Final Exam
Compilation and Program Analysis (CAP)
January, 25th, 2019
Duration: 3 Hours

Instructions :

1. The first 2 exercices are to be done on the exam subject.
2. Return the whole exam subject.
3. We give some typing/operational/code generation rules in a companion sheet.
4. Explain your results !
5. We give indicative timing.
6. Vous avez le droit de répondre en Français.

Numéro d'anonymat (reporter le numéro inscrit sur la copie d'examen "officielle") :

Solution: In blue, correction remarks and not fully redacted answers. Exam with 30 pts : ex1 is 7,5pts, exo2 is 2,5 pt, exo3 : 8pts, exo4 : 3pts, exo5 : 9pts.

Exercise #1 ▸ Lab questions

Grammars/ Attributes

Question #1.1

Explain how the visitor pattern works when traversing a parse tree.

Solution: Cf explanation in Lab 3.

Question #1.2

Explain how do you implement a given grammar attribution with synthetised (from terminal to root) attributes with a ANTLR-Python/visitor.

Solution: The attributes are propagated from terminal to root by function calls. A given visitor returns its computed attribute, and its parents has an access via a call to $visit(grammarelement)$.

Question #1.3

Same question with herited attributes.

Solution: The only way is to have a global variable to update when traversing.

Typing

Question #1.4

Explain how did/would you implement implicit type coercion (dealing with 1+0.14 for instance) in your MuTyping visitor.

Solution:

Question #1.5

Complete the visitor method to check the typing of the "while" construct of the Mu language :

```
def visitWhileStat(self, ctx):






#
```

Evaluation

Question #1.6

Complete the visitor for the condition of a while statement.

```
def visitCondBlock(self, ctx):
#cond : condition / stat-block : the statement
# exec the stat-block and return True if the cond evaluates
    to True
# else return False.
```

```
#
```

Question #1.7

During the evaluation of a given mini-while program, what happens if the program loops ?

Solution:

Dataflow

Question #1.8

Complete the initialisation of Kill and Gen sets for the liveness analysis :

```
    def addInstructionADD(self, dr, sr1, sr2orimm):
        if isinstance(sr2orimm, Immediate):
            ins = Instru3A("add3i", dr, sr1, sr2orimm)
        else:
            ins = Instru3A("add3", dr, sr1, sr2orimm)
         # add in  ins._gen and / or ins._kill if necessary




        self.add_instruction(ins)
```

Question #1.9

From the result of the dataflow analysis, how would you eliminate dead code (course + lab question) ?

Solution: From the liveness dataflow analysis, if a given variable is defined (killed) in a block and not alive afterwards, then this definition is dead code.
In the lab it would mean eliminating a block in the CFG.

Smart Code Generation

Question #1.10

Complete the squares with the order of the different compiler phases.

- ☐ Conflict graph construction
- ☐ Typechecking
- ☐ Liveness analysis
- ☐ Lexical and syntactic analysis

- ☐ Allocation with actual registers
- ☐ 3-address code generation
- ☐ Register allocation via graph coloring

Solution: cf course.

Question #1.11

Complete the generated code for the 3 address instruction `add temp_1 temp_2 temp_3` where t_1 is allocated in memory at offset 2 (2×16 bits in memory), t_2 in register r_3 and t_3 in memory at offset 0 :

```
 1  ; add3 t1 t2 t3
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13  ;
```

Exercise #2 ▶ A grammar attribution

Solution: Adapted from M. Nebut for Univ Lille, 2005.

We consider the problem of modelizing the composition of software components. A component (depicted in Figure 1) has (possibly 0) inputs and (possibly 0) outputs. It can be composed with serial or parallel composition (Figure 2).

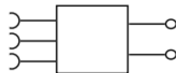


Figure 1 – A component

We propose the grammar depicted in Figure 3 where $< nbi, nbo >$ depicts a base components with *nbi* inputs and *nbo* outputs, the serial composition is the concatenation, the parallel composition is denoted by &. The start symbol is C, all non-terminals are in capital letters.

Question #2.1

According to this grammar, which of serial composition or parallel composition is prioritary ?

