CSE 302: Compilers | Lab 5

SSA and Register Allocation

Starts:	`2020-11-05`
Checkpoint 1:	`2020-11-12 23:59:59`
Lab due:	**`2020-11-19 23:59:59`**

1 INTRODUCTION AND STRUCTURE

In this lab you will transform the control flow graphs (CFG) you built for BX2 into *static single assignment* (SSA) form. You will then use liveness information to compute *chordal interference graphs*, which you will then perform register allocation using the *max cardinality search* algorithm. In the end you will be able to produce a version of TAC that can easily be converted to x64 assembly that uses registers instead of stack slots whenever possible.

This lab will involve no changes to the BX2 source language.

This lab will be assessed. It is worth **15%** of your final grade. You should work in groups of 2–3.

This lab has one checkpoint (worth 50% credit). Keep in mind that partial credit is given for incomplete attempts. Make sure to submit something on every due date regardless of how far you get. This assignment sheet is written in the same order as the checkpoint progression. The deliverables for the checkpoints are enumerated in their own subsections, specifically 2.4, 3.6.

Covid-19

Observe all regulations and health guidelines—particularly the lockdown, curfew, and interpersonal distancing guidelines—when working in groups. Use a video-conferencing platform to collaborate.

Continues on next page...

```
class Gvar:  ## global variable
    def __init__(self, name, value):
        # name: string
        # value: 64-bit signed integer

class Instr: ## instruction
    def __init__(self, dest, opcode, arg1, arg2):
        # opcode: string
        # dest, arg1, arg2: varies, but usually temporaries

class Proc:  ## procedure
    def __init__(self, name, t_args, body):
        # t_args: list of argument temporaries
        # body: list of Instr

def load_tac(tac_filename):
    # Parses the file with name `tac_filename' as TAC
    # Returns a list of Gvars and Procs

def execute(tac_prog, proc, args):
    # Interpret the given TAC program, `tac_prog'
    # tac_prog: a 2-tuple of dictionaries (gvars, procs) where:
    #            gvars: maps names to Gvar
    #            procs: maps names to Proc
    # proc: the procedure to execute
    # args: the arguments (as a tuple)
```

Figure 1: `tac.py`: TAC representation, and API

2 STATIC SINGLE ASSIGNMENT FORM (SSA)

This lab will focus on the backend of the compiler – going from TAC to x64. Therefore, we are going to largely ignore the frontend. You can still use your `bx2tac.py` program that you wrote during Lab 4 to generate the TAC files from BX2.

2.1 *TAC and CFG*

You are encouraged to start from your own code for lab 4. However, if you wish, you can also start from the following files:

- A TAC implementation in `tac.py` that has the API shown in figure 1.
- A simple implementation of control flow graphs in `cfg.py`, together with inference and linearization functions (`cfg.infer()` and `cfg.linearize()`). The full API is in figure 2.

This `tac.Instr` class is implemented in such a way that it can be used to index a dictionary. Thus, the Def/Use/Live sets can be easily built as dictionaries mapping instructions to sets of temporaries. In order to compute these sets, you will sometimes need to iterate over every instruction or every instruction pair in the entire CFG. The `cfg.CFG` class provides two iterators for this purpose: `.instrs()` to iterate over the instructions, and `.instr_pairs()` to iterate over instruction pairs. For instance, here is how you might initialize the Live-In set based on a function `use_set(instr)` that returns the temporaries that are used (read) in `instr`:

```python
class Block:  ## basic blocks
    def __init__(self, label):
        self.label = label # canonical entry label
        self.body = []     # list of tac.Instr (no jumps, ret, or label)
        self.jumps = []    # list of tac.Instr (only jumps or ret)

    def instrs(self):
        # Returns an iterator over all instructions in the block
        # (i.e., over self.body + self.jumps)

class CFG:   ## control flow graph
    def __init__(self, proc_name, lab_entry, blocks):
        # proc_name: name of the proc
        # lab_entry: label of the entry block
        # blocks: list of Block
        #   (blocks will be identified by their labels)

    def __getitem__(self, lab):
        # Return the block with the input label `lab'
        # Recall that cfg.__getitem__(lab) can be written as cfg[lab]
    def successors(self, lab_from):
        # Returns an iterator over immediate successor blocks
    def predecessors(self, lab_to):
        # Returns an iterator over immediate successor blocks
    def edges(self):
        # Returns an iterator over all the edges. Each edge is
        # represented as 2-tuples of (source, target) labels.

    def add_node(self, block):
        # Add new block to the CFG
    def remove_node(self, block):
        # Remove block and all its input/output edges from the CFG

    def add_edge(self, lab_from, lab_to):
        # Add directed edge
    def remove_edge(self, lab_from, lab_to):
        # Remove directed edge (if exists)

    def instrs(self):
        # Returns an iterator over all non-label instructions in all
        # blocks in the CFG (in some arbitrary order)

    def instr_pairs(self):
        # Returns an iterator over all pairs of instructions (i1, i2)
        # such that i2 can be executed immediately after i1 in some
        # execution path

# ------------------------------------------------------------------------------

def infer(tac_proc):
    # tac_proc:    a tac.Proc
    # returns:     its corresponding CFG
    # side-effect: tac_proc is left with an empty body

def linearize(tac_proc, cfg):
    # tac_proc: a tac.Proc
    # cfg:      a CFG
    # Replaces tac_proc.body with a linearized form of cfg.
    # After this call, cfg is left in some unknown state and
    # should no longer be used!
```

