Final Exam

COSE312 Compilers, Fall 2017

Instructor: Hakjoo Oh

Problem 1. (10pts) Consider the expression language:

$$e \to n \mid x \mid e_1 + e_2$$

Define its semantics in big-step operational style.

$$\frac{}{\langle n, s\rangle \Rightarrow n}$$

$$\frac{}{\langle x, s\rangle \Rightarrow s(x)}$$

$$\frac{\langle e_1, s\rangle \Rightarrow n_1 \qquad \langle e_2, s\rangle \Rightarrow n_2}{\langle e_1 + e_2, s\rangle \Rightarrow n} \; n = n_1 + n_2$$

where $s \in$ denotes a memory state ($s \in \mathbf{State} = \mathit{Var} \to \mathbb{Z}$).

Problem 2. (15pts) Recall the denotational semantics of **While**:

$$\begin{array}{rcl}
\mathcal{C}[\![\, a \,]\!] & : & \mathbf{State} \hookrightarrow \mathbf{State} \\
\mathcal{C}[\![\, x := a \,]\!] & = & \lambda s.s[x \mapsto \mathcal{A}[\![\, a \,]\!](s)] \\
\mathcal{C}[\![\, \texttt{skip} \,]\!] & = & \mathsf{id} \\
\mathcal{C}[\![\, c_1; c_2 \,]\!] & = & \mathcal{C}[\![\, c_2 \,]\!] \circ \mathcal{C}[\![\, c_1 \,]\!] \\
\mathcal{C}[\![\, \texttt{if}\ b\ c_1\ c_2 \,]\!] & = & \mathsf{cond}(\mathcal{B}[\![\, b \,]\!], \mathcal{C}[\![\, c_1 \,]\!], \mathcal{C}[\![\, c_2 \,]\!]) \\
\mathcal{C}[\![\, \texttt{while}\ b\ c \,]\!] & = & \mathit{fix} F
\end{array}$$

where

$$\begin{array}{rcl}
F(g) & = & \mathsf{cond}(\mathcal{B}[\![\, b \,]\!], g \circ \mathcal{C}[\![\, c \,]\!], \mathsf{id}) \\
\mathsf{cond}(f, g, h) & = & \lambda s. \begin{cases} g(s) & \text{if } f(s) = \mathit{true} \\ h(s) & \text{if } f(s) = \mathit{false} \end{cases}
\end{array}$$

1. (8pts) Determine the function F for the while loop below:

```
while ¬(x = 0) x:=x-1
```

$$F(g)(s) = \begin{cases} g(s[\mathtt{x} \mapsto s(\mathtt{x}) - 1]) & \text{if } s(\mathtt{x}) \neq 0 \\ s & \text{if } s(\mathtt{x}) = 0 \end{cases}$$

2. (7pts) Determine $\mathit{fix} F$ for the loop above.

$$g(s) = \begin{cases} s[\mathtt{x} \mapsto 0] & \text{if } s(\mathtt{x}) \geq 0 \\ \mathsf{undef} & \text{if } s(\mathtt{x}) < 0 \end{cases}$$

Problem 3. (15pts) Consider the expression language:

$$E \to n \mid -E \mid E + E$$

with the semantics:

$$[\![\, n \,]\!] = n,\ [\![\, -E \,]\!] = -[\![\, E \,]\!],\ [\![\, E_1 + E_2 \,]\!] = [\![\, E_1 \,]\!] + [\![\, E_2 \,]\!]$$

Let us translate the language into a stack machine M. The machine instruction is defined as follows:

$$C \to \epsilon \mid \texttt{push}\ n.C \ \ (n \in \mathbb{Z}) \mid \texttt{add}.C \mid \texttt{rev}.C$$

A machine configuration consists of an instruction and a stack:

$$\langle c, s\rangle \in C \times \mathbb{Z}^*$$

For instance, the instruction `push 1.push 2.add.rev` is executed as follows:

Stack	Instruction
ϵ	`push 1.push 2.add.rev`
1	`push 2.add.rev`
$2 :: 1$	`add.rev`
3	`rev`
-3	

1. (8pts) The semantics of M is defined by small-step operational semantics. Define the transition relation:

$$(\triangleright) \subseteq (C \times \mathbb{Z}^*) \times (C \times \mathbb{Z}^*).$$

$$\begin{array}{rcll}
\langle \texttt{push}\ n.C, s\rangle & \triangleright & \langle C, n :: S\rangle & \\
\langle \texttt{pop}.C, n :: s\rangle & \triangleright & \langle C, s\rangle & \\
\langle \texttt{add}.C, n_1 :: n_2 :: S\rangle & \triangleright & \langle C, n :: S\rangle & (n = n_1 + n_2) \\
\langle \texttt{rev}.C, n :: S\rangle & \triangleright & \langle C, -n :: S\rangle &
\end{array}$$

2. (7pts) Define the function trans that translates E to C:

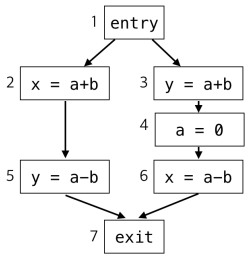
$$\mathsf{trans} : E \to C$$

Note that your translation must preserve the semantics:

$$\forall e \in E.\ \langle \mathsf{trans}(e), \epsilon\rangle \triangleright^* (\epsilon, [\![\, e \,]\!])$$

$$\begin{array}{rcl}
\mathsf{trans}(n) & = & \texttt{push}\ n \\
\mathsf{trans}(-E) & = & \mathsf{trans}(E).\texttt{rev} \\
\mathsf{trans}(E_1 + E_2) & = & \mathsf{trans}(E_1).\mathsf{trans}(E_2).\texttt{add}
\end{array}$$

Problem 4. (30pts) An expression e is said to be *very busy* at a program point p if on every path from p, the expression e is evaluated before any of the variables occurring in the expression are redefined. For example, consider the program:



Expression $a + b$ is very busy at block 1 but expression $a - b$ is not because a is redefined at block 4 before being used at block 6. This information can be used, for example, to hoist very busy expressions and to reduce code size (e.g. put $t = a + b$ at block 1 and replace $a + b$ in blocks 2 and 3 by t).