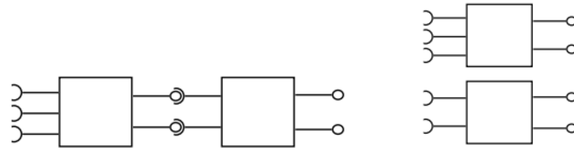


FIGURE 2 – Component composition serial (left) parallel (right)

$$
\begin{array}{lcl}
C & \longrightarrow & P\ C' \\
C' & \longrightarrow & \texttt{\&}\ P\ C' \mid \varepsilon \\
P & \longrightarrow & E\ P' \\
P' & \longrightarrow & E\ P' \mid \varepsilon \\
E & \longrightarrow & \texttt{< int ; int >} \mid \texttt{(}\ C\ \texttt{)}
\end{array}
$$

FIGURE 3 – The component grammar

Solution: The "chain" `a b & c d` is parsed as `(a b)& (c d)` (a,b,c,d here are 'E' elements) because the rules deriving `&` are "higher" in the grammar than the ones deriving sequence (P and P' rules). Thus sequences are prioritary to parallel composition.

Question #2.2

Give the expression (conform to the grammar) associated to the composition depicted in figure 4.

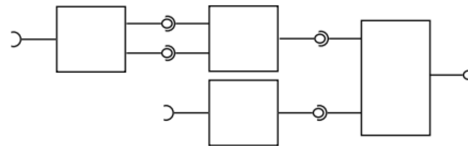


FIGURE 4 – A composition

Solution: `( < 1 ; 2 > < 2 ; 1 > & < 1 1 > ) < 2 1 >`

Question #2.3

Draw the parse tree for this example.

Solution: No difficulty.

Of course all compositions must respect the following rule :

— For serial composition c_1c_2, the number of inputs of c_2 should be equal to the number of outputs of c_1.

For instance `<2 ; 3 > ( < 1 ; 2 > <2 ; 1 > & < 1 ; 3> )` is syntactically correct but should be rejected because it doesn't respect the "serial" conditions.

Question #2.4

Write an attribution for the grammar to reject improper expressions. *Do not forget to propely define all attributes, their types, say if they are synthetised or herited, and give semantic attributes (right-hand side of the grammar rules) to compute them.*

Solution: It is sufficient to propagate *nbi,nbo* ("inputs, outputs"), deux synthetised attributes of type int, from base blocs where you only propagate lexing info, (rule E), to other non terminal C, C', P , P', and at P or P' rules you have to check if $E.nbo = E.nbi$ and raise an Error if if is not the case.

Solution:

Solution: Adapted from Thibaut Balabonski for Univ. PSud, 2016.

We consider a mini imperative language with optional values similar to the option type of Ocaml. Hence, `Some e` is an optional value computed by `e` and `None` is an absent optional value. To access to the value of an option `e`, we invent a construction `?e:d` that evaluates e, then :

- if it evaluates to `Some v`, then the result is v
- if it evaluates to `None`, then the result is the value of expression d ("default result").

The language instructions are :

- `while e { S }` which executes the statement block S until the evaluation of the expression e is different from 0.
- `ifz e { S }` which executes S if the evaluation of e gives 0.
- Assignements `x:= { S e }` executes S then assigns to x the value of e.

A program is a list of variable declarations, followed by an instruction bloc (list) followed by an expression whose evaluation is the final value. Base types are int, bool (but now you have constructed types). **For notations, take inspiration from the companion sheet**

Question #3.1

Give an abstract syntax for these programs.

Solution: Programs, smts, expressions :

```
vardecl : var : typ

e ::= c | v | Some e | None | ?e : d | e+e | e * e

smt ::= x := {smt e} | smt;smt | eps | ife z smt | while e smt

prog ::= vardecl+ { smt e } # with a ';' if you want
```

Question #3.2

What is the form of a typing judgment? Give rules to type programs (without declarations) under a given environment Γ. Explain.

Solution: yes, the possibilities for types are not defined at this step of the exercise (sorry!), but if you read a little bit in advance, you see that you have basic types and option types Typing judgments for expressions : $\Gamma \models e : \tau$ says that e is of type τ in the Γ environment. The rest is rather classical, except that we want a type option of any type for None :

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

$$\frac{}{\Gamma \vdash \texttt{None} : \tau\ \texttt{option}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{Some}\ e : \tau\ \texttt{option}}$$

For expressions of the form $?e_1 : e_2$, be careful we cannot execute :

$$\frac{\Gamma \models e_1 : \tau\texttt{option} \quad \Gamma \models e_2 : \tau}{\Gamma \models ?e_1 : e_2}$$