Figure 2: `cfg.py`: CFG representation of TAC procedures

```
# suppose cfg is a CFG
livein = dict()
for instr in cfg.instrs():
    livein[instr] = use_set(instr)
```

Then, to run the steps of the live-in computation you could write it as follows:

```
# run the livein set update loop until there are no more changes
dirty = True
while dirty:
    dirty = False
    for (i1, i2) in cfg.instr_pairs():
        t_live = livein[i2].difference(def_set(i1))
        if not t_live.issubset(livein[i1]):
            dirty = True # there was a change, so run it again
            livein(i1).update(t_live)
```

2.2 *Crude SSA Generation*

SSA is represented using *versioned temporaries* and *ϕ-functions*. In order to define them, we slightly enlarge the TAC language as follows:

⟨instr⟩ ::= · · · | ⟨var⟩ `'='` `'phi'` `'('` ⟨phi-args⟩ `')'` `';'`

⟨phi-args⟩ ::= ϵ | ⟨phi-args1⟩
⟨phi-args1⟩ ::= ⟨phi-arg⟩ | ⟨phi-arg⟩ `','` ⟨phi-args1⟩
⟨phi-arg⟩ ::= `LABEL` `':'` `TEMP`

`TEMP` ← `%(_|0|[1-9][0-9]*|[A-Za-z][A-Za-z0-9_]*)(\.[0-9]+)?`

To represent a ϕ-function definition, use a `tac.Instr` with opcode `phi`, taking one argument which is the argument list as a dictionary mapping labels to temporaries. So, for example, `%0.0 =` ϕ`(.L1:%0.1, .L2:%0.3);` is represented as:

```
    Instr('%0.0', 'phi', {'.L1': '%0.1', '.L2': '%0.3'}, None)
```

As mentioned in the lecture, the crude SSA generation algorithm has the following steps:

Algorithm: Crude SSA Generation

1. Add ϕ-function definitions for all temporaries that are live-in at the start of each block. These ϕ-function definitions must come before any other instruction in the block.
2. Uniquely version every temporary that is def'd by any instruction in the entire CFG.
3. Update the uses of each temporary within the same block to their most recent versions.
4. For every edge in the CFG (use the `.edges()` iterator) fill in the arguments of the ϕ functions. For an edge from block `.L1` to `.L2`, there would be a ϕ-argument (in the `.L2` block) of the form `.L1:%n` for every temporary `%n` that comes to `.L2` from `.L1`.

An example of crude SSA generation was worked out in detail in lecture 8.

2.3 *SSA Minimization*

Once you have a crude SSA, you will need to perform a number of minimization transformations (in any order) to simplify and eliminate redundancies in the SSA. For this course we will only implement two minimization procedures: *null choice elimination* and *rename elimination*.

Null Choice Elimination This step removes redundant ϕ-function definitions from the SSA.

Null Choice Elimination

The following kind of instruction may be removed anywhere it occurs.

```
    %1.i = ϕ(.L₁:%1.i, .L₂:%1.i, …, .Lₖ:%1.i);
```

Here `%1` stands for any temporary root, and i stands for any version.

