CSE 302: Compilers | Lab 5

Dataflow Optimizations

Out: `2022-11-17`
Due: **`2022-11-27 23:59:59`**

---

1 INTRODUCTION

In this lab you will implement the following dataflow optimizations:

- Global Copy Propagation (GCP)
- Global Dead Store Elimination (DSE)

You will work with input TAC files, which you will convert into CFG in (crude) SSA form using the starter code that is provided to you. Your target will be TAC in SSA form, which you will be able to interpret using the provided TAC interpreter.

This lab will be assessed. It is worth 10% of your final grade.

You may work alone, or you may work in groups of size **2**. In the latter case, your submission *must* contain a file called `GROUP.txt` that contains the names of the group members.

> **Note**
>
> This lab shares no code with lab 4 or earlier labs. You should build upon the provided starter code.

2 STARTER CODE: TAC, CFG, AND CRUDE SSA

The compiler pass you implement in this lab will slot in after the generation of TAC and before the invocation of the `tac2x64.py` pass you wrote in lab 4. The source language is therefore the same TAC that you already used in lab 4.

2.1 TAC *Library: Parser and Interpreter*

You are provided with an implementation of a TAC parser and interpreter in the file `tac.py`. This class represents TAC instructions, procedure declarations, and global variable declarations using the classes `Instr`, `Proc`, and `Gvar` respectively. These classes are fairly self-documenting: read their `__init__()` member functions to see their relevant data attributes.

PARSER This library can also load TAC in either textual or JSON form using the function `load_tac()`, which takes a single TAC file name as input. The function will either raise a `ValueError` exception or return a list of `Proc` and `Gvar` instances. The library performs some minimal sanity checks on the provided TAC, but does not make sure that all the labels and symbols are well formed.

INTERPRETER To execute a TAC program, use the `execute()` function. Note that this function requires two dictionaries, `gvars` and `procs`, that map global symbols to global variables and procedures respectively. You will have to construct these dictionaries yourself from the output of `load_tac()`; for example, you can do the following:

```
gvars, procs = dict(), dict()
for decl in tac.load_tac('example.tac.json'):
  if isinstance(decl, Gvar): gvars[decl.name] = decl
  else: procs[decl.name] = decl
tac.execute(gvars, procs, '@main', [])
```

2.2 CFG *library*

The CFG library is in `cfg.py`. Most of the `CFG` class therein is self-documenting. To create a CFG from a TAC procedure, use the `infer()` function; dually, to linearize a CFG back into the body of a TAC procedure, use the `linearize()` function.

LIVENESS The live-in and live-out sets can be computed using the function `recompute_liveness()`. It takes two dictionaries, `livein` and `liveout`, as parameters; both of these map instructions (i.e., `tac.Instr` objects) to sets of temporaries. Both these dictionaries are emptied out and their contents are *replaced* with the corresponding live sets.

DISPLAY To help with debugging the CFG, you can use the `write_dotfiles()` function to generate a DOT file corresponding to the CFG. This is then processed with the Graphviz[1] toolkit to produce a PDF. There is an example of its use in the driver code at the bottom of `cfg.py`.

2.3 *Crude* SSA *Generation*

The SSA generation is done in the file `ssagen.py`, specifically with the function `crude_ssagen()`. It takes a `tac.Proc` instance and its `cfg.CFG` instance (produced with `cfg.infer()`) as arguments, and updates the `cfg.CFG` instance with its SSA form (crude).

The ϕ-functions of SSA are represented in an extended TAC language with a new opcode, `"phi"`, whose first argument is a dictionary mapping block entry labels to (versioned) temporaries. The meaning of this representation is explained in lecture 8.

Note

`crude_ssagen()` assumes that its input CFG is not in SSA form, i.e., it has no `phi` occurrences.

[Continued…]

[1] Graphviz: `https://graphviz.org`. It is already installed on the lab computers.

3 DATAFLOW OPTIMIZATIONS

You will implement two optimizations in this lab that make use of liveness and SSA form.

3.1 *Global Dead Store Elimination (DSE)*

For every instruction without any side effects, if it writes to a temporary that is not in its own live-out set, then the instruction is said to have a *dead store*. All such instructions may be safely deleted from the CFG without any alterations to the behavior of the code.

Note: Effectful Instructions

The following instructions may have runtime effects (IO or arithmetic exceptions) and should therefore never be deleted: `div`, `mod`, `call`.

Note: Multiple Iterations

After every round of DSE, you should recompute liveness information and check again if there are any new instructions with dead stores. If so, you should run another round of DSE.

3.2 *Copy Propagation (GCP)*

In the CFG in SSA form, for every copy instruction of the form `%u = copy %v;`, you can globally replace every occurrence of `%u` by `%v` and delete the `copy` instruction. As a result your code should be free of *all* `copy` instructions. GCP is a "one shot" procedure; you do not need to rerun it.

3.3 *Deliverables*

You will write the program `tac_dfopt.py` that will read a TAC file specified in the command line and output the optimized TAC (with SSA `phi` instructions) to standard output, or to a file specified using the `-o` option.

```
$  python3 tac_dfopt.py prog.tac.json
-- prints the optimized TAC(JSON) to standard output --

$  python3 tac_dfopt.py -o prog.dfopt.tac.json prog.tac.json
-- saves the optimized TAC(JSON) to prog.dfopt.tac.json --
```

Regression Testing

You can run any `.tac.json` file using the provided `tac.py` library. Use it to make sure that `prog.tac.json` and `prog.dfopt.tac.json` have the same behavior.