CSE 302: Compilers | Project 3

Dataflow Optimizations

Group Size:	3
Starts:	2020-11-23
Due:	**2020-12-17 23:59:59**

1 SYNOPSIS

In this project you will update the compiler backend you built in lab 5 to support a number of dataflow optimizations in SSA form. Your starting point will be the BX2 frontend (`bx2tac.py`) that you wrote during lab 4. Your output will also be TAC in SSA form which is suitable for input to a register allocator (such as the one you wrote in lab 5, week 2). The following will be specifically expected of you.

- (Stretch Goal) Implement constant folding at the typed AST level
- Produce TAC and compute a compact SSA form (effectively combining lab 4 and lab 5, week 1)
- Implement the following:
  - Global Copy Propagation (GCP)
  - Dead Store Elimination (DSE)
  - Sparse Conditional Constant Propagation (SCCP)
- Compute the dominator tree of the CFG
- Implement *one* of the following:
  - Common Subexpression Elimination (CSE)
  - Global Value Numbering (GVN)
- (Stretch Goal) Implement *both* CSE and GVN

Warning

Stretch goals are worth extra credit, but should be attempted only after completing all other goals.

2 CONSTANT FOLDING IN THE TYPED AST (STRETCH GOAL)

One optimization that can be performed directly on the (typed) AST is *constant folding*, where any subexpression that is formed with constants alone is precomputed at compile time. Take this code:

```
x = 42 * 100;
y = x + 84;
z = (2 + 3) * y;
```

Constant folding would identify that the expression used to set the value of `x` is the constant `4200`. Likewise, it would determine that the subexpression `2 + 3` in the definition of `z` is constant and replace it by `5`. Thus we obtain the following simplified code:

```
x = 4200;
y = x + 84;
z = 5 * y;
```

Note that constant folding will not replace `y` with `4284` because that requires *propagating* the fact that `x` is a constant in the expression `x + 84`. This further optimization step will be taken in the *constant propagation* optimization pass below.

You should exploit the associativity and commutativity of certain operators to simplify the constant subexpresions. For example, suppose you have the following:

```
x = 100 + (20 + (x + 30));
```

You may use the fact that addition is associative and commutative to rewrite the expression to:

```
x = (100 + 20 + 30) + x;
```

so that constant folding can subsequently compute the value of the constant subexpression. Consider implementing the following algebraic simplifications:

- Use commutativity or symmetry to bring constants to the left hand sides of binary operator applications, whenever possible. In particular, you can rewrite $x - y$ as $x + (-y)$ to make enable commutativity. (Likewise for division and modulus.)
- Use associativity to combine the constant portions of the expression tree.
- Use distributivity judiciously. For example, if k_1 and k_2 are constants, then you can simplify $k_1 \cdot (k_2 + x)$ to $k_1 \cdot k_2 + k_1 \cdot x$ and then fold $k_1 \cdot k_2$.

3 STATIC SINGLE ASSIGNMENT (SSA) FORM AND SIMPLE OPTIMIZATIONS

Like in lab 5, the first step is to compute the SSA form of the given input TAC, which entails crude SSA generation and minimization. However, as explained in lecture 10, one change we are going to make to the TAC is to allow for immediates (i.e., `NUM64`) to be used wherever a ⟨var⟩ would have occured in the argument of a TAC instruction. This makes `const` redundant with `copy`, so we will just use `copy` for both.

You need to be able to perform liveness analysis on SSA form. You can start with the `tac.zip` provided to you in lab 5 and update it to the generalized TAC that supports immediates as arguments.

3.1 *Global Copy Propagation (GCP)*

In SSA form, you can remove every instruction of the form `ta = copy tb;`, where `ta` and `tb` are temporaries, by globally replacing all uses of `ta` by `tb`. If you followed the SSA generation algorithm presented in the lectures, the SSA form you build will have the property that all uses of a temporary are dominated by their definitions.[1] Therefore, you don't have to worry about such replacements introducing potential use-before-define errors at runtime.

3.2 *Dead Store Elimination (DSE)*

Recall that an instruction that writes to a temporary that is not live-out in that instruction is said to be a *dead-store*. You have already implementation dead store elimination (DSE) for the earlier simple form of TAC. You should now extend it to handle TAC with immediates for arguments.

[1]This is sometimes called *strict* SSA.

4 SPARSE CONDITIONAL CONSTANT PROPAGATION (SCCP)

One of the benefits of SSA form is that it makes it easy to run sophisticated *abstract interpretation* operations your programs to find much more precise information about what temporaries are used and what values are written to them. This added precision gives the compiler many more opportunities for optimization. An example of such an optimization is *sparse conditional constant propagation* (SCCP) that extends the basic constant propagation of section 3, but which can handle many more possibilities. You will see SCCP outlined in the lectures in December.