Rename Elimination A *rename* is a ϕ-function definition where, once you remove the temporary in the LHS from the arguments of the ϕ-function, you are left with only a single temporary in the arguments (possibly repeated). This case is almost exactly like a `copy` instruction, and can be eliminated by uniting the LHS and RHS versions.[1]

Rename Elimination

Given an instruction of the following form

```
 %1.u = ϕ(.L₁:%1.v₁, .L₂:%1.v₂, …, .Lₙ:%1.vₙ);
```

where `%1` stands for any temporary root and $u, v_1, v_2, \ldots, v_n$ are versions:

- if there exists $v \neq u$ such that $\{v_1, v_2 \ldots, v_n\} \subseteq \{u, v\}$,
- then replace every occurrence of `%1.`u with `%1.`v in every instruction.

Lecture 8 contains a fully worked out example of both minimization steps.

2.4 *Checkpoint 1: Deliverables*

Your task is to write the program `ssagen.py` that:

1. Reads in a TAC file, computes its CFG, and then performs liveness analysis
2. Uses the live sets information to do crude SSA generation
3. Performs all the SSA minimization steps until no further minimizations can be done
4. Then linearizes the SSA CFG back to TAC, and then outputs it to a file with the extension `.ssa.tac`. That is, an input file `prog.tac` should be converted to `prog.ssa.tac`.

The original and the SSA version of the TAC should have identical behavior. The `tac.py` file you are provided has support for interpreting (executing) TAC with ϕ-functions. Run it to make sure that you generate the same output.

Note that in this checkpoint you are not expected to perform SSA-*destruction* (removal of ϕ-functions).

[1] Use a union-find data structure to implement this efficiently.

3 REGISTER ALLOCATION

In the second week of the lab you will be allocating temporaries to registers as much as possible. In order to do this, you will use a graph coloring algorithm that is described in this section.

Note: Procedure Calls

If your CFG contains procedure calls (i.e., the instructions `param` and `call`), then you are not requried to perform register allocation. In other words, all temporaries in such CFGs can be left on the stack like you already did in lab 4. We will cover register allocation in the presence of procedures in more detail later when discussing register coalescing, which will be a proposed project.

Color Mapping In the register allocation algorithms to follow, you will be trying to color the temporaries with a color set $K = \{1, 2, \ldots, k\}$ for some $k \in \mathbb{N}$. This color set is usable for x64 only if $k \leq 13$. When such a coloring has been obtained, it is then important to map color numbers to the registers in some canonical way. Unless you have a strong reason to prefer a different mapping, use this one:

```
color_map = ('%rax', '%rcx', '%rdx', '%rsi', '%rdi', '%r8', '%r9', '%r10',
             '%rbx', '%r12', '%r13', '%r14', '%r15', '%r11', '%rbp', '%rsp')
reg_map = {reg: color_map.index(reg) + 1 for reg in color_map}
def color_to_reg(col):   return color_map[col - 1]
def reg_to_color(reg):   return reg_map[reg]
```

It gives lower color values to caller-save registers, so the lower the register pressure in a function the less likely you are to need a callee-save register (which would need an additional `pushq`/`popq`).

Outline Register allocation proceeds in the following order.

0. Start from SSA (see sec. 2)
1. Compute the interference graph (sec. 3.1)
2. Use max-cardinality search to find a simplicial elmination ordering (SEO) (sec. 3.2)
3. Use greedy coloring based on the SEO (sec. 3.3)
4. If needing > 13 colors, spill some temporary and redo from step 2 (sec. 3.4)
5. Finally, compute the allocation record (sec. 3.5)

3.1 *Interference Graph*

From a CFG, the first step is to compute the *interference graph*, which is an undirected graph where:

- the nodes consist of TAC temporaries; and
- an edge between two nodes means that the nodes *cannot* be allocated to the same machine register.

The register allocation problem is equivalent to coloring the interference graph using as many colors as available machine registers (13).

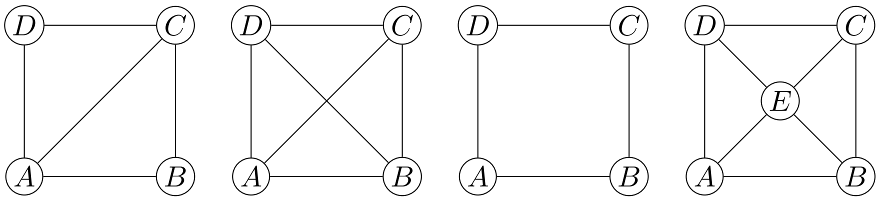
To create an interference graph, you first need to compute information for every instruction. As you saw in lecture, these sets are defined in terms of the sets: use( ), def( ), livein( ), and liveout( ). Based on these definitions, we can say that an edge in the interference graph is created if any of the following rules apply for any instruction I in the control flow graph.

