Partial Exam
Compilation and Program Analysis (CAP)
October 24th, 2019
Duration: 2 Hours

Instructions :

1. We give some typing/operational/code generation rules in a companion sheet.
2. Explain your proofs, semantic, typing and code generation rules !
3. We give indicative timing.
4. Vous avez le droit de répondre en Français.
5. **Exercises 3 + 4 should be on a separate sheet !**

Solution: In blue, correction remarks and not fully redacted answers.

Exercise #1 ▶ A grammar attribution (15 min)

Inspiration : Schmitz, LSV 2008

Let us consider a fragment of a programming language :

prog	::=	*block*	*program*
block	::=	*stmtlist*	*block*
stmtlist	::=	*stmt*	*listbase*
		|*stmt*; *stmtlist*	*list*
stmt	::=	decl *id*	*var declaration*
		|*exec*	*some execution*
		|begin *block* end	*local block*

For each non terminal *block*, *stmtlist*, *stmt*, *exec* we desire to compute the list of identifiers declared in its scope. For instance, in a program of the form :

```
<exec>_1 ; begin <exec>_2 ; decl i end
```

i is in the list for the non terminal `exec2` but not in the list for `exec1`

Question #1.1

Define **by induction on the grammar** a **synthetised (from leaves to node)** and/or **inherited** attribution that computes the result. Please explain your answer.

Solution: This was not so easy as expected. The idea is to propagate the info ("we are inside a local block") to all statements of the block. This attribute, is of boolean type, named "local", and is inherited :

```
prog -> block    {block.local = false}
block -> smtlist {smtlist.local = block.local}
stmtlist -> stmt  {smt.local = stmlist.local}
          | stmt ; smtlist2 {stmt.local = stmtlist2.local = stmtlist.local}
stmt -> decl id { decl.local = stmt.local}
       | exec   { exec.local = stmt.local}
       | begin block end {block.local = true}
```

We can compute lists of ids : attribution named id, of type list of ids, synthetised :

```
prog -> block    {}
block -> smtlist {block.ids = smtlist.ids}
stmtlist -> stmt  { stmtlist.ids = stmt.ids }
          | stmt ; smtlist2 { stmtlist = stmt.ids union smtlist2.ids}
stmt -> decl id { stmt.ids = {id}}
       | exec   { stmt.ids = {} }
       | begin block end { stmt.ids = {} }
```

Then we can repropagate the stmt ids from top to bottom -inherits, to a final result named ndecl, of type list of ids :

```
prog -> block    {block.decls={}}
block -> smtlist {smtlist.decls = block.decls}
stmtlist -> stmt  { stmt.decls = smtlist.ids}
          | stmt ; smtlist2 { stmt.decls = stmtlist2.decls = stmtlist.ids}
stmt -> decl id { if stmt.local then decl.ids = stmt.ids }
       | exec   { if exec.local then exec.decls = stmt.ids }
       | begin block end { block.decls = {} }
```

A proper redaction would use the course notations and merge the above attributions. Perhaps there are simplier solutions.

Exercise #2 ▶ Mini-While : typing + code generation (20 min)

Here is a program in the Mini-C language seen in the lab :

```
int main() {
    int a,n;
    n = 1;
    a = 7;
    while (n < 67) {
      n = n * 3;
    }
    return 0;}
```

Question #2.1

Show that this program is well-typed (declarations, statements).

Solution: Be careful to apply the declaration rules to get. $\Gamma = \{x \mapsto int, y \mapsto int\}$. And then a proof tree.

Question #2.2

Generate the RISCV 3-address code [1] for the given program **according to the code generation rules**. *Recursive calls, auxiliary temporaries, code, must be separated and clearly described.* .

Solution: I was expecting "real" code generation, thus the recursive calls have to be instantiated and produce real code, link in the exercises we did in the course. I gave very few points to code coming with no explanation.