The basic idea of this optimization is to determine which temporaries are *definitely* constant and which instructions are *definitely* reachable (at runtime) from the entry node. To do this, the algorithm starts by assuming that all temporaries are never used, and all instructions are never executed. Then, it traverses the CFG to find evidence for temporaries that are written to and instructions that are executed and updates the assumptions as needed. These updated assumptions could then make more parts of the CFG reachable so the process is iterated. The iterations continue until all the assumptions become stable, meaning further iterations do not change the assumptions. Thus, the algorithm finds a *minimal* set of constant temporaries and executed instructions that are dynamically reachable;[2] as a side-effect it also finds those temporaries that are never (dynamically) defined at all. The compiler can then get rid of the unnecessary temporaries and unreachable instructions. Thus, SCCP performs DCE, DSE, and propagation all at once.

Let V be the set of temporaries and B the set of basic blocks in the `CFG`. The SCCP algorithm maintains two mappings:

- Val : $V \to \{\top, \bot\} \cup \mathbb{Z}$ that maps temporaries to integer or boolean constants or the two distinguished constants $\bot$ (meaning "unused") and $\top$ (meaning "may be non-constant").
- Ev : $B \to \mathbb{B}$ that maps every block to a boolean where `false` means the block will definitely never be executed and `true` means that the instruction may be executed. We call the latter blocks *executable*.

Initially Val$(v) = \bot$ for all temporaries v, and Ev$(B) =$ `false` for all blocks B. Then:

- For every temporary that is an input to the procedure, we set Val$(v) = \top$ to denote out lack of knowledge about the constant-ness of the temporary.
- For the initial block B_o, we set Ev$(B_o) =$ `true`.
- For every block B, if Ev$(B) =$ `true` then go through B's jumps top-to-bottom:
  - If $I \in B$ is a conditional jump to B' and use$(I) = \{x\}$ with Val$(x) = \top$, then Ev$(B') =$ `true`.
  - If $I \in B$ is a conditional jump to B' and use$(I) = \{x\}$ with Val$(x) = \bot$, then stop further updates based on this and later jumps in B
  - If $I \in B$ is a conditional jump to B' and use$(I) = \{x\}$ with Val$(x) \notin \{\top, \bot\}$, then Ev$B' =$ `true` depending on Val(x). Call this a *definite jump*.
  - If $I \in B$ is an unconditional jump to B', then Ev$B' =$ `true` *unless* one of the earlier conditional jumps was a definite jump.
- In any instruction in an *executable* block that *uses* temporaries $t_1, t_2, \ldots$ with Val$(t_i) \notin \{\top, \bot\}$, set Val$(t_d)$ for each *def* temporary t_d according to the instruction
  - E.g., for `%3 = add %1, %2;`
    Val(`%1`) $= c \notin \{\top, \bot\}$, and

[2] In mathematical jargon, it computes a *least fixed point* of an abstraction process.

Val(`%2`) $= d \notin \{\top, \bot\}$, then
Val(`%3`) $= c + d$.

- If any of the used temporaries (per prev. case) is mapped to $\top$, so is every def temporary.
  - E.g., for $\boxed{\texttt{\%3 = add \%1, \%2;}}$,
    if Val(`%1`) $= \top$, then Val(`%3`) $= \top$ regardless of Val(`%2`).
  - This is clearly an approximation, since `%1 + %2` may actually be constant even if neither `%1` nor `%2` is constant.
- For an executable $\boxed{\texttt{\%d = } \phi\texttt{(L}_1\texttt{:\%1, L}_2\texttt{:\%2, \ldots, L}_k\texttt{:\%}k\texttt{);}}$
  - If Val(`%`i) $= c \neq d =$ Val(`%`j) with $c, d \notin \{\top, \bot\}$, then Val(`%d`) $= \top$.
    In other words, a ϕ-choice between two different constants always gives $\top$.
  - If Val(`%`i) $= \top$ and the block labeled `L`$_i$ is executable then Val(`%d`) $= \top$.
  - If Val(`%`i) $= c \notin \{\top, \bot\}$ and for all $j \neq i$:
    - Val(`%`j) $= \bot$, or
    - the block labeled `L$_j$` is not executable, or
    - Val(`%`j) $= c$

    then Val(`%d`) $= c$.

The above steps are repeated until both Val and Ev stop changing. Afterwards, all instructions that use or define temporaries u for which Val$(u) = \bot$ can be deleted, as can all blocks B for which Ev$(B) =$ `false`. Moreover, whenever we have Val$(u) = c \notin \{\top, \bot\}$, we can replace u with c and delete the instruction that sets u.

5 COMPUTE THE DOMINATOR TREE

More complex optimizations need to answer the following question: *for any given instruction, which is the last instruction that is **guaranteed** to have run before it?* It turns out that this is what is known as a strict dominator node in the CFG.

In a directed graph with a designated entry node e, a node u is said to *dominate* a node v if every path from e to v passes through u. Domination is a transitive property: if u dominates v and v dominates w, then u dominates w. A *strict dominator* of v is a dominator $u \neq v$.

