Exam : Compilation and Program Analysis (CAP)
3 hours, January, 14th

Instructions :

1. We give you a companion sheet (at the end of this pdf file)
2. Every single answer must be informally explained AND formally proved.
3. We give indicative timing : roughly one third per part (a bit less for part 2)
4. Vous avez le droit de rédiger en Français.
5. Manuscript only.
6. When scanning, be aware of the size of the generated pdf (<5 MB). You will have 10 (not more than 15) minutes to make the deposit on Tomuss ("exam_deposit", a **unique pdf file** please).

Problem 1 : SSA form to help static analyses

Adapted from "the SSA book" (F. Rastello et al.) and "Modern Compiler Implementation in C/ML/Java" (A. Appel)

Listing 1 – Program 1

```
1   i = 1;   j = 1;   k = 0;
2   while(k<100){
3     if (j<20) {
4       j = i;
5       k = k+1;
6     }
7     else {
8       j = k;
9       k = k+2;
10    }
11  }
12  return j;
```

Basic questions In this section we consider the program in Listing 1.

Question #1

Turn it into a CFG, using the exact same variables (this CFG will not be in SSA form).

Question #2

Compute the Domination Tree of this CFG. Briefly explain how you arrive to your result.

Question #3

Associate to each block of the CFG its dominance frontier.

Question #4

Give the SSA form of the CFG. Briefly explain how you exploit the domination tree and dominance frontiers to compute this representation.

SSA and dead code elimination Recall that dead code elimination is an optimization aiming at removing instructions with no side-effect whose assigned variable is dead. For this part we consider the CFG already in SSA form of Figure 1.

Question #5

Describe how dead code elimination is made easier over CFGs in SSA form. Give a naive, high level algorithm for dead code elimination over CFGs in SSA form.

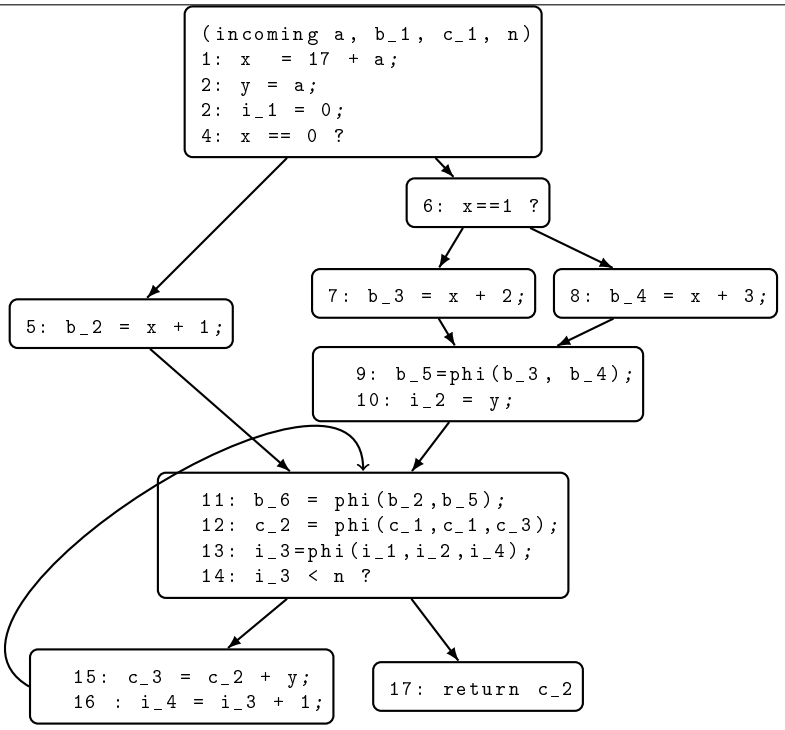

FIGURE 1 – A foo program already in SSA form

Eliminating an instruction can cause new instructions to become dead : simply iterating the elimination algorithm globally would be inefficient, being quadratic in the size of the program in the worst case.

Question #6

Devise an algorithm whose execution is linear in the size of the program.
Hint : it will more precisely execute in time proportional to the size of the program + the number of instructions eliminated.

Question #7

Run your dead code elimination algorithm on the example of Figure 1.

SSA and constant propagation We now consider another optimization : constant propagation. Remember that the purpose of a simple constant propagation optimization is the following : If an affectation $v \leftarrow c$ is such that c is (syntactically) a constant, then any use site for v can be replaced by c and this instruction can be removed.

Question #8

Can you formulate a similar simplification rule for phi nodes ?

Question #9

Can you explain in a sentence how the SSA form makes this optimization simpler to implement ?