Solution:
Here is the output of our compiler :

1. We recall that the RISCV three address code has the same instruction set as the RISCV regular code except for conditions which use the idiom `condJUMP(label,t1,condition,t2)` and temporaries/virtual registers instead of regular registers).

```
        # (stat (assignment n = (expr (atom 1)) ;))
        li  temp_2, 1
        mv temp_0, temp_2
        # (stat (assignment a = (expr (atom 7)) ;))
        li  temp_3, 7
        mv temp_1, temp_3
        # (stat (while_stat while (  (expr (expr (atom n)) < (expr (atom 67))) ) (
    stat_block { (block (stat  (assignment n = (expr (expr (atom n)) * (expr (atom 3))) ;))
    ) }))
lbl_l_while_begin_0:
        li  temp_4, 67
        li  temp_5, 0
        bge temp_0, temp_4, lbl_end_relational_1
        li  temp_5, 1
lbl_end_relational_1:
        beq temp_5, zero, lbl_l_while_end_0
        # (stat (assignment n = (expr (expr (atom n)) * (expr (atom 3))) ;))
        li  temp_6, 3
        mul temp_7, temp_0, temp_6
        mv temp_0, temp_7
        j  lbl_l_while_begin_0
lbl_l_while_end_0:
        # Return at end of function:
        li  temp_8, 0
        mv a0, temp_8
```

Exercise #3 ▶ Semantics / typing – direct course application (15 min)

We consider the Mini-While language of the course (see Companion).

Question #3.1

Specify the small step semantics for the `while` construction that does not rely on the `if` statement (you need two rules depending on the valuation of the condition).

Solution:

$$\textsc{NewWhileT}\ \frac{Val(b,\sigma) = True}{(\texttt{while}\ b\ \texttt{do}\ S, \sigma) \Rightarrow (S; \texttt{while}\ b\ \texttt{do}\ S, \sigma)} \qquad \textsc{NewWhileF}\ \frac{Val(b,\sigma) = False}{(\texttt{while}\ b\ \texttt{do}\ S, \sigma) \Rightarrow \sigma}$$

The typing rule for while is still valid but as the reduction rule changed we need to prove again safety.

Question #3.2

Prove that the new semantics still ensures the two type safety properties seen in the course (skip the cases already done in the course), the properties to prove are :

Lemma 1 (progress for mini-while) *If* $\Gamma \vdash (S, \sigma)$, *then there exists* S', σ' *such that* $(S, \sigma) \Rightarrow (S', \sigma')$ *OR there exists* σ' *such that* $(S, \sigma) \Rightarrow \sigma'$.

Lemma 2 (preservation) *If* $\Gamma \vdash (S, \sigma)$ *and* $(S, \sigma) \Rightarrow (S', \sigma')$ *then* $\Gamma \vdash (S', \sigma')$.

Recall that we proved the following lemma in the course :

Lemma 3 *Suppose for all variables* $\sigma(x) : \tau$ *when* $\Gamma(x) = \tau$ *then we have type correctness for* $Val(e, \sigma)$, *formally :*

$$(\forall x \in vars(e). \emptyset \vdash \sigma(x) : \tau \iff \Gamma(x) = \tau) \iff (\Gamma \vdash e : \tau \implies \emptyset \vdash Val(e, \sigma) : \tau)$$

Solution: By induction on the statement we prove Lemma 1. Because of correct typing of while and because of lemma 3, we have b is of type boolean. Thus val(b) evaluates to a boolean , two cases are possible : True or False. In each case we do a reduction $\rightarrow$ (applying the two rules above and collecting the hypotheses is important here).

lemma 1 done by case analysis ; just checking that in each case a reduction is performed

To prove lemma 2 we come back on the previous analysis, and prove that in each case the obtained configuration is well typed *and* Γ *and* σ *agree on typing*. In case True, we re-build the new type of statement by composing existing hypotheses (by typing rule for seq). Detail the typing rule.

Case false is not required to prove lemma 2 but :

Note : This is the part of preservation proven in the course. To really prove preservation (in case of seq) you need another lemma that states preservation of agreement also in the case $(s, \sigma) \rightarrow \sigma'$ Not checked on the exam but *we did it on board during the course.*

In each case we do not modify the store $\sigma \rightarrow$ lemma 2 preserved if Γ and the store are unchanged.