We write $\Gamma \models p : \tau$ to say that p is a well-typed program that produces a value of type τ ; and $\Gamma \models smt$ to say that a given statement is well-types (we could have typed "void") :

$$\frac{\Gamma \vdash b \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \{\ b\ e\ \} : \tau} \qquad \frac{}{\Gamma \vdash \emptyset} \qquad \frac{\Gamma \vdash b \qquad \Gamma \vdash i}{\Gamma \vdash b\ i;}$$

$$\frac{\Gamma(x) = \tau \qquad \Gamma \vdash p : \tau}{\Gamma \vdash x := p} \qquad \frac{\Gamma \vdash e : \mathtt{int} \qquad \Gamma \vdash b}{\Gamma \vdash \mathtt{ifz}\ e\ \{\ b\ \}} \qquad \frac{\Gamma \vdash e : \mathtt{int} \qquad \Gamma \vdash b}{\Gamma \vdash \mathtt{while}\ e\ \{\ b\ \}}$$

Question #3.3

Are these program well-typed withint $\Gamma(x) = int, \Gamma(y) = int\ option$? *proof trees if possible.*

```
P1 : while x { y := {Some (x+1)} ; x := { ?y:0 }; }
P2 : y := { Some 1 }; 1+y
```

Solution: P1 ok, not P2 because addition of an int option and a int.

Question #3.4

Give rules to construct typing environments from your declarations. Explain.

Solution: Same as in the course.

Operational Semantics. A value computed by programs is either an int/ a bool, or emptyset, or a set containing a value :

$$v ::= n|b|\emptyset|\{v\}$$

(be careful these are sets, do not make the confusion with the curly brackets used in the syntax of our language).

Question #3.5

Take inspiration from the companion file and give a denotational semantics for expressions.

Solution:

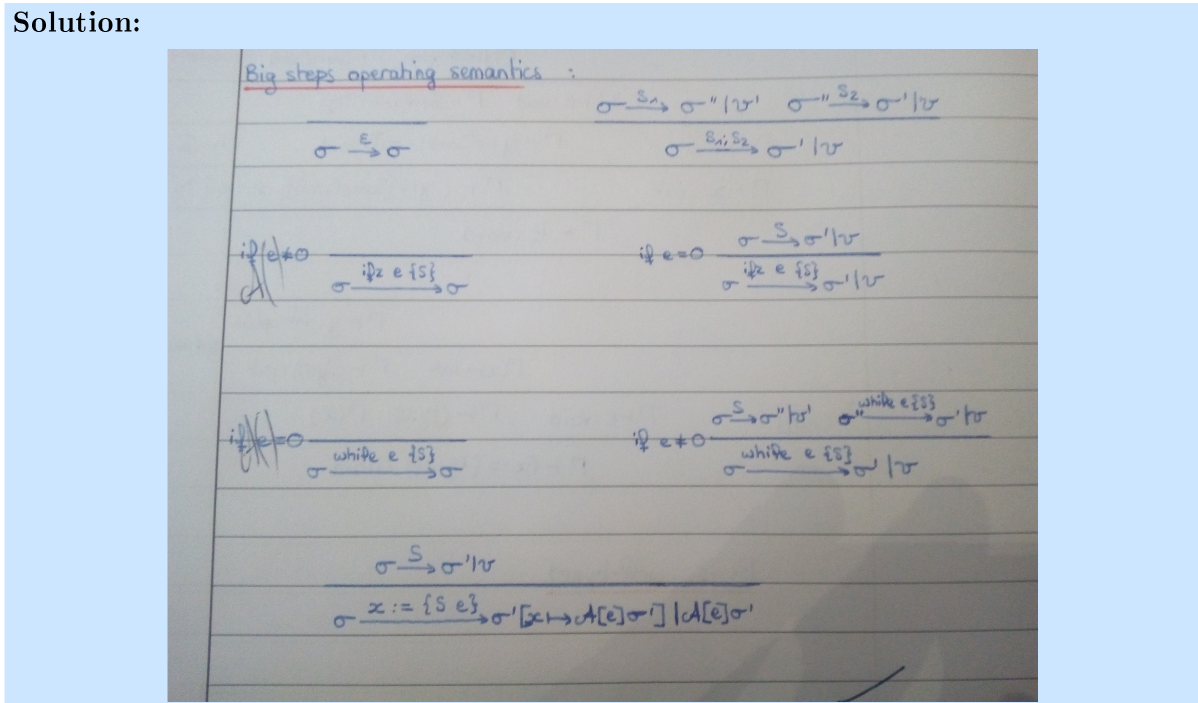
$$\begin{array}{rcll} [\![n]\!]_S & = & n & \\ [\![x]\!]_S & = & S(x) & \\ [\![\mathtt{None}]\!]_S & = & \emptyset & \\ [\![\mathtt{Some}\ e]\!]_S & = & \{[\![e]\!]_S\} & \\ [\![e_1\ +\ e_2]\!]_S & = & [\![e_1]\!]_S + [\![e_2]\!]_S & \\ [\![?e_1 : e_2]\!]_S & = & v_1 & \text{si } [\![e_1]\!]_S = \{v_1\} \\ [\![?e_1 : e_2]\!]_S & = & [\![e_2]\!]_S & \text{si } [\![e_1]\!]_S = \emptyset \end{array}$$