Question #10

Can you give an algorithm to perform constant propagation that uses a work-list of instructions (i.e. a list of instructions that remain to be considered) as its main datastructure ? What is the time complexity of your algorithm ?

Question #11

Can you suggest at least two other simple optimizations that benefit similarly from the SSA form, and could hence be trivially performed simultaneously by incorporating them in the same work-list algorithm ? (no need to rewrite your algorithm, you can simply argue informally)

Question #12

Recall the program example from Question 4 that you turned into SSA form. Run your constant elimination algorithm on the SSA form you obtained.
Hint : it should have an effect.

Question #13

(**) Look back at the initial C-like program from Question 4. What can you say about the variable j ? Can you explain why your algorithm did not notice this fact ? Can you suggest an algorithm that would be able to tackle this case ?
Hint1 : you need to expand the notion of constant beyond syntactic constants.
Hint2 : as often in compilation, you are looking for a fixed-point.

Problem 2 : A static analysis for security

Adapted from Nielson and Nielson

We present here a simple forward static analysis for MiniC/MiniWhile based on abstract interpretation with a finite height "pointwise" lattice. The objective is to propagate information of "public" (`LOW`) versus "private"(`HIGH`) data or variable, at every control point of the program. The final analysis will ensure that :

— whenever public value is assigned to a variable, the variable is made public (`LOW`) from this point of the program

— whenever a variable that is currently private is used in an expression to be assigned to a variable, then the destination variable becomes classified as private (`HIGH`) after assignment.

— whenever a boolean expression in a `if` or `while` construct involves private data (`HIGH`), then all variables assigned in the body must be classified as private (`HIGH`) after the body.

Listing 2 – Factorial Program

```
1  y := 1 ;
2  while !(x==1) {
3    y := y * x;
4    x := x - 1
5  }
```

Listing 3 – Factorial Program variant

```
1  // no more initialisation of y
2  while !(x==1) {
3    y := y * x;
4    x := x - 1
5  }
```

Question #1

Make the asumption that x is classified as public (`LOW`) and y as private (`HIGH`) at the beginning of the program. What abstract value should be computed by the analysis for y at the end label of Listing 2) ?

Question #2

Same question for Listing3.

Let us denote by $P = \{\texttt{LOW}, \texttt{HIGH}\}$ the lattice depicted by the following Hesse Diagram of Figure 2. Values of P will be denoted by p. We note $p_1 \sqcup_P p_2$ the maximum of p_1 and p_2 in the lattice P.

Let us now try to design a static analysis where abstract values for each control points would be of type $PState_0 = Var \to P$. The ordering $\sqsubseteq_{PS0}$ on this abstract domain is defined pointwise, like in the course. We also use the convenient functional notation ps_0x to get the single abstract value (in P) of a given variable $x \in Var$. The analysis is then a classical fixpoint iteration with forward propagation of information from an initial state where all private variables are assigned to `HIGH` and public variables to `LOW`. As the underlying lattice is finite, if we properly define our transfer functions, the analysis will safely terminate.

FIGURE 2 – Hesse diagram defining $\sqsubseteq_P$

Listing 4 – Simple statement S

```
1 if (x==1) then x := 1 else x := 2
```

Question #3

Under the assumption $ps_0 x = \texttt{HIGH}$ at the beginning of the program, explain the steps of the analysis (consider any abstract evaluation for the test) and the final conclusion for x at the end of program depicted in 4. *Hint : does the final value of x give an information about the initial one ?*

To track additional information, we add a token `history`. The idea is that if `history` is mapped to `LOW`, we know for sure that we are not in a body of an if or while expression conditioned by a private expression ; if it is mapped to `HIGH`, then we may be in this situation (thus we need to be careful). Now we have the new abstract domain being : $PState_1 = (Var \cup \{\texttt{history}\}) \to P$.

Let us now define the transfer functions (abstract semantics evaluation). We denote by $\mathcal{SA}(exp, ps)$ the evaluation of the arithmetical expression exp considering the current abstract value ps. Similarly we define $\mathcal{SB}$ for boolean expressions.

Question #4

Define $\mathcal{SA}(n, ps)$, and $\mathcal{SA}(x, ps)$ and explain informally how it solves the problem of Question 3. *Tip : We recall that the values should depend on the current value* $ps_{\texttt{history}}$.

Question #5

Suppose that $ps_{\texttt{x}} = \texttt{HIGH}$, what should be the value of $\mathcal{SB}(x == 1, ps)$?

Question #6

Define $\mathcal{SA}(e_1 + e_2, ps)$.