Exercise #4 ▶ Subtyping and Semantics (1h10)

We use the following definition for expressions and statements for a mini-mini-While language : this is the minimal excerpt of the language that is sufficient for studying the problem below.

Expressions :

$$\begin{array}{rcll} e & ::= & c & \textit{constant} \\ & | & x & \textit{variable} \\ & | & e + e & \textit{addition} \\ & | & e/e & \textit{division} \end{array}$$

Statements :

$$\begin{array}{rcll} S(Smt) & ::= & x := e & \text{assign} \\ & | & \textit{skip} & \text{do nothing} \\ & | & S_1; S_2 & \text{sequence} \end{array}$$

Programs declare variables :

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

The typing and semantics rules are recalled in the companion : they are kept unchanged except the ones we explicitly change in the text below (some rules do not apply anymore, e.g. `if` and `while`).

We want to implement a subtyping rule and subtypes for numerical types, i.e. float and int. We restrict ourselves to 2 binary operations : + and / for conciseness. Consider first the following typing rules for expressions :

$$\text{VAR}\ \ \Gamma \vdash x : \Gamma(x) \qquad \text{CONSTINT}\ \ \Gamma \vdash n : \texttt{int} \qquad \text{CONSTFLOAT}\ \ \Gamma \vdash f : \texttt{float}$$

$$\text{PLUS}\ \ \frac{\Gamma \vdash e_1 : \texttt{float} \quad \Gamma \vdash e_2 : \texttt{float}}{\Gamma \vdash e_1 + e_2 : \texttt{float}}$$

$$\text{DIV}\ \ \frac{\Gamma \vdash e_1 : \texttt{float} \quad \Gamma \vdash e_2 : \texttt{float}}{\Gamma \vdash e_1 / e_2 : \texttt{float}} \qquad \text{SUBTYPE}\ \ \frac{\Gamma \vdash e_1 : \tau \qquad \tau <: \tau'}{\Gamma \vdash e_1 : \tau'} \qquad \texttt{int} <: \texttt{float}$$

Question #4.1

Suppose $\Gamma(x) = \texttt{int}$ and $\Gamma(t) = \texttt{float}$. What are the valid types (`int`, `float`, or both ?), in the environment Γ of the following expressions : $x + 3$, $t + 2$. Write the type inference leading to your result.

Solution: both are float. inference is trivial (plus cannot type int).

We now place ourselves in the context of the mini-mini-while language with declarations, and use the subtyping described above. We consider the program :

```
P1= var x:int; var t,n,m:float;
    x:= 6;
    t:= 3.2;
    m:= x/2;
    n:= m/2;
    x:= x+1;
```

Question #4.2

In the program above, which are the lines that are correctly typed, according to the current type-system ? If some lines are not correctly typed, what could be the error message of the type checker ? Explain briefly.

Solution: Only the last line can raise an issue. According to the companion it could type because $\Gamma(x)$ is int but the type judgement used in type checking allows subtyping in this case.

With a more precise type jusgement that would check $\Gamma(x)$ equals the type of the srtored expression we would have : the last statement does not type because $x + 1$ is of type float and x is an integer.

Both were accepted if explained.

Question #4.3

Modify the type system so that operations + and / only produce a `float` type when at least one of the operator has a `float` type.

Solution: One possible solution for division

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1/e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{float} \qquad \Gamma \vdash e_2 : \texttt{float} \qquad \text{not}((\Gamma \vdash e_1 : \texttt{int}) \wedge \Gamma \vdash e_2 : \texttt{int})}{\Gamma \vdash e_1/e_2 : \texttt{float}}$$

Question #4.4

In your new type system, which are the lines of program `P1` that are correctly typed ? If some lines are not correctly typed, what could be the error message of the type checker ? Explain briefly.

Solution: all lines should now type correctly, even the last one in the case of the careful check : this last line types everything as int.

