Compilation and Program Analysis (#6) :

Intermediate Representations: CFG, DAGs (Instruction Selection and Scheduling), SSA

Laure Gonnord

http://laure.gonnord.org/pro/teaching/capM1.html

Laure.Gonnord@ens-lyon.fr

Master 1, ENS de Lyon

2018-2019

Big picture

source code

↓

lexical+syntactic analysis + typing

↓

decorated AST

↓

code production (numerous phases)

↓

assembly language

In context 1/2

In the last course we saw the need for a better data structure to propagate and infer information. We need :

- A data structure that helps us to reason about the flow of the program.
- Which embeds our three address code.

► Control-Flow Graph.

In context 2/2

decorated AST

↓

IR Construction

↓

Control-Flow Graph

↓

Clever analyses/code generation

↓

assembly language

Definitions

Definition (Basic Block)

Basic block : largest (3-address TARGET18*) instruction sequence without label. (except at the first instruction) and without jumps and calls.*

Definition (CFG)

It is a directed graph whose vertices are basic blocks, and edge $B_1 \to B_2$ *exists if* B_2 *can follow immediately* B_1 *in an execution.*

- two optimisation levels : local (BB) and global (CFG)

Identifying Basic Blocks (from 3@code)

- The first instruction of a basic block is called a **leader**.
- We can identify leaders via these three properties :
  1. The first instruction in the intermediate code 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.
- Once we have found the leaders, it is straighforward to find the basic blocks : for each leader, its basic block consists of the leader itself, plus all the instructions until the next leader.

Exercise

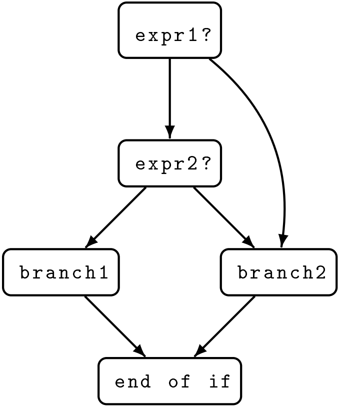
Generate the "high level" CFG for the given program :

```
p:=0;i:=1;
while (i <= 20) do
  if p>60 then
     p:=0;i:=5;
  endif
  i:=2*i+1;
done
k:=p*3;
```

(inside your compiler, blocks will be a list of 3@-TARGET18 code)

CFG for tests

```
if (expr1 and expr2)
   ...branch1...
else
   ...branch2...
```

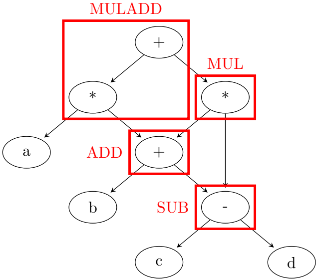


(blocks are subgraphs)

Big picture

- Front-end → a CFG where nodes are basic blocks.
- Basic blocks → DAGs that explicit common computations

```
u1 := c - d
u2 := b + u1
u3 := a * u2
u4 := u2 * u1
u5 := u3 + u4
```



▶ choose instructions(**selection**) and order them (**scheduling**).

Instruction Selection

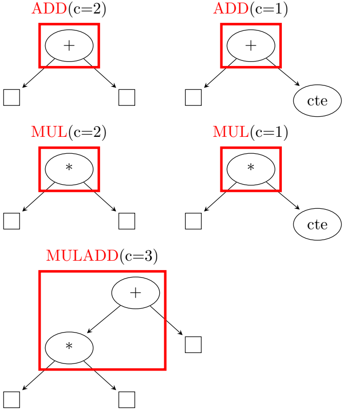
The problem of selecting instructions is a DAG-partitioning problem. But what is the objective ?

The best instructions :

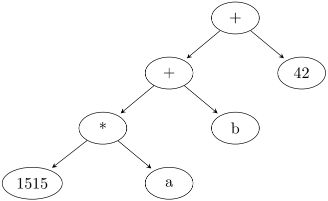
- cover bigger parts of computation.
- cause few memory accesses.