We now have to define the abstract transfer functions for statements, as a function of the type.

$$Stm \to PState_1 \to Pstate_1$$

We call this function $\mathcal{SS}$ and give below the easy part of the definition :

— $\mathcal{SS}(\texttt{skip})ps = ps$

— $\mathcal{SS}(\texttt{S1; S2}) = \mathcal{SS}(\texttt{S2}) \circ \mathcal{SS}(\texttt{S1})$.

— $\mathcal{SS}(\texttt{x:=a})ps = ps[x \mapsto \mathcal{SA}(a, ps)]$

Question #7

Define $\mathcal{SS}$ for the conditional : `if b then S1 else S2`.

Question #8

Apply this algorithm on Listing 4.

The last two questions are a a bit harder.

Question #9

Define $\mathcal{SS}$ for the while using a fixpoint definition. Explain why it terminates.

Question #10

Apply your analysis on the second factorial example (Listing 3).

Problem 3 : a simple reactive language

Adapted from an homework given by F. Pereira, UFMG, Brasil

We consider the reactive language `MiniReac`. In a reactive language, some constructions, here called triggers, allow to react to events launched by other constructions of the language. In `MiniReac`, the only events considered are the modification of variable. The `MiniReac` grammar is given in Figure 3. V is a set of variables identifiers. The star (*) denotes a list. Some programs are depicted below, with one or several of their possible outputs.

$$
\begin{array}{lr}
P ::= D^*S^* & \text{(Program)} \\
D ::= \texttt{VAR}\ V = E & \text{(Declaration)} \\
S ::= V := E & \text{(Assignment)} \\
\quad \mid \texttt{ON}\ V\ (C)\ \{P\} & \text{(Trigger)} \\
\quad \mid \texttt{OUT}(E) & \text{(Outputs)}
\end{array}
\qquad
\begin{array}{lr}
E ::= c & \text{(Constants)} \\
\quad \mid V & \text{(Variables)} \\
\quad \mid V \oplus E \text{ with } \oplus \in \{+, -, \ldots\} & \text{(Operator)} \\
\quad \mid \texttt{IN}(T) \text{ with } T \in \{\texttt{int}, \texttt{float}, \texttt{bool}\} & \text{(Inputs)} \\
C ::= \texttt{true} \mid \texttt{false} & \text{(Boolean constants)} \\
\quad \mid V \oplus E \text{ with } \oplus \in \{==, !=, <, \ldots\} & \text{(Conditions)}
\end{array}
$$

FIGURE 3 – MiniReac grammar

Listing 5 – Prints the range 0..x if x is positive

```
VAR x = IN(int)
VAR b = 1
ON b (b < x) {OUT(b); b := b+1; }
# OUTPUT: 0 1 2 ... x
```

Listing 6 – Non deterministic Program

```
VAR x = 42
ON x (x > 0) { x := 1; OUT(0); }
ON x (x == 42) { x := 1; OUT(42); }
# POSSIBLE OUTPUT: 42 0 0 0 0 ...
# POSSIBLE OUTPUT: 0 0 0 0 ...
```

Listing 7 – Program with nested binds

```
VAR x = 1
VAR y = 42
VAR u = 12
ON x (u == 12) {
  ON y (true) { OUT(u); }
};
ON u (u == 12) { u := 13; }
# POSSIBLE OUTPUT: 12
# POSSIBLE OUTPUT: 13
# POSSIBLE OUTPUT:
```

Semantics Informal semantics for expressions (E or C).

— Numerical and boolean operators have the usual meaning.

— an $\texttt{IN}(T)$ expression gets a value of type $T \in \{\texttt{int}, \texttt{float}, \texttt{bool}\}$.

Informal semantics for programs (P) and statements (S) :

— Variables are declared first and globally via the construction `VAR` $X = E$.

— an `OUT`(E) statement evaluates the expression and prints out its value on the terminal.

— Assignments assigns the value to the variable, and indicate that the variable can launch a trigger again.

— `ON` $V(C)\{P\}$ sets up a trigger : every time the variable V is (newly) assigned, the condition C is evaluated, if it evaluates to true, then the variable is consumed and the body P is executed **sequentially in order**. All the triggers defined at the top level are set up from the start. When several triggers apply one of them is picked arbitrarily.

Please justify your answers. If you think that some specification is lacking, provide explicitely your asumptions on your paper.

Question #1

Informally describe the behavior of the following program :

Listing 8 – A mysterious program

```
1 VAR x = IN(bool)
2 ON b (x == true)  { b := b + 1; OUT(b);}
3 ON b (x == false) { b := b - 1; OUT(b);}
```

Question #2

Write a static semantics (typing rules) for this language. Use typing judgments of the form $\Gamma \vdash P$ for programs and $\Gamma \vdash E : T$ for expressions. Write only the rules for declarations and statements.