The goal of very busy analysis is to compute

$$\begin{array}{rcl} \mathsf{in} & : & \mathit{Block} \to 2^{\mathit{Expressions}} \\ \mathsf{out} & : & \mathit{Block} \to 2^{\mathit{Expressions}} \end{array}$$

which map blocks to very busy expressions. The gen and kill sets store the set of evaluated and killed expressions, respectively:

$$\begin{array}{rcl} \mathsf{gen}(B) & : & \text{the set of expressions evaluated at } B \\ \mathsf{kill}(B) & : & \text{the set of expressions whose variables are killed at } B \end{array}$$

For example, when B is the statement $x = a$, $\mathsf{gen}(B) = \{a\}$ and $\mathsf{kill}(B) = \{e \mid e \text{ contains } x\}$.

1. (10pts) Set up the data-flow equations for very busy analysis. Assume kill and gen are given.

   $\mathsf{out}(B) = \bigcap_{B \to S} \mathsf{in}(S), \quad \mathsf{in}(B) = (\mathsf{out}(B) - \mathsf{kill}(B)) \cup \mathsf{gen}(B)$

   (To be precise, $\mathsf{in}(\mathit{exit}) = \emptyset$ is needed, but it can be omitted as the following fixed point algorithm requires explicit initialization.)
2. (10pts) The equations can be solved by the fixed point algorithm:

   // Initialization
   For all i, $\mathsf{in}(B_i) = \mathsf{out}(B_i) =$ (1)
   $\mathsf{in}(\mathit{exit}) =$ (2)
   // Fixed point computation
   while (changes to any in and out occur) {
   Update in and out according to the equations
   }

   Complete (1) and (2). (1): the set of all expressions, (2): $\emptyset$
3. (5pts) Does the analysis compute the least fixed point or greatest fixed point? Explain. gfp
4. (5pts) Is the analysis forward or backward? Explain. backward

Problem 5. (30pts) True/False questions. (Do not answer when you are uncertain; A correct answer gets you 2 points but you lose 2 points for each wrong answer.)

1. Semantic analysis assumes that the input program is syntactically correct. O
2. Consider the set of arithmetic expressions:

   $$e \to n \mid x \mid e_1 + e_2$$

   The small-step operational semantics can be defined as follows:

   $$\frac{}{\langle n, s\rangle \Rightarrow n} \quad \frac{}{\langle x, s\rangle \Rightarrow s(x)} \quad \frac{\langle e_1, s\rangle \Rightarrow \langle e_1', s\rangle}{\langle e_1 + e_2, s\rangle \Rightarrow \langle e_1' + e_2, s\rangle}$$

   $$\frac{\langle e_1, s\rangle \Rightarrow n_1}{\langle e_1 + e_2, s\rangle \Rightarrow \langle n_1 + e_2, s\rangle} \quad \frac{\langle e_2, s\rangle \Rightarrow \langle e_2', s\rangle}{\langle n_1 + e_2, s\rangle \Rightarrow \langle n_1 + e_2', s\rangle}$$

   X
3. In big-step operational semantics, non-terminating programs do not have meaning. O
4. With big-step operational semantics, semantic equivalence of S_1 and S_2, denoted $S_1 \equiv S_2$, can be defined as follows:

   $$\forall s, s'.\ \langle S_1, s\rangle \to s' \iff \langle S_2, s\rangle \to s'$$

   O
5. The two programs P_1 and P_2 are semantically equivalent:

   $$\begin{array}{rcl} P_1 & \stackrel{def}{=} & \texttt{repeat}\ S\ \texttt{until}\ b \\ P_2 & \stackrel{def}{=} & S;\ \texttt{while}\ \neg b\ \texttt{do}\ S \end{array}$$

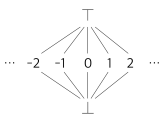
   O
6. The two programs P_1 and P_2 are semantically equivalent:

   $$\begin{array}{rcl} P_1 & \stackrel{def}{=} & \texttt{while true do skip} \\ P_2 & \stackrel{def}{=} & \texttt{x} := 1; \texttt{while x} > 0 \texttt{ do x} := \texttt{x} + 1 \end{array}$$

   O
7. The type of the least fixed point operator *fix* in Problem 2 is as follows:

   $$\mathit{fix} : (\mathbf{State} \hookrightarrow \mathbf{State}) \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

   X
8. When a program is in SSA (Static Single Assignment), every variable is defined only once at runtime. X
9. The reaching definition analysis we defined in class is a "may-analysis"; even when the analysis finds out that a definition d reaches a program point p, d may not reach p at runtime. O
10. The liveness analysis we defined in class is a "must-analysis"; when the analysis finds out that a variable x is live at a program point p, x is guaranteed to be live at p at runtime. X
11. The available expression analysis we defined in class is a "may-analysis"; even when the analysis finds out that an expression e is available at a point p, e may not be available at p at runtime. X
12. The constant propagation analysis we defined in class is a "must-analysis"; when the analysis finds out that a variable x has a constant value at a point p, x is guaranteed to be constant at p at runtime. X
13. The constant propagation domain

    is a complete lattice. O

14. In interval analysis, the conventional fixed point computation does not terminate for all programs. X
15. We can use constant propagation analysis to detect use of uninitialized variables. O