To compute dominators, we will build a mapping Dom : $V \to \mathcal{P}(V)$ where V is the set of nodes of the CFG (with $e \in V$ being the designated entry node). The interpretation is that every node in Dom(v) dominates v. This function satisfies the following recurrence:

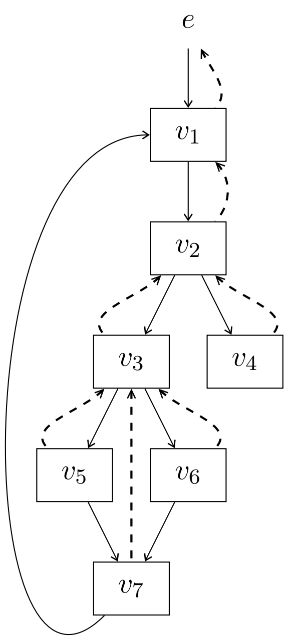
$$\begin{aligned} \text{Dom}(e) &= \{e\} \\ \text{Dom}(v) &= \{v\} \cup \left(\bigcap_{u \in \text{prev}(v)} \text{Dom}(u) \right) && \text{if } v \neq e \end{aligned}$$

This yields a natural quadratic algorithm shown in figure 1.

Dominators can be indicated by means of adding *dominator edges* to the CFG. In fact, from Dom we can extract a *dominator tree* where the parent links are to strict dominators (recall: a strict dominator u of v is when $u \in$ Dom(v) and $(u, v) \in E$), so it is traditional to just retain the dominator tree edges in the CFG. As an example, here is a CFG with ordinary edges (jumps and branches) drawn a solid arrows, and strict dominator edges drawn as dashed arrows.

```
Algorithm: Compute Dominators
Input: a control flow graph $G = \langle V, E \rangle$.
Set $\mathrm{Dom}'(e) = \{e\}$
For every $v \in V \setminus \{e\}$, set $\mathrm{Dom}'(v) = V$
Do:
    Set $\mathrm{Dom} = \mathrm{Dom}'$
    For each $v \in V \setminus \{e\}$:
        Set $\mathrm{Dom}'(v) = \{v\} \cup \left(\bigcap_{u \in \mathrm{prev}(v)} \mathrm{Dom}(u)\right)$
while $\mathrm{Dom} \neq \mathrm{Dom}'$.
Return Dom.
```

Figure 1: Computing dominators



Dominators can be found much faster—in nearly linear time—with the algorithm of Lengauer and Tarjan [2]. If you want to implement this, follow Appel's presentation [1, pp. 410–417].

6 COMMON SUBEXPRESSION ELIMINATION (CSE) (ALTERNATIVE)

The idea of this optimization is that when you have code that looks like this:

```
%3 = add %1, %2;
%4 = neg %3;
%5 = add %1, %2;
```

then you can replace the "*common subexpression*" that computes the sum of `%1` and `%2` with the previously assigned value, `%3`. That is, the above code is replaced with:

```
%3 = add %1, %2;
%4 = neg %3;
%5 = copy %3;
```

Together with copy propagation (see the previous section), this would eliminate both the redundant computation and the redundant copy.

One way to implement this is to maintain a vector of *available expressions* as you traverse the code from top to bottom. In the above code, at the end of line 2, there are two available expressions—`(%1 + %2)` computed in line 1 and `(- %3)` computed in (line 2)—whose values are stored in the temporaries `%3` and `%4` respectively. When one of these expressions is encountered in line 3, instead of recomputing it, the temporary holding the value is simply copied.

Extending the intuition of the above algorithm to the case of arbitrary control flow graphs requires an appeal to the dominator links. The available expressions book-keeping now also needs to remember the label of the block that generated that expression. Then, suppose the expression made available in block `.L1` is encountered again in `.L3`. Now we have two options.

- `.L1` dominates `.L3`, i.e., every code path to `.L3` goes through `.L1`. In this case, we are sure that the expression is computed before we reach `.L3`, so we can replace it with a `copy` as shown above.
- `.L1` does not dominate `.L3`. In this case, there is a path to `.L3` that does not go through `.L1`, so we are not sure that the expression has definitely been computed before. So we have no choice but to compute the expression (possibly redundantly) at `.L3`, and now *this* computation at `.L3` becomes available for subsequent instructions.

To implement available expressions, you will need a table mapping expressions to labeled TAC instructions. Don't spend a lot of time trying to build a hashtable structure for expressions. This is a hard problem in general and not needed for the scale of the compiler you're writing. A simple unsorted vector of available expression strings using structural comparisons to test for membership will suffice.

7 GLOBAL VALUE NUMBERING (GVN) (ALTERNATIVE)

UNDER EDIT

This section is currently being fixed. Please check back later this week.

8 DELIVERABLES

Submit a program `bx2tac_opt.py` that takes a BX2 program, say `prog.bx`, and generates optimized TAC in SSA form, `prog.tac`. Also document all of your design choices in a `README.txt` (or `README.md`).

REFERENCES

[1] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, second edition, 2004.

[2] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):115–120, July 1979.