We want to write the semantics of these expressions based on two different division operators : $/.$ is the floating point division and $/_i$ is the integer division. To take into account different division operators, we consider two different options and want to compare them :

Version 1, we use types of computed values :

$$\textsc{divIntV1}\ \frac{Val(e,\sigma) = v \qquad Val(e',\sigma) = v' \qquad \emptyset \vdash v : \texttt{int} \qquad \emptyset \vdash v' : \texttt{int}}{Val(e/e',\sigma) = v /_i v'}$$

$$\textsc{divFloatV1}\ \frac{Val(e,\sigma) = v \qquad Val(e',\sigma) = v' \qquad \text{not}(\emptyset \vdash v : \texttt{int}) \vee \text{not}(\emptyset \vdash v' : \texttt{int})}{Val(e/e',\sigma) = v /. v'}$$

Version 2, we remember the typing environment and use it to type variables, recalling the expression type checker during operations :

$$\textsc{divIntV2}\ \frac{Val(e,\sigma) = v \qquad Val(e',\sigma) = v' \qquad \Gamma \vdash e : \texttt{int} \qquad \Gamma \vdash e' : \texttt{int}}{\Gamma \vdash Val(e/e',\sigma) = v /_i v'}$$

$$\textsc{divFloatV2}\ \frac{Val(e,\sigma) = v \qquad Val(e',\sigma) = v' \qquad \text{not}(\Gamma \vdash e : \texttt{int}) \vee \text{not}(\Gamma \vdash e' : \texttt{int})}{\Gamma \vdash Val(e/e',\sigma) = v /. v'}$$

Question #4.5

For each version, what is the value of each variable at the end of the execution of program `P1` ?

Solution: Trivial part : $m = 3$ and $x = 7$

In V1 all divisions are integer division and thus at the end $n = 3/_i 2 = 1$. In V2, the division is a floating point division $n = 3/.2 = 1.5$.

Recall the notion stating that a typing environment Γ and a store σ agree on the type of variables, defined in the course as follows :

$$\textsc{Agreement}$$
$$\forall x. \emptyset \vdash \sigma(x) : \tau \iff \Gamma(x) = \tau$$

It is used to prove type progress and type safety

Question #4.6

AGREEMENT is too strong to be useful. Indeed suppose $\sigma(x) = 42$, then $\emptyset \vdash \sigma(x) : \texttt{int}$ but also $\emptyset \vdash \sigma(x) : \texttt{float}$ and $\Gamma(x)$ cannot be `int` and `float`. Find a weaker, valid version of "a typing environment Γ and a store σ agree on the types of variables".

Solution:

$$\forall x. \Gamma(x) = \tau \implies \emptyset \vdash \sigma(x) : \tau$$

AND the domain of σ is the same as the domain of Γ.

There is an alternative formulation that ensures $\tau \leq \Gamma(x)$ (however it must be checked carefully, it is easy to have it wrong).

Question #4.7

For each version of expression valuation, prove type correctness for expression evaluation (use the typing rules you defined in **Question 4.3** :

> Suppose Γ and a store σ agree on the type of variables (as you defined in the previous question) prove type correctness for $Val(e, \sigma)$, i.e. : $\Gamma \vdash e : \tau \implies \emptyset \vdash Val(e, \sigma) : \tau$

Explain the general proof schema (how the recurrence is built), and detail the proof in the case of the / operator.

Solution: V1 : structural induction on the expression similar to the correction in the course except new operations :

correct typing for division implies

— either the result is an int and

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1/e_2 : \texttt{int}}$$

> By induction hypothesis we obtain $Val(e_1, \sigma) : int$ and $Val(e_2, \sigma) : int$ the integer division is thus applied, we obtain an int ; they cannot be float if they are typed int.

— or the result is a float and

$$\frac{\Gamma \vdash e_1 : \texttt{float} \qquad \Gamma \vdash e_2 : \texttt{float} \qquad \text{not}((\Gamma \vdash e_1 : \texttt{int}) \wedge \Gamma \vdash e_2 : \texttt{int})}{\Gamma \vdash e_1/e_2 : \texttt{float}}$$

> By induction hypothesis we obtain $Val(e_1, \sigma) : float$ and $Val(e_2, \sigma) : float$ and ... we do not know an integer might be typed float, thus either the int division or the float one is applied depending on the real value. At the end types are still preserved because $v/_i v'$ is of type int but also float because $int <: float$ and we apply the subtyping rule.

la redaction complete avec l utilisation des regles d inferences et tout est pas courte !