Let us denote by $\sigma \xrightarrow{S} \sigma|v$ the following concrete semantics judgment meaning : "given a memory state σ, the instruction(s) S leads to the new state σ' and produces the value v. For the semantics of statements that do not produce a value, $\sigma \xrightarrow{S} \sigma'$ is the simplified version. ε denotes the empty instruction.

Question #3.6

Give big steps operating semantics for our language (6 rules).

Solution:



To facilitate the analysis of programs, we want to flatten the assignments instructions, so that to transform for instance the program (without declaration) :

```
{ x := { y := { Some 2 };
         ?y:0 + 1
        } ;
  x }
```

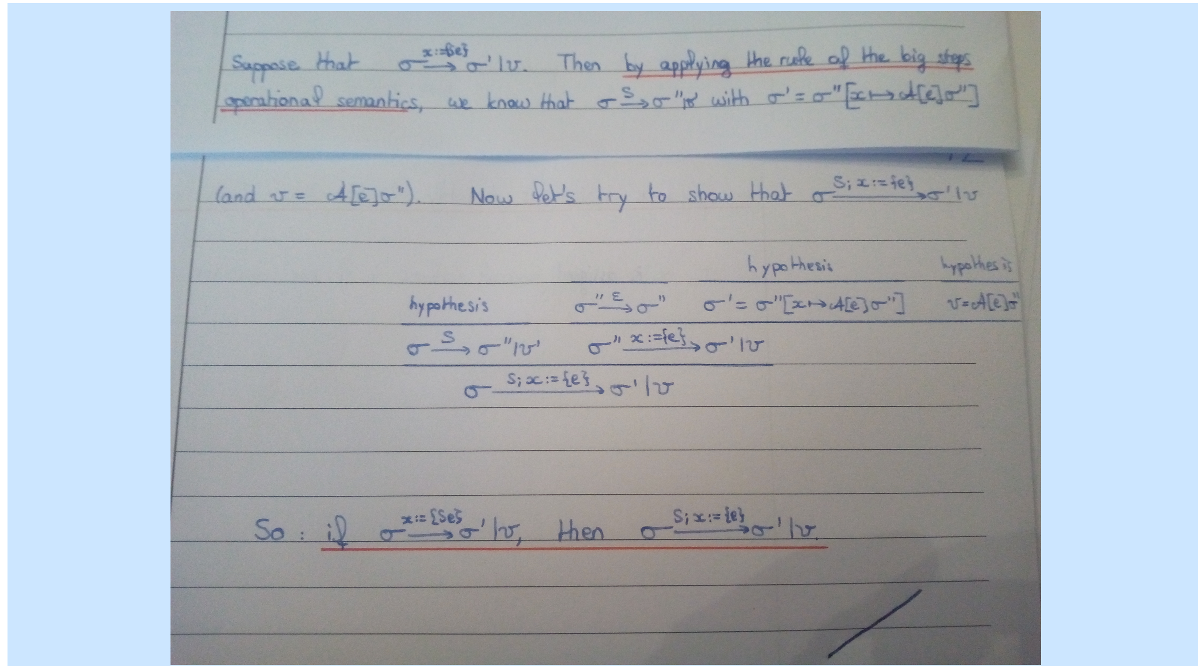
into

```
{ y:= { Some 2 };
  x := {?y:0+1};
  x
}
```

Question #3.7

Prove that if $\sigma \xrightarrow{x:=\{Se\}} \sigma|v$, then $\sigma \xrightarrow{S;x:=\{e\}} \sigma|v$,

Solution: Some well redaction, thank you, anonymous student :