Question #3

We are now interested in the dynamic evaluation of `Minireac`.

In Program 6 and 7, explain why we can observe two behaviors. Does the evaluation environment need to contains additional information ?

We decide to use an environment Env as a map Var $\mapsto$ (Val, `B`) where `B` is a boolean indicating the freshness of the variable.

Question #4

Write the operational big step semantics for expressions $E_1 \oplus E_2$ and V. We use a judgement of the form Env, $E \rightarrow Val$ where Env is the runtime environment.

Question #5

In Program 7, explain why we can observe two behaviors.

We are now ready to build the complete semantic for `MiniReac`.

We model the triggers as a workset, noted W. A workset is simply an unordered set of triggers : $\{\texttt{ON}\ V_1(C_1)\{P_1\}; \ldots, \texttt{ON}\ V_n(C_n)\{P_n\}\}$. $\emptyset$ is the empty workset.

We consider two reductions :

— $\text{Env}, P \xrightarrow{s} \text{Env}', W$ reduces the program P in environment Env and return an updated environment Env' and a workset W. This reduction evaluates the program sequentially and gather all the triggers.

— $\text{Env}, W \xRightarrow{s} \text{Env}', W'$ executes the workset W in environment Env and updates them. This version evaluates the triggers arbitrarily.

Both reductions are also labelled by a list s containing the output produced by the `OUT` commands. We note `[]` the empty list and $+$ the concatenation on lists.

Question #6

Write the sequential evaluation on statements (4 rules)

Triggers are evaluated in a arbitrary order. The evaluation strategy must thus pick a trigger to try. The rule NonDet does such choice arbitrarily.

$$\textsc{NonDet}\ \frac{\text{Env}, \texttt{ON}\ V(C)\{P\} \xRightarrow{s} \text{Env}', W}{\text{Env}, \{W_1; \texttt{ON}\ V(C)\{P\}; W_2\} \xRightarrow{s} \text{Env}', \{W_1; W; W_2\}}$$

Question #7

Write the remaining reduction rules on worksets (3 rules)

Code generation We now choose to generate imperative code from `miniReac` programs.

To make this task easier, we only consider programs without nested triggers. From Program 8, the compilation scheme outputs the program of Figure 9.

Listing 9 – A miniC program after code generation of program 8

```
1  int main(){
2    bool x = random();
3    int b = 0;
4    bool present_b = true;
5
6    while (true) {
7      if (present_b) {
8        if (x==true) {
9          present_b = false;
10         b = b+1;
```

```
11          present_b = true;
12          print(b);
13        }
14      }
15      if (present_b) {
16        if (x==false) {
17          present_b = false;
18          b = b-1;
19          present_b = true;
20          print(b);
21        }
22      }
23    }
24  }
```

Question #8

Explain the choices made by the compiler : instruction scheduling and compilation of triggers.

Question #9

Give code generation rules for programs and statements. You can make recursive calls to auxiliary functions like `GenCodeExprMiniC(e)` or variable names generators provided you give at least an informal functionality of them.

Question #10

For programs without nested triggers, give a relation between the size of the workset computed by the dynamic semantic and the size of the generated program.

What happens in the presence of nested triggers ?

Question #11

We now consider the following C library.

```
typedef (void (*fun)(void)) trigger; // A trigger
typedef struct worksetStr* workset; // A workset of triggers
void push (trigger t, workset w); // Add a new trigger
void schedule (workset w); // Call triggers from the workset
```

Provide a general compilation scheme.

CAP Companion Sheet

Mini-while abstract syntax

Boolean expr:

$$\begin{array}{rcll} b \in \mathcal{B} & ::= & \texttt{true} \mid \texttt{false} & \textit{constant} \\ & \mid & b\,\texttt{or}\,b & \textit{or} \\ & \mid & b\,\texttt{and}\,b & \textit{and} \\ & \mid & (e == e) & \textit{tests} \\ & \mid & \cdots & \end{array}$$

Numerical expressions:

$$\begin{array}{rcll} e \in \mathcal{E} & ::= & n & \textit{constant} \\ & \mid & x & \textit{variable} \\ & \mid & e + e & \textit{addition} \\ & \mid & e \times e & \textit{multiplication} \\ & \mid & \ldots & \end{array}$$

Statements:

$$\begin{array}{rcll} S \in Stm & ::= & x := e & \text{assign} \\ & \mid & \textit{skip} & \text{do nothing} \\ & \mid & S_1; S_2 & \text{sequence} \\ & \mid & \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 & \text{test} \\ & \mid & \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done} & \text{loop} \end{array}$$