► Assign a cost to each instruction, depending on their addressing mode.

Instruction Selection : an example



What is the optimal instruction selection for :



- Finding a tiling of minimal cost : it is **NP-complete** (SAT reduction).

Tiling trees / DAGs, in practise

For tiling :

- There is an optimal algorithm for **trees** based on dynamic programing.
- For DAGs we use heuristics (decomposition into a forest of trees, . . .)

▶ The litterature is pletoric on the subject.

1. Control flow Graph

2. Basic Bloc DAGs, instruction selection/scheduling
   - Instruction Selection
   - Instruction Scheduling

3. Other IRS

Instruction Scheduling, what for ?

We want an evaluation order for the instructions that we choose with **Instruction Scheduling**.

A scheduling is a function θ that associates a **logical date** to each instruction. To be correct, it must respect data dependancies :

```
(S1) u1 := c - d
(S2) u2 := b + u1
```

implies $\theta(S_1) < \theta(S_2)$.

- How to choose among many correct schedulings ? depends on the target architecture.

Architecture-dependant choices

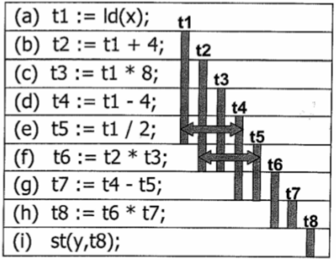
The idea is to exploit the different ressources of the machine at their best :

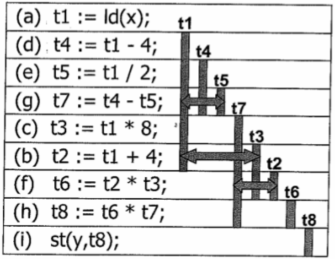
- instruction parallelism : some machine have parallel units (subinstructions of a given instruction).
- prefetch : some machines have non-blocking load/stores, we can run some instructions between a load and its use (hide latency !)
- pipeline.
- registers : see next slide.

(sometimes these criteria are incompatible)

Register use

Some schedules induce less **register pressure** :





- How to find a schedule with less register pressure ?

Scheduling wrt register pressure

Result : this is a linear problem on trees, but NP-complete on DAGs (Sethi, 1975).

- Sethi-Ullman algorithm on trees, heuristics on DAGs

Sethi-Ullman algorithm on trees

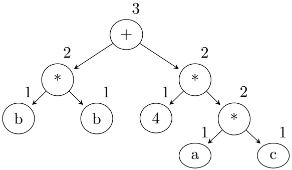
$\rho(node)$ denoting the number of (pseudo)-registers necessary to compute a node :

- $\rho(leaf) = 1$
- $\rho(nodeop(e_1, e_2)) = \begin{cases} max\{\rho(e_1), \rho e_2\} & \text{if } \rho(e_1) \neq \rho(e_2) \\ \rho(e_1) + 1 & \text{else} \end{cases}$

(the idea for non “balanced” subtrees is to execute the one with the biggest ρ first, then the other branch, then the op. If the tree is balanced, then we need an extra register)

▶ then the code is produced with postfix tree traversal, the biggest register consumers first.

Sethi-Ullman algorithm on trees - an example



	tmp_1	tmp_2	tmp_3	tmp_4
`mul tmp1, b b`				
`mul tmp2, a c`	■			
`leti tmp3, 4`	■	■		
`mul tmp4, tmp2, tmp3`	■	■	■	
`mul tmp5, tmp1 ,temp4`	■			■

Conclusion (instruction selection/scheduling)

Plenty of other algorithms in the literature :

- Scheduling DAGs with heuristics, ...
- Scheduling loops (M2 course on advanced compilation)

Practical session :

- we have (nearly) no choice for the instructions in the TARGET18 ISA.
- evaluating the impact of scheduling is a bit hard.

We won't implement any of the previous algorithms.

SSA

Later.