V2 : idem except the second item above, which concludes differently :

By induction hypothesis we obtain $Val(e_1, \sigma) : float$ and $Val(e_2, \sigma) : float$ and ... the float division is applied because it is the type of e_1 and e_2 that is checked . $v/.v'$ is of type float.

Question #4.8

In the course, to prove type safety we defined a configuration as a couple statement, store : (S, σ). A configuration (S, σ) is well typed in Γ iff the statement is well typed, $\Gamma \vdash S$, and Γ and σ agree on the types of variables. As the second definition has slightly changed, type safety has to be proven again ... we focus on one interesting case : assignment. 1) recall the small step reduction rule for assignment (denoted $\Rightarrow$) ; 2) Prove preservation for assignment, i.e. :

> If $\Gamma \vdash (x := e, \sigma)$ and $(x := e, \sigma) \Rightarrow \sigma'$ then Γ and σ' agree on the types of variables (with your new definition of AGREEMENT).

Solution:

$$(x := e, \sigma) \Rightarrow \sigma[x \mapsto Val(e, \sigma)]$$

$\Gamma \vdash (x := e, \sigma)$ implies that there is a τ s.t. $\Gamma \vdash e : \tau$ and $\Gamma(x) = \tau$ to be able to apply the type rule for assignment

We proved above that $\emptyset \vdash Val(e, \sigma) : \tau$ which is sufficient to ensure (knowing that Γ and σ agree on types of variables) :

$$\forall x. \Gamma(x) = \tau \implies \emptyset \vdash \sigma(x) : \tau$$

AND the domain of σ is the same as the domain of Γ.

Remark : the global induction to prove type safety relies on the preceding proofs, and other cases that are kept similar wrt the course

3 address-code generation In the RISCV double-precision extension, the machine keeps its 64-bit registers $(x_0, \ldots x_{31})$, and has additional 64-bits floating point registers $(f_0, \ldots f_{31})$, the extension also defines the following instructions :

- `fsub, fadd, faddi` for the floating points operations.
- `fcvt.l.d xi, fi` converts a floating point register to an integer register (with a rounding mode that we do not really care of here).
- `fcvt.d.l fi, xi` makes the converse.
- `fmv.s` is the floating-point move.
- There is no immediate load, but consider there exists one : `fli`.
- There is no floating point register containing zero, deal with it !

We define a function Type that takes an expression and returns the most specific type for this expression, i.e. if e 42 = `int`, even if 42 could also be typed as `float`. For code generation, the Type function can also be used on registers and temporary variables.

Now let us come to the code generation phase. We recall in the companion sheet the code generation rules of the course where the only numerical type was `int`. The code generation rules we ask for **should have the same signature**.

Question #4.9

Write the new code generation rules for expressions (constants, variable, division) and assignments with the help of the Type function and these helpers :

— `newTempInt` : $() \to \mathbb{N}$ denoted by $tempi_k$.
— `newTempFloat` : $() \to \mathbb{N}$ denoted by $tempf_k$.
— `newLabel` : $() \to \mathbb{N}$.

Solution:

cst c	`match type(c) with:`
	`| int -> let r = newTempInt() in`
	`code.add(instructionLI(r,c))`
	`| float -> let r = newTempFloat() in`
	`code.add(instructionFLI(r,c))`

