a `PUSH` for each leaf value, an arithmetic instruction for each interior node, and a `POP` instruction to assign a value to a variable.

Our example expression would look like this in a stack machine IR:

```
PUSH a
PUSH 10
ITOF
FADD
COPY
FMUL
POP x
```

And if executed directly, the IR would work like this:

IR Op:	`PUSH a`	`PUSH 10`	`ITOF`	`FADD`	`COPY`	`FMUL`	`POP x`
Stack	5.0	10	10.0	15.0	15.0	225.0	-
State:	-	5.0	-	-	15.0	-	-

A stack machine IR has many advantages. It is much more compact than a 3-tuple or 4-tuple linear representation, since there is no need to record the details of registers. It is also straightforward to implement this language in a simple interpreter.

However, a stack-based IR is slightly more difficult to translate to a conventional register-based assembly language, precisely because the explicit register names are lost. Further transformation or optimization of this form requires that we transform the implicit information dependencies in the stack-basic IR back into a more explicit form such as a DAG or a linear IR with explicit register names.

8.8 Examples

Nearly every compiler or language has its own intermediate representation with some peculiar local features. To give you a sense of what's possible, this section compares three different IRs used by compilers in 2017. For each one, we will show the output of compiling this simple arithmetic expression:

```
float f( int a, int b, float x ) {
    float y = a*x*x + b*x + 100;
    return y;
}
```

8.8.1 *GIMPLE - GNU Simple Representation*

The **GNU Simple Representation (GIMPLE)** is an internal IR used at the earliest stages of the GNU C compiler. GIMPLE can be thought of as a drastically simplified form of C in which all expressions have been broken down into individual operators on values in static single assignment form. Basic conditionals are allowed, and loops are implemented using `goto`.

Our simple function yields the following GIMPLE. Note that each SSA value is declared as a local variable (with a long name) and the type of each operator is still inferred from the local type declaration.

```
f (int a, int b, float x)
{
  float D.1597D.1597;
  float D.1598D.1598;
  float D.1599D.1599;
  float D.1600D.1600;
  float D.1601D.1601;
  float D.1602D.1602;
  float D.1603D.1603;
  float y;

  D.1597D.1597 = (float) a;
  D.1598D.1598 = D.1597D.1597 * x;
  D.1599D.1599 = D.1598D.1598 * x;
  D.1600D.1600 = (float) b;
  D.1601D.1601 = D.1600D.1600 * x;
  D.1602D.1602 = D.1599D.1599 + D.1601D.1601;
  y = D.1602D.1602 + 1.0e+2;
  D.1603D.1603 = y;
  return D.1603D.1603;
}
```

8.8.2 LLVM - Low Level Virtual Machine

The Low Level Virtual Machine (LLVM)[3] project is a language and a corresponding suite of tools for building optimizing compilers and interpreters. A variety of compiler front-ends support the generation of LLVM intermediate code, which can be optimized by a variety of independent tools, and then translated again into native machine code, or bytecode for virtual machines like Oracle's JVM, or Microsoft's CLR.

Our simple function yields this LLVM. Note that the first few `alloca` instructions allocate space for local variables, followed by `store` instructions that move the parameters to local variables. Then, each step of the expression is computed in SSA form and the result stored to the local variable `y`. The code is explicit at each step about the type (32-bit integer or float) and the alignment of each value.

```
define float @f(i32 %a, i32 %b, float %x) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca float, align 4
  %y = alloca float, align 4
  store i32 %a, i32* %1, align 4
  store i32 %b, i32* %2, align 4
  store float %x, float* %3, align 4
  %4 = load i32* %1, align 4
  %5 = sitofp i32 %4 to float
  %6 = load float* %3, align 4
  %7 = fmul float %5, %6
  %8 = load float* %3, align 4
  %9 = fmul float %7, %8
  %10 = load i32* %2, align 4
  %11 = sitofp i32 %10 to float
  %12 = load float* %3, align 4
  %13 = fmul float %11, %12
  %14 = fadd float %9, %13
  %15 = fadd float %14, 1.000000e+02
  store float %15, float* %y, align 4
  %16 = load float* %y, align 4
  ret float %16
}
```

[3] http://llvm.org

8.8.3 JVM - Java Virtual Machine

The **Java Virtual Machine (JVM)** is an abstract definition of a stack-based machine. High-level code written in Java is compiled into `.class` files which contain a binary representation of the JVM bytecode. The earliest implementations of the JVM were interpreters which read and executed the JVM bytecode in the obvious way. Later implementations performed just-in-time (JIT) compiling of the bytecode into native assembly language, which can be executed directly.

Our simple function yields the following JVM bytecode. Note that each of the `iload` instructions refers to a local variable, where parameters are considered as the first few local variables. So, `iload 1` pushes the first local variable (`int a`) on to the stack, while `fload 3` pushes the third local variable (`float x`) on to the stack. Fixed constants are stored in an array in the class file and referenced by position, so `ldc #2` pushes constant in position two (`100`) on to the stack.

```
 0: iload   1
 1: i2f
 2: fload   3
 4: fmul
 5: fload   3
 7: fmul
 8: iload   2
 9: i2f
10: fload   3
12: fmul
13: fadd
14: ldc     #2
16: fadd
17: fstore  4
19: fload   4
21: freturn
```

8.9 Exercises

1. Add a step to your compiler to convert the AST into a DAG by performing a post-order traversal and creating one or more DAG nodes corresponding to each AST node.

2. Write the code to export a DAG in a simple external representation as shown in this chapter. Extend the DAG suitably to represent control flow structures and function definitions.

3. Write a scanner and parser to read in the DAG external representation and reconstruct it as a data structure. Think carefully about the grammar class of the IR, and choose the simplest implementation that works.

4. Building on steps 2 and 3, write a standalone optimization tool that reads in the DAG format, performs a simple optimization like constant folding, and writes the DAG back out in the same format.

8.10 Further Reading

1. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Kenneth Zadeck. "Efficiently computing static single assignment form and the control dependence graph." ACM Transactions on Programming Languages and Systems (TOPLAS) volume 13, number 4, 1991.
   https://doi.org/10.1145/115372.115320

2. J. Merrill, "Generic and GIMPLE: A new tree representation for entire functions." GCC Developers Summit, 2003.

3. C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", IEEE International Symposium on Code Generation and Optimization, 2004.
   https://dl.acm.org/citation.cfm?id=977673