Compilation and Program Analysis (#6) :
Intermediate Representations: CFG, DAGs (Instruction Selection and Scheduling), SSA

Laure Gonnord & Ludovic Henrio & Matthieu Moy

https://compil-lyon.gitlabpages.inria.fr/compil-lyon/
Prenom.Nom@ens-lyon.fr

Master 1, ENS de Lyon

2019-2020
Big picture

source code
↓
lexical+syntactic analysis + typing
↓
decorated AST
↓
code production (numerous phases)
↓
assembly language
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
1. Control flow Graph

2. Basic Bloc DAGs, instruction selection/scheduling

3. SSA Control Flow Graph
Definitions

Definition (Basic Block)

Basic block: largest (3-address RISC-V) 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 $B_1 \rightarrow B_2$ exists if $B_2$ can follow immediately $B_1$ in an execution.

- two optimisation levels: local (BB) and global (CFG)
Let us consider the program:

```c
int x, y;
if (x < 4) y = 7; else y = 42;
x = 10;
```

We already generated the (linear code) for a large part of it.
An example 2/2

```
li temp3, 4
li temp2, 0
geq temp0, temp3, lbl0
li temp2, 1
lbl0:  # if false, jump
eq tmp2, zero, lelse1
li temp4, 7
mv temp1, temp4  # y gets 7
jump lendif1
lelse1:
  li temp4 42
  mv temp1, temp4  # y gets 42
lendif1:
  li temp5, 10
  mv temp0, temp5  # end
```
An example 2/2

\begin{verbatim}
li  temp3, 4
li  temp2, 0
geq temp0, temp3, lbl0
li  temp2, 1
lbl0:  # if false, jump
eq tmp2, zero, lelse1
li  temp4, 7
mv  temp1, temp4  # y gets 7
jump lendif1
lelse1:
li  temp4 42
mv  temp1, temp4  # y gets 42
lendif1:
li  temp5, 10
mv  temp0, temp5  # end
\end{verbatim}
Identifying Basic Blocks (from 3 address 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 straightforward to find the basic blocks: for each leader, its basic block consists of the leader itself, plus all the instructions until the next leader.
1. Control flow Graph

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

3. SSA Control Flow Graph
Big picture

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

\[
\begin{align*}
    u_1 &:= c - d \\
    u_2 &:= b + u_1 \\
    u_3 &:= a \times u_2 \\
    u_4 &:= u_2 \times u_1 \\
    u_5 &:= u_3 + u_4
\end{align*}
\]

▶ choose instructions (selection) and order them (scheduling).
1. Control flow Graph

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

3. SSA Control Flow Graph
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 practice

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

▶ The literature is plethoric on the subject.
1. Control flow Graph

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

3. SSA Control Flow Graph
Instruction Scheduling, what for?

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

A scheduling is a function $\theta$ 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 resources of the machine at their best:

- instruction parallelism: some machines 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

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

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

(the idea for non “balanced” subtrees is to execute the one with the biggest \( \rho \) 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

```
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 RISCV ISA.
- evaluating the impact of scheduling is a bit hard.

We won’t implement any of the previous algorithms.
1. Control flow Graph

2. Basic Bloc DAGs, instruction selection/scheduling

3. SSA Control Flow Graph
Credits

The Static Single Assignment Form

• This name comes out of the fact that each variable has only one definition site in the program.
• In other words, the entire program contains only one point where the variable is assigned a value.
• Were we talking about Single Dynamic Assignment, then we would be saying that during the execution of the program, the variable is assigned only once.

Variable $i$ has two static assignment sites: at $L_0$ and at $L_4$; thus, this program is not in Static Single Assignment form. Variable $s$, also has two static definition sites. Variable $x$, on the other hand, has only one static definition site, at $L_2$. Nevertheless, $x$ may be assigned many times dynamically, i.e., during the execution of the program.
A first Example (Cytron 1991)

Each variable is assigned only once (Static Single Assignment form):

```
x ← 5
x ← x - 3
x < 3 ?

y ← 2 * x
w ← y

w ← x - y
z ← x + y
```

```
x_1 ← 5
x_2 ← x_1 - 3
x_2 < 3 ?

y_2 ← x_2 - 3
y_1 ← 2 * x_2
w_1 ← y_1

y_3 ← phi(y_1, y_2)
w_2 ← x_2 - y_3
z_1 ← x_2 + y_3
```
Pro/cons

- Another IR, and cost of construction/deconstruction

+ (some) Analyses/optimisations are easier to perform (like register allocation):
  