Typing, static semantics

For mini-while We add declarations for the language:

$$\begin{array}{rcll} P & ::= & D; S & \text{program} \\ D & ::= & \textit{var}\ x : t & \text{type declaration} \end{array}$$

From declarations we infer $\Gamma : Var \to Basetype$ with the two following rules:

$$\frac{}{\textit{var}\ x : t \to_d [x \mapsto t]}$$

$$\frac{D_1 \to_d \Gamma_1 \qquad D_2 \to_d \Gamma_2 \qquad Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset}{D_1; D_2 \to_d \Gamma_1 \cup \Gamma_2}$$

Then a typing judgment for expressions is $\Gamma \vdash e : \tau \in Basetype$. Statements have no type.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash S_1; S_2}$$

$$\frac{D \to_d \Gamma \qquad \Gamma \vdash S}{\emptyset \vdash D; S} \qquad \frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x := e}$$

$$\frac{\Gamma \vdash b : \texttt{bool} \qquad \Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2} \qquad \frac{\Gamma \vdash b : \texttt{bool} \qquad \Gamma \vdash S}{\Gamma \vdash \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done}}$$

Typing configurations: $\Gamma \vdash (S, \sigma) \iff (\Gamma \vdash S \wedge \forall x.\, \emptyset \vdash \sigma(x) : \tau \iff \Gamma(x) = \tau)$

Operational semantics

$$\begin{array}{rl} \textit{Arithmetic} : & Val : \mathcal{E} \to State \to \mathbb{Z} \\ & Val(n, \sigma) = value(n) \\ & Val(x, \sigma) = \sigma(x) \\ & Val(e_1 + e_2, \sigma) = Val(e_1, \sigma) + Val(e_2, \sigma) \end{array}$$

$$\textit{Boolean} : Val : \mathcal{B} \to State \to \mathbb{B}$$

Small step for mini-while $(Stm, State) \Rightarrow (Stm, State)$ or $(Stm, State) \Rightarrow State$

$$(x := e, \sigma) \Rightarrow \sigma[x \mapsto Val(e, \sigma)] \qquad (\texttt{skip}, \sigma) \Rightarrow \sigma \qquad \frac{(S_1, \sigma) \Rightarrow \sigma'}{\big((S_1; S_2), \sigma\big) \Rightarrow (S_2, \sigma')}$$

$$\frac{(S_1, \sigma) \Rightarrow (S_1', \sigma')}{\big((S_1; S_2), \sigma\big) \Rightarrow (S_1'; S_2, \sigma')} \qquad \frac{Val(b, \sigma) = tt}{(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma) \Rightarrow (S_1, \sigma)}$$

$$\frac{Val(b, \sigma) = ff}{(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma) \Rightarrow (S_2, \sigma)}$$

$$(\texttt{while}\ b\ \texttt{do}\ S\ \texttt{done}, \sigma) \Rightarrow (\texttt{if}\ b\ \texttt{then}\ (S; \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done})\ \texttt{else skip}, \sigma)$$

SSA Intermediate representation

- A dominates B if any path from entry to B contains A.
- A strictly dominates B is A dominates B and $A \neq B$.

The domination tree stores the domination relation: A is parent to B if:

- A strictly dominates B
- and A does not stricly dominates any C that strictly dominates B

B belongs to A's dominance frontier if:

- A does not strictly dominate B
- and A dominates a direct predecessor of B.

Dataflow Analysis (liveness) and Abstract Interpretation

- A variable that appears on the left hand side of an assignment is *killed* by the block. Tests do no kill variables.
- A *generated* variable is a variable that appears in the block.

$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell = final \\ \bigcup\{LV_{entry}(\ell') | (\ell, \ell') \in flow(G)\} \end{cases}$$

$$LV_{entry}(\ell) = \big(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\big) \cup gen_{LV}(\ell)$$

Abstract Interpretation Sets X of valuations are abstracted by elements of an abstract domain $(\mathcal{A}, \sqsubseteq)$ s.t. $X \subseteq \gamma(\alpha(X)$. Defining an abstract domain:

- $\mathcal{A}, \sqsubseteq, (\alpha), \gamma, \sqcap, \sqcup$.
- Abstract transfer functions $(+^\sharp, \ldots)$.
- Prove correctness of each operation

Then:

- Perform abstract iterations on the CFG or the AST
- If the lattice is of finite height, iterations terminates on a postfixpoint.
- If not: invent a widening operator to ensure finite convergence. Property $(X \sqsubseteq Y)$: $Y \sqsubseteq X \nabla Y$ + finite chain condition.