- If I is a `copy` or a ϕ instruction, then every temporary $x \in \text{liveout}(I)$ interferes with every temporary $y \in \text{use}(I) \cup \text{def}(I)$.

- If I is any other instruction, then every temporary $x \in \mathsf{liveout}(I)$ interferes with every temporary in $y \in \mathsf{def}(I)$.

Note that in these rules, we implicitly assume that no self-loop is created in the interference graph.

It turns out that the SSA guarantees that the inteference graph is *chordal*. That is, every cycle with 4 or more nodes in the interference graph has a *chord*, which is an edge that links nodes that are not immediate neighbors in the cycle. The following are four graphs of which the first two are chordal but the second two are not because the cycle $A - B - C - D$ does not have a chord.



3.2 *Simplicial Elimination Orderings (SEO) and Max-Cardinality Search*

The graph coloring algorithm in the next section is based on computing an ordering of the nodes of the interference graph. As explained in lecture, we are looking for a *simplicial elimination ordering*. A node in the interference graph is *simplicial* if the nodes it is connected to are all connected to each other by single edges.[2] An ordering of nodes $v_1, v_2, \ldots, v_n$ is said to be a *simplicial elimination ordering* (SEO) if each v_i is simplicial in the subgraph of the interference graph formed by the nodes $v_1, \ldots, v_i$.

To find a SEO, we will use the so called *max-cardinality search* algorithm. In this algorithm each node v of the graph is assigned a *weight* $\mathsf{wt}(v)$, which is initialized to 0 and updated during the algorithm. (Intuitively, $\mathsf{wt}(v)$ stands for the number of neighbors of v that have been selected for the final ordering before v itself.) The algorithm is given in figure 3. The proof that it produces a SEO for chordal graphs is non-trivial.

3.3 *Greedy Coloring*

Using an elimination ordering to color the interference graph is a simple matter. The algorithm is shown in fig. 3. If the ordering is a SEO, then the greedy algorithm is guaranteed to find the minimal number of colors required to color the chordal interference graph. Obviously this property is not guaranteed to hold for other kinds of interference graphs and elimination orderings. However, the algorithm always returns a valid coloring.

The greedy algorithm takes a *partial coloring* as input, which is a color assigned to some of the temporaries. These color assignments come from the calling conventions. In particular, the temporaries that contain the input arguments of the procedure are preassigned the colors corresponding to the input registers. Likewise, the temporar(y/ies) containing the return value of the procedure should be precolored to the color mapped to `RAX`.

3.4 *Spilling Temporaries*

If the greedy coloring step returns a minimal coloring $\mathsf{col} : V \to K$ with $|K| > 13$, then the interference graph cannot be colored without *spilling* some temporaries to the stack. Specifically, $|K| - 13$ temporaries need to be spilled. To spill a register, perform the following steps:

[2]That is, the successors of the node form a clique.

Algorithm: Max Cardinality Search

Input: $G = \langle V, E\rangle$, an interference graph with $|V| = n$.

Output: a simplicial elimination ordering $[v_1, v_2, \ldots, v_n]$ where $V = \{v_1, v_2, \ldots, v_n\}$.

```
1. Create a mapping $\mathsf{wt}() : V \to \mathbb{N}$ and initialize $\mathsf{wt}(v) \leftarrow 0$ for all $v \in V$.
2. Let $W \leftarrow V$ (i.e., initially a copy of the vertex set $V$).
3. For $i$ from 1 to $n$:
    • Set $v_i \leftarrow \underset{w \in W}{\text{argmax}}\ \mathsf{wt}(w)$.        (i.e., the $i$th element of the SEO.)
      (If more than one node can be picked for $v_i$, pick one arbitrarily.)
    • For all $w \in W \cap \mathsf{next}(v_i)$: set $\mathsf{wt}(w) \leftarrow \mathsf{wt}(w) + 1$
    • $W \leftarrow W \setminus \{v_i\}$
```

(a) max-cardinality search

Algorithm: Greedy Coloring

Input: $G = \langle V, E\rangle$ and an elimination ordering $[v_1, \ldots, v_n]$ of V.
Input: A partial coloring $\mathsf{col} : V \rightharpoonup K$ where $K = \{1, 2, \ldots, k\}$ for some $k \in \mathbb{N}$.
Output: A complete coloring $\mathsf{col} : V \to K$

```
1. For every $u \in V$, if $u \notin \mathsf{dom}(\mathsf{col})$, then initialize $\mathsf{col}(u) \leftarrow 0$.
2. For $i$ from 1 to $n$:
    • If $\mathsf{col}(v_i) \neq 0$, skip to the next iteration of this loop.
    • Let $c \in K$ be the smallest color that is not used as a neighbor of $v_i$ by col; i.e.,
          $c = \underset{c \in K}{\text{argmin}}$ for every $w$ with $(v_i, w) \in E$: $\mathsf{col}(w) \neq c$.
      Note that $c \neq 0$ because $0 \notin K$.
    • Set $\mathsf{col}(v_i) \leftarrow c$.
```