$e_1 + e_2$	`dr1 = GenCodeExpr(e1)`
	`dr2 = GenCodeExpr(e2)`
	`match type(e1,e2) with:`
	`| int,int -> let r = newTempInt() in`
	`code.add(instructionADD(r,dr1,dr2))`
	`| int, float -> let r = newTempFloat() in`
	`let dr1b = newTempFloat() in`
	`code.fcvt.d.l dr1b dr1`
	`code.add(instructionFADD(r,dr1b,dr2b))`
	`| ...`

x = e	`dr <- GenCodeExpr(e)`
	`#a code to compute e has been generated`
	`find loc the location for var x`
	`match type(x),type(dr) with:`
	`| int,int -> code.add(instructionMV(loc,dr))`
	`| float,float -> code.add(instructionFMV(loc,dr)`
	`| int, float -> # this does not happen in practice but the code would be`
	`let reg=newtempInt() in`
	`code.add(instructionFCVT.L.D(reg,dr)`
	`code.add(instructionMV(loc,reg))`
	`| float, int -> let reg=newtempFloat() in`
	`code.add(instructionFCVT.D.L(reg,dr)`
	`code.add(instructionFMV(loc,reg))`

Question #4.10

Generate the code for the three last assignments of program `P1`.

Solution:

```
m = x/2  -> int division + float cast while assignment
n = m/2  -> div float and no  cast in assignment
x = x+1 --> no problem
```

Question #4.11

What semantics for Val corresponds to this behavior ?

Solution: We do not use the dynamic types (V1), which is impossible since we are compiling ! If any operand is of type float, then the division is float, thus we are implementing V2.

Type annotation semantics

Question #4.12

Calling the type checker at each step of the code generation is not efficient. Design a variant of the type system that annotates expressions with types. More precisely, let et be the set of expressions with annotated types :

$$et ::= (x : \tau)|(c : \tau)|(et + et : \tau)|(et/et : \tau)$$

— The typing judgment now has the form : $\Gamma \vdash e : et$. Write the new rule for variables (VAR), and the rules for the / operator (replacing the DIV rule).

— The expression valuation now has the form $Val(et, \sigma) = v$ write the definition of expression valuation in case of a variable : $Val(et, \sigma)$ and in case of the division.
Can you define the equivalent [2] of DIVINTV1 and DIVFLOATV1 without a call to the type checker ? If no, why ? if yes give the definition and explain it.
Can you define the equivalent[2] of DIVINTV2 and DIVFLOATV2 without a call to the type checker ? If no, why ? if yes give the definition and explain it.

For convenience you might use variables ee, ee', ... to range over expressions with type annotations inside but not at top level, e.g. : `(x:int)+(y:float)`.

Solution: 1)

$$\Gamma \vdash x : (x : \Gamma(x)) \qquad \frac{\Gamma \vdash e_1 : (ee : \texttt{int}) \qquad \Gamma \vdash e_2 : (ee' : \texttt{int})}{\Gamma \vdash e_1/e_2 : (ee/ee' : \texttt{int})}$$

$$\frac{\Gamma \vdash e_2 : (ee' : \texttt{float}) \qquad \begin{array}{c}\Gamma \vdash e_1 : (ee : \texttt{float}) \\ \nexists ee_1, ee_2.\, ((\Gamma \vdash e_1 : (ee_1 : \texttt{int})) \wedge \Gamma \vdash e_2 : (ee_2 : \texttt{int}))\end{array}}{\Gamma \vdash e_1/e_2 : (ee/ee' : \texttt{float})}$$

2) Version 1 is not possible because we do not have the dynamic tyoe of expressions
Version 2 :

$$\text{DIVINTV2}\ \frac{Val(et, \sigma) = v \qquad Val(et', \sigma) = v' \qquad et = (ee : \texttt{int}) \qquad et' = (ee' : \texttt{int})}{\Gamma \vdash Val(e/e', \sigma) = v /_i v'}$$

$$\text{DIVFLOATV2}\ \frac{Val(e, \sigma) = v \qquad Val(e', \sigma) = v' \qquad (et = (ee : \texttt{float}) \vee et' = (ee' : \texttt{float}))}{\Gamma \vdash Val(e/e', \sigma) = v /. v'}$$

2. By equivalent here we mean a rule with the same semantics, i.e. returning the same result

It works because if we tag as float, we know that it cannot be typed as int (there is a unique tag valid for operands of / and it is the most specific one)

Question #4.13

Open question : In the implementation of an evaluator in Python, how would you implement each of the two versions of the semantics. Explain and give a few code sketches. Note : recall that Python only provides dynamic types.