Compiler Design — Lecture note week 3

- Author: Ruben Schenk
- Date: 12.10.2021
- Contact: ruben.schenk@inf.ethz.ch

3.7.5 Directly Generating x86

For simple languages, there is no need for intermediate representations, e.g. the arithmetic expression language as in `(X1 + X2) - X3`. The main idea is to **maintain invariants**, e.g. code emitted for a given expression computes the answer into `rax`.

The key challenges are:

- storing intermediate values needed to compute complex expressions
- some instructions use specific registers (e.g. shift)

One simple strategy for directly generating x86 is based on the following ideas:

- Compilation emits instructions into an instruction stream
- To compile an expression `e1 op e2`:
    a. Recursively compile its sub-expressions
    b. Process the results
- Invariants:
    - Compilation of an expression yields its result in `rax`
    - Argument `Xi` is store in a dedicated operand
    - Intermediate values are pushed onto the stack
    - The stack is popped after use (such that the space is reclaimed)
- Resulting code is wrapped to comply with cdecl calling conventions

4. Intermediate Representations

Up until now, we followed a simple *syntax-directed* translation, this meant that:

- Input syntax uniquely determined the output, i.e. no complex analysis or code transformation was done
- Worked fine for simple languages

However, the resulting *code quality is poor*. Example: The expression `(x1 - x1) + 3` is turned into the following code:

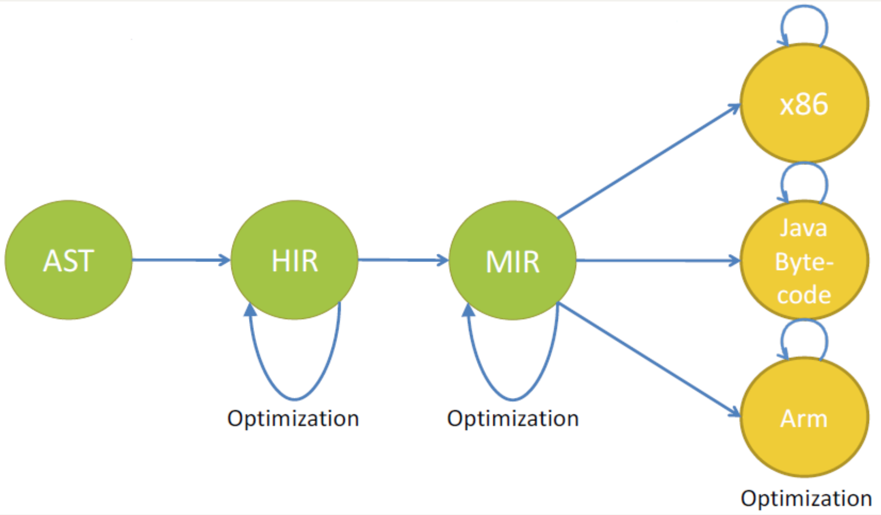
```
.text
.globl _program

_program:
    pushq       %rbp
    movq        %rsp, %rbp
    movq        %rdi, %rax
    pushq       %rax
    movq        %rdi, %rax
    imulq       $-1, %rax
    popq        %r10
    addq        %r10, %rax
    pushq       %rax
    movq        $3, %rax
    popq        %r10
    addq        %r10, %rax
    popq        %rbp
    retq
```

But, obviously `(x1 - x1)` is 0 and the program therefore could be much more simple.

4.1 Intermediate Representations (IR's)

Abstract machine code (IR) hides the details of the target architecture and allows machine independent code generation and optimization.

The goal of this is to get the program closer to machine code without losing the information needed to do analysis and optimization. We might also have multiple IR's.



4.1.1 What makes a good IR?

A good IR should tick the following points:

- Easy translation target (from the level above)
- Easy to translate (to the level below)
- Narrow interface (fewer constructs means simpler phases/optimizations)

Example: Source languages may have "while", "for", and "for each" loops while the IR might only have "while" loops and sequencing.
A "for" loop may be translated as follows:

```
    [[for(pre; cond; post) {body}]]
    [[pre; while(cond) {body; post}]]
```

Remark: Here, the notation `[[cmd]]` denotes the "translation/compilation" of `cmd`.

4.2 Simple Let-Based IR (SLL)

4.2.1 Eliminating Nested Expressions

One fundamental problem is compiling complex and nested expression forms to simple operations. As an example, consider the following steps of translations:

```
# Source
((1 + X4) + (3 + (X1 * 5)))

# AST
Add(Add(Const 1, Var X4),
    Add(Const 3, Mul(Var X1,
        Const 5)))
```

How would the IR-translation of the above example look like? The idea is to name intermediate values and to make the order of evaluation explicit, i.e. not to have any nested operations.

4.2.2 Translation to SLL

Considering the given example in the previous section, we can translate the code to the desired SLL form:

```
let tmp0 = add 1L varX4 in
let tmp1 = mul varX1 5L in
let tmp2 = add 3L tmp1 in
let tmp3 = add tmp0 tmp2 in
    tmp3
```

We take the following notes on the translation above:

- Translation makes the order of evaluation explicit
- Translation names intermediate values
- Introduced temporaries are never modified

4.2.3 Basic Blocks and CFGs

A **basic block** is a sequence of instructions that is always executed starting at the first instruction and always exits at the last instruction:

- Starts with a label that names the *entry point* of the basic block
- Ends with a control-flow instruction (e.g. branch or return), i.e. the *link*
- Contains no other control-flow instruction
- Contains no interior label used as a jump target

Basic blocks can be arranged into a **control-flow graph (CFG)**:

- The nodes of the graph are basic blocks
- There is a directed edge from node A to node B if the control flow instruction at the end of

block A might jump to the label of block B