Chapter 1

Optimization

1.1 Intermediate code

Intermediate code uses a language between the source language and the target language. It provides an intermediate level of abstraction: more details than the source and fewer details than the target. High-level source languages like `Cool` and `C` reveals less low-level conceptions such as registers, making it difficult to find room for optimization, while low-level languages like assembly language are often limited to a specific type of machine architecture.

We will introduce the conception with an intermediate language that can be called a "high level assembly language". It uses register names, but has an unlimited number of registers. It uses control structures like assembly language. Opcodes are used, but some of them are higher level (e.g. `push`, which will be translated into a few assembly instructions on a machine of a particular architecture). Each instruction is either binary ($\mathtt{x} \coloneqq \mathtt{y}\ \mathtt{op}\ \mathtt{z}$) or unary($\mathtt{x} \coloneqq \mathtt{op}\ \mathtt{y}$). Arguments on the right are always registers or constants. This is actually a wide-used form of intermediate code called **three-address code**. In this language, every intermediate value will have its own name.

Generating intermediate code is similar to generating assembly code, except that unlimited number of registers can be used, which renders easier code generation. If we use $\mathtt{igen}(\mathtt{e},\mathtt{t})$ to denote the function that generates code for expression `e` and stores the result in register `t` , we will have

$$\begin{aligned}
&\mathtt{igen}(\mathtt{e}_1 + \mathtt{e}_2, \mathtt{t}) = \\
&\qquad \mathtt{igen}(\mathtt{e}_1, \mathtt{t}_1) \\
&\qquad \mathtt{igen}(\mathtt{e}_2, \mathtt{t}_2) \\
&\qquad \mathtt{t} \coloneqq \mathtt{t}_1 + \mathtt{t}_2
\end{aligned}$$

1.2 Optimization overview

Optimizations can be performed at different times:

1. On AST

   Pro Machine independent

   Con Too high level

2. On assembly language

   Pro Exposes the most optimization opportunities

   Con Machine dependent. Has to be reimplemented when re-targeting to different architectures.

3. On intermediate language

   Pro Machine independent. Exposes optimization opportunities.

We will discuss optimizations performed on intermediate languages. The intermediate language we use can be described with the following CFG:

```
P → SP | S
S → id := id op id
    | id := op id
    | push id
    | id := pop
    | if id relop id goto L
    | L :
    | jump L
```

`Id`'s are registers, and can be substituted by constants when they serve as arguments. Typical operations are +,-,*,/.

A **basic block** is a maximal sequence of instructions with no labels (except for the first instruction) and no jumps(except for the last instruction). The execution can only jump into a basic block at the first instruction and jump out of it at the last instruction. There is no other way to jump into it, and all instructions inside the block are guaranteed to be executed sequentially before the execution jumps out. This property enables us to conduct a series of optimizations.

A **control flow graph** is a directed graph with basic blocks as nodes. An edge from block A to block B exists if the execution can pass from the last instruction in A to the first instruction in B. The body of a method can be represented as a control flow graph. There is one initial node, and all "return" nodes are terminals.

The purpose of optimization is to **improve a program's resource utilization**. Most of the time we try to make the program run faster, i.e. reduce the execution time. Other resources that optimization could be concerned about are code size, memory footprint, network messages sent, disk accesses, power, etc. The bottom line is that optimization should not alter what the program computes.

For languages like `C` and `Cool` there are 3 granularities of optimizations:

Local optimization Applies to an isolated basic block.

Global optimization Applies to an isolated control flow graph (method body).

Inter-procedural optimization Applies across method boundaries.

Local optimizations are performed by most mainstream compilers. Many of them also perform global optimizations. Few compilers touch inter-procedural optimizations, not only because it's hard to implement, but also because it often does not provide as much improvement as the first two. In practice, it is usually a wise decision not to implement the fanciest optimizations because they tend to be hard to implement, costly in compile time while not much payoff can be gained.

1.3 Local optimization

Local optimization focuses on a single basic block. There is no need to analyze the entire method body.

1.3.1 Constant folding

For an instruction $\mathtt{x} := \mathtt{y}\,\mathtt{op}\,\mathtt{z}$ in which `y,z` are both constants, $\mathtt{y}\,\mathtt{op}\,\mathtt{z}$ can be computed at compile time. E.g., $\mathtt{x} := 2 + 2 \Rightarrow \mathtt{x} := 4$.

Constants folding can be dangerous when the compiler and the target code it generates are run on different machines, which is not uncommon in reality, e.g. in most embedded platforms. The two machines might feature different round-offs of floating point numbers. If we do constant folding according to the floating point semantics of the compile machine directly at compile time, we may end up with unwanted result at runtime. An obvious solution is to keep full precision inside the compiler and represent floating pointer numbers as string literals, and leave it to the runtime machine to handle the round offs.

1.3.2 Eliminate unreachable basic blocks

By eliminating basic blocks that cannot be reached from the initial block, we can make the program smaller, and sometimes faster, because of cache effects.

1.3.3 Common subexpression elimination

Some optimizations can be simplified if each register occurs only once on the lhs of an assignment. Intermediate code can be rewritten into **single assignment form** by introducing new registers to substitute earlier appearances of registers that are assigned more than once.

If a basic block is in single assignment form and a definition `x :=` is the first use of `x` in a block, then when two assignments have the same rhs, they are guaranteed to compute the same value, which allows us saving the trouble of computing the same expression twice. We can simply substitute the second appearance of the rhs with the register assigned in its first appearance.

1.3.4 Copy propagation

if `w := x` appears in a block, subsequent use of `w` can be replaced with `x`.

1.3.5 Dead instruction elimination

if `w := rhs` appears in a basic block, and `w` does not appear anywhere else in the program, then the instruction is dead and can be eliminated.

Each local optimization does little by itself, but typically optimizations will interact with each other. Performing one optimization might enable another one. Thus the usual approach is to iterate until no more improvements can be made. The optimizer can also be stopped at any point to limit compilation time.

1.3.6 Peephole optimization

Local optimizations can be applied directly to assembly code rather than to intermediate code. **Peephole optimization** is effective for improving assembly code. The "peephole" is a short sequence of contiguous instructions. The optimizer replaces the sequence with an equivalent but faster sequence. The process is repeated for maximum effect.

1.4 Global optimization

1.4.1 Dataflow analysis

In order to replace a use of `x` by a constant `k` we must make sure that on every path to the use of `x`, the last assignment to `x` is `x := k`. This condition is not trivial to check, considering the existence of loops and conditional branches. Global **dataflow analysis** is required to check this condition.

Global optimization tasks share several common traits:

- The optimization depends on knowing a property X at a particular point in program execution.
- Proving X at any single point requires knowledge of the entire program.
- It is OK to be conservative. If the optimization requires X being true, then we want to figure out whether X is definitely true or we don't know if X is true. It is always safe to say "don't know".

1.4.2 Global constant propagation