Suppose that $\sigma \xrightarrow{x:=\{S;e\}} \sigma' | v$. Then by applying the rule of the big steps operational semantics, we know that $\sigma \xrightarrow{S} \sigma'' | v'$ with $\sigma' = \sigma''[x \mapsto \mathcal{A}[e]\sigma'']$ (and $v = \mathcal{A}[e]\sigma''$). Now let's try to show that $\sigma \xrightarrow{S;\, x:=\{e\}} \sigma' | v$

$$\dfrac{\dfrac{\text{hypothesis}}{\sigma \xrightarrow{S} \sigma'' | v'} \qquad \dfrac{\sigma'' \xrightarrow{E} \sigma'' \qquad \dfrac{\text{hypothesis}}{\sigma' = \sigma''[x \mapsto \mathcal{A}[e]\sigma'']} \qquad \dfrac{\text{hypothesis}}{v = \mathcal{A}[e]\sigma''}}{\sigma'' \xrightarrow{x:=\{e\}} \sigma' | v}}{\sigma \xrightarrow{S;\, x:=\{e\}} \sigma' | v}$$

So : if $\sigma \xrightarrow{x:=\{S;e\}} \sigma' | v$, then $\sigma \xrightarrow{S;\, x:=\{e\}} \sigma' | v$.

A simple code generation scheme would be to generate a test for each expression `?x:e` to determine if the value of x is of the form $\{v\}$. We want to avoid such a test in the case where we can predict that the value of x has the good shape, so we propose a dataflow algorithm to get some information.

We say that a variable x of type `type option` is *defined* at a given control point if for all execution path reaching this point the last assigment to x is of the form `x:= { Some e }`.

For simplicity, we consider here that assigments of the form `x := { e' }` where e' is not directly of the form `Some e` do not cause x to de defined, even if we could be able to predict that e' could produce a value $\{v\}$.

Question #3.8

Give the dataflow equations that enable to determine, for each node of the CFG, the set of defined variables. Is the analysis forward, backward ? Where does the information comes from ? Give definitions of *Gen* and *Kill* sets for statements.

Solution: The propagation should be forward, from definitions to end of programs. The information originates from assigments $x :=$ .... If a given variable is assigned to `Some e` then it is generated, else it is killed. For a given block, the set of defined vars at is entry is the intersection of all its predecessors. At the exit, those defined before, except kill, union gen.

Question #3.9

For the following program, give the CFG, and solve your equations. Which expressions are you able to optimise ?

```
{ x := { Some 1 };
  y := { Some x };
  ifz 1 {
    x := { None };
    z := { Some 0 };
```

```
    };
    while ?x:0 {
     x := { Some 2 };
     y := { Some ?y:None };
     z := { Some ?x:1 };
    };
?x:0 + ??y:Some 0:1 + ?z:0
}
```

Solution: flemme.

Question #3.10

(Bonus) Give a program where we could predict that x produces a value $\{v\}$ but we fail with the previous definition/analysis. Propose a refinement of the analysis.

Exercise #4 ▶ Abstract Interpretation 1 : affine equations

Solution: From a problem of Eric Goubault for Polytechnique.

We consider here the abstract domain $\mathcal{K}$ of affine relationships between variables, introduced by M. Karr in 1974. For a set of program variables (such as the ones appearing in a Mu/miniwhile program) $\mathcal{V} = \{v_1, \dots, v_n\}$, we associate an abstract value made of $k \leq n$ independent affine relationships :

$$a_1^i v_1 + \dots a_n^i v_n = b^i$$

$(i = 1, \dots, k)$

Question #4.1

Show that $\mathcal{K}$ is a lattice (ordered set where all pairs of elements have a unique sup and inf). You can informally define join and meet operations.

Question #4.2

Express the concretisation operator $\gamma : \mathcal{K} \to \mathcal{P}(\mathbb{R}^n)$.

Question #4.3

Abstract transfer function for + : what would be the result of the application of the abstract transfer functions for the assignment $v_3 = v_2 + v_1$ starting with the abstract value :

$$\begin{cases} -v_1 + v_3 & = & 1 \\ v_1 - v_2 & = & 3 \end{cases}$$

Question #4.4

(Bonus) Do we need a widening operator ? (is there a maximal number of iterations ?) Give simple arguments, no need for a full proof.

Solution: All is about affine spaces. Concretization is nothing more that "id". https://www.di.ens.fr/~rival/semverif-2015/sem-12-ai.pdf

be careful for the transformation, since v_3 is changed, the first equation is not true any more. I gave bonus points to people that did a matrix implementation of the operations. Do not forget top and bottom.