(b) coloring based on elimination orders

Figure 3: Graph algorithms

1. Pick a temporary and assign it a stack slot. This temporary will not be allocated to a register. Note that you cannot pick one of the precolored temporaries (input arguments, output value) to spill, because they are required to have a color.
2. Remove that temporary from the interference graph. This will remove that node and all edges that have the node as an endpoint from the graph.
3. Recompute the SEO with max-cardinality search and retry coloring.

This process will have to be repeated until you obtain a coloring with ≤ 13 colors.

Selecting good temporaries to spill can be tricky. A good rule of thumb is to spill those temporaries that stay live for the shortest ranges. However, any spillable temporary can be chosen.

3.5 *Allocation Record*

After you have successfully computed the coloring for the registers, you will have to compute an *allocation record* for this CFG. The allocation record contains the following fields:

- How much stack space to allocate for spilled temporaries
- Which registers are mapped to which temporaries

More specifically, the activation record is an ordinary Python dictionary with the follwing fields:

- `'stacksize'`: an integer, representing the number of bytes that will be taken up by temporaries on the stack in the current (callee's) stack frame.
- `'alloc'`: a dictionary that maps each temporary to one of the following:
  - one of the x64 GPRs, written i.e., one of the following strings:

    ```
    {'%rax', '%rcx', '%rdx', '%rsi', '%rdi', '%r8', '%r9', '%r10',
     '%rbx', '%r12', '%r13', '%r14', '%r15', '%r11', '%rbp', '%rsp'}
    ```

  - an `RBP` offset, which is an integer. Negative values indicate temporaries, while positive values indicate stack arguments.

This dictionary should be placed as a comment that appears right before the TAC procedure. To convert a Python dictionary to a string, you can use the built-in `str()` function. An example of the allocation record is given in figure 4.

3.6 *Deliverables*

The final deliverable is a program called `regalloc.py` that performs register allocation on a (non-SSA) TAC input file. The output should also be a TAC file with an allocation record as a comment before every procedure. You do not have to generate x64 assembly in this lab.

You should use `ssagen.py` that you wrote in checkpoint 1 to create the CFG in SSA, and then perform the algorithms outlined in this section to allocate the (versioned) temporaries to registers. Once done, convert the SSA CFG back to TAC (leaving the ϕ-functions in place), and print the allocation record in a comment before each procedure.

For an input file `prog.tac`, produce the output `prog.alloc.tac`. If you wish, you may also output the SSA form before register allocation, `prog.ssa.tac`, like you did in checkpoint 1.

```
proc @fib(%n):
  %0 = const 0;
  %1 = const 1;
  %2 = const 1;
.L1:
  jz %n, .L3;
.L2:
  %n = sub %n, %2;
  %3 = add %0, %1;
  %0 = copy %1;
  %1 = copy %3;
  jmp .L1;
.L3:
  ret %0;
```

$\longrightarrow$

```
// {'stacksize': 8, 'alloc': {'%n': '%rdi',
    '%0.1': '%rax', '%1.2': '%rcx', '%2.3':
    '%rsi', '%n.4': '%rdi', '%0.5': '%rax',
    '%1.6': '%rcx', '%n.7': '%rdi', '%3.8': -8,
    '%0.9': '%rax', '%1.10': '%rcx'}}
proc @fib(%n):
.L0:
  %0.1 = const 0;
  %1.2 = const 1;
  %2.3 = const 1;
  jmp .L1;
.L1:
  %n.4 = φ(.L1:%n, .L2:%n.6);
  %0.5 = φ(.L1:%0.1, .L2:%0.9);
  %1.6 = φ(.L1:%1.2, .L2:%1.10);
  jz %n.4, .L3;
  jmp .L2;
.L2:
  %n.7 = sub %n.4, %2.3;
  %3.8 = add %0.5, %1.6;
  %0.9 = copy %1.6;
  %1.10 = copy %3.8;
  jmp .L1;
.L3:
  ret %0.5;
```

Figure 4: An example of allocation