Exercise #5 ▶ Abstract Interpretation 2 : linear patterns

The analysis defined in this problem is the "template" analysis from Sriram Sankaranarayanan and Zohar Manna, 2005.

Solution: Problem and solution (in French, sorry), by S. Putot for Polytechnique, some questions were adapted. I introduced a typo that was not Sylvie's one, sorry, we should have $\mathcal{P}(\mathbb{Z}^n)$ and not $\mathbb{Z}$.

Let $A = (a_{ij})_{i=1,\dots,n,j=1,\dots,m}$ be a matrix of real numbers, with $m \geq 1$. We call the domain D_A, parametrized by A, the set of abstract elements b, where b is a vector $(b_1, \dots, b_m)$ of real numbers or plus or minus infinity (∞ ou $-\infty$), representing the set of values that can take the n variables $x = (x_1, \dots, x_n)$, satisfying $Ax + b \geq 0$, i.e.

$$\begin{cases} a_{11}x_1 + \ldots + a_{1n}x_n + b_1 \geq 0 \\ \ldots \\ a_{m1}x_1 + \ldots + a_{mn}x_n + b_m \geq 0 \end{cases}$$

The abstract ordering $b \preceq b'$ in D_A is the inclusion of all the points $x = (x_1, \dots, x_n)$ satisfying $Ax + b \geq 0$ in the set of points satisfying $Ax + b' \geq 0$.

In the sequel, we might have to use the fact that we can minimise linear functions on set of values described by an abstract element b of D_A using linear programming.

An instance of a linear programming problem $LP(A, b, c)$ is given by an objective function $f(x) = c^T x = \sum_{i=1}^n c_i x_i$ for all $x = (x_1, \dots, x_n) \in \mathbb{R}^n$ and also a set of affine inequalities $Ax+b \geq 0$.

The LP algorithm applied to A, b, c, $LP(A, b, c)$, can return one of the following three results :

- A vector x reaching the minimum of f on the x defined by $Ax + b \geq 0$
- Some solutions x but none of them reaches the minimum of f on the x such that $Ax + b \geq 0$ because f is not bounded on $\{x \mid Ax + b \geq 0\}$, non empty.
- No solution because $\{x \mid Ax + b \geq 0\}$ is empty

Question #5.1

In the abstract domain, what does the element $(b_1, \dots, b_{i-1}, -\infty, b_{i+1}, \dots, b_m)$ represents (whatever be the values of b_1, ..., b_{i-1}, b_{i+1}, ..., b_m, and the index i). What about $(\infty, \dots, \infty)$?

Solution: Respectivement $\bot$ et $\top$ (comme $m \geq 1$).

Question #5.2

Suppose $n = 2$, and A has the value

$$A_1 = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix}$$

then

$$A_2 = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 1 & -1 \\ -1 & 1 \end{pmatrix}$$

What are the abstract domains D_{A_1} and D_{A_2} ? Describe the kind of information they can infer.

Solution: D_{A_1} correspond au domaine des intervalles et D_{A_2} au domaines des zones, c'est à dire des unions de contraintes de la forme $v_i - v_j \leq cte$ (donc des rectangles un peu rabotés mais toujours dans un certain sens, faire un dessin).

Question #5.3

We recall that the abstract domain of octagons express constraints of the form $\pm x \pm y \leq c$. Define a matrix A corresponding to the octagons domains, with $n = 2$. Can we define the polyhedra abstract domain ?

Solution: La matrice A_3 correspondant au domaine des octogones est par exemple :

$$A_2 = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 1 & -1 \\ -1 & 1 \\ 1 & -1 \\ -1 & 1 \\ 1 & 1 \\ -1 & -1 \end{pmatrix}$$

On ne peut pas définir le domaine général des polyèdres par patrons linéaires car il faudrait représenter une infinité de faces possibles, statiquement, dans la matrice correspondante.

Question #5.4

Define the concretization function : $\gamma : D_A \rightarrow \mathcal{P}(\mathbb{Z})$. should be $\mathbb{Z}^n$

Solution: The concretisation of an abstract value is the set of variables values that satisfy the constraints...

Question #5.5

For the given two instructions :

```
x = x+1;
y = y+2;
```

what should be the final abstract value for the linear template D_A with :

$$A = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 1 & 1 \\ -1 & -1 \end{pmatrix}$$

with the initial abstract value $b = (-1, 2, -2, 3, -4, 4)$?

Solution: On a donc au départ :

$$\begin{array}{rcl} x & \geq & 1 \\ x & \leq & 2 \\ y & \geq & 2 \\ y & \leq & 3 \\ x+y & = & 4 \end{array}$$

Et on doit minorer et majorer, après les deux affectations, les formes linéaires :

$$\begin{array}{l} x+1 \\ y+2 \\ x+y+3 \end{array}$$

On a facilement :

$$\begin{array}{rcl} x+1 & \geq & 2 \\ x+1 & \leq & 3 \\ y+2 & \geq & 4 \\ y+2 & \leq & 5 \\ x+y+3 & = & 7 \end{array}$$

sous les contraintes initiales sur x et y. Donc la nouvelle valeur abstraite est $b' = (-2, 3, -4, 5, -7, 7)$.

Question #5.6

Give an algorithm to compute the abstract transfer function for $+$: given A and an abstract element b, how do you compute b' the result of the interpretation of $x_k := x_i + x_j$ where i, j and k are indices between 1 and n. Give arguments for correctness, and the fact we have the best abstraction.

Solution: Supposons que l'on cherche à interpréter l'affectation $x_k = x_i + x_j$ où i, j et k sont des indices entre 1 et n. Les valeurs minimales des fonctions linéaires que l'on doit déterminer et qui sont susceptibles de changer après affectation, sont notées $f_1, \ldots, f_u$, ce sont les lignes de la matrice A qui ont une entrée non nulle en la colonne k. Pour une telle fonction linéaire f_v ($v = 1, \ldots, u$), on considère le programme linéaire de fonction objectif $f_v(x_1, \ldots, x_{k-1}, x_i + x_j, x_{k+1}, \ldots, x_n)$ et de contraintes, celles données par $Ax + b \geq 0$. Notons ${b_v}'$ la solution s'il existe, sinon ∞ si f_v est non borné sur le domaine de contraintes, sinon $-\infty$. On prouve maintenant que dans D_A, b' où on a remplacé les b_v par les ${b_v}'$, représente bien une

sur-approximation des valeurs après affectation. On doit donc prouver que

$$\{(x_1, \ldots, x_{k-1}, x_i + x_j, x_{k+1}, \ldots, x_n) \mid (x_1, \ldots, x_n) \in \gamma(b)\} \subseteq \gamma(b')$$

Autrement dit, on doit prouver que pour tout $(x_1, \ldots, x_n) \in \gamma(b)$, $f_i(x_1, \ldots, x_{k-1}, x_i+x_j, x_{k+1}, \ldots, x_n) \geq b'_i$ pour tout i entre 1 et m, f_i étant la forme linéaire déduite de la ième ligne de A. C'est exactement ce que fait le programme linéaire, pour les cas où ${b_i}'$ peut être différent de b_i, de façon optimale.

Question #5.7

(Not so easy) We say that b and b' are equivalent, denoted by $b \sim b'$, if they are such that $b \preceq b'$ et $b' \preceq b$. We suppose that the existence of a canonical representation for all equivalence classes defined par $\sim$. How can you compute $can(b)$ using linear programming ?

Solution: pour chaque ligne (A_{i*}) de A on calcule $LP(A, b, (A_{i*}))$ qui donne potentiellement un $b'_i \leq b_i$, la $i$ème coordonnée de $can(b)$.

Question #5.8

Give an intuition of what is the interpretation of can in the case of the matric A_2.

Solution: le calcul de chemin de poids minimal.

Question #5.9

How can you decide if b is lesser or equal to b' in D_A ?

Solution: $can(b) \leq can(b')$, composante par composante.

In the sequel we want to define an abstraction $\alpha : \mathcal{P}(\mathbb{Z}) \rightarrow D_A$. Let us take a finite set of points of $\mathbb{R}^n$ for which we can to compute one abstraction in D_A, and its convex hull as a polyhedra P defined by constraints $Cx + d \geq 0$, where C is a matrix $k \times n$, and d a vector of k reals.

We recall the Farkas' Lemma. Let C be a matrix $k \times n$, d a vector of reals of size k, e un vector of reals of size n and f a real.
The set of affine inequalities $Cx + d \geq 0$ implies the single affine inequality $e^T x + f \geq 0$ iff there exists a vector of k positive real numbers $\lambda = (\lambda_i \geq 0)_{i=1,\ldots,k}$ such that $C^T \lambda = e$ and $d^T \lambda \leq f$. Moreover $Cx + d \geq 0$ is insatisfiable (has no solutions) if and only if there exists $\lambda \geq 0$ such that $C^T \lambda = 0$ and $d^T \lambda \leq -1$.

Question #5.10

Now give a way to compute the image of the set of points trying to remain as precise as possible, with the help of the Farkas' Lemma.

Solution: On considère un ensemble fini de points de $\mathbb{R}^n$ et son enveloppe convexe, définie par les contraintes $Cx + d \geq 0$. On veut trouver des constantes b_i, $i = 1, \ldots, m$ telles que pour tous les $(x_1, \ldots, x_n)$ tels que $Cx + d \geq 0$, $A_i x + b_i \geq 0$ et b_i minimal (où A_i est la ligne i de la matrice A). Par le lemme de Farkas, cela revient à calculer b_i comme le minimum de $d^T \lambda$ sous la contrainte $C^T \lambda = A_i$, $\lambda \geq 0$.

Question #5.11

Define the union of two canonical abstract elements ($b = can(b)$ and $b' = can(b')$) of D_A as functions of their components b_i and ${b_i}'$.

Solution: C'est $(min(b_i, {b_i}'))_{i=1,\ldots,m}$.

Question #5.12

Same for intersection.

Solution: C'est $can(max(b_i, {b_i}'))_{i=1,\ldots,m}$.

CAP Companion Sheet

Mini-while, Mini-ML (abstract syntax)

Mini-while:
Boolean expr:

$$\begin{array}{rcll} b & ::= & true & \textit{constant} \\ & | & false & \textit{constant} \\ & | & b \text{ or } b & \textit{or} \\ & | & b \text{ and } b & \textit{and} \\ & | & \ldots & \end{array}$$

Numerical expressions:

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

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

Mini-ML

$$\begin{array}{rcll} e & ::= & x & \text{identificator} \\ & | & c & \text{constant } (1, 2, \ldots, true, \ldots) \\ & | & op & \text{primitive } (+, \times, fst, \ldots) \\ & | & \texttt{fun } x \to e & \text{function} \\ & | & e\ e & \text{application} \\ & | & (e, e) & \text{pair} \\ & | & \texttt{let } x = e \texttt{ in } e & \text{local binding} \end{array}$$

Grammars-Visitors

Attributes An attribute is a mapping from elements of a given grammar to "an information". Defining a grammar attribution consists in giving a name and a type to the attributes you give to terminal/non terminal symbols of the grammar, and a recursive way to compute them from the grammar rules.

Python-ANTLR syntax

- Access to a given non-terminal `name`: `ctx.name()`, if more than one: `ctx.name(0)`, `ctx.name(1)`...
- Recursive calls to children: `self.visit(child)`
- Parsed chain of a terminal: `xx.getText()`.

Typing, static semantics

For mini-while We add declarations for the language:

$$D(decl) \quad ::= \quad var\ x : t \quad \text{type declaration}$$

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

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

$$\frac{D_1 \to_d \Gamma_1 \quad D_2 \to_d \Gamma_2 \quad Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset}{D1; D_2 \to_d \Gamma_1 \cup \Gamma_2}$$

Then a typing judgment for expressions is $\Gamma \vdash e : \tau \in Basetype$. For statements we invent the `void` type.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while } b \texttt{ do } S \texttt{ done} : \texttt{void}} \qquad \frac{D \to \Gamma \quad \Gamma \vdash C : \texttt{void}}{\emptyset \vdash DC : \texttt{void}}$$

Monomorphic typing of mini-ML

$$\frac{x \in Dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \to e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{}{\Gamma \vdash n : \texttt{int}}$$

$$\frac{}{\Gamma \vdash + : \texttt{int} \times \texttt{int} \to \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2}$$

Operational semantics

Natural semantics (big step) for mini-while : $(Stm, State) \to State$ where $State = Var \to \mathbb{Z}$

$$(x := a, \sigma) \to \sigma[x \mapsto \mathcal{A}[a]\sigma]$$

$$(\texttt{skip}, \sigma) \to \sigma$$

$$\frac{(S_1, \sigma) \to \sigma' \ (S_2, \sigma') \to \sigma''}{((S_1; S_2), \sigma) \to \sigma''}$$

$$\text{if } \mathcal{B}[b]\sigma = tt : \frac{(S_1, \sigma) \to \sigma'}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \to \sigma'}$$

$$\text{if } \mathcal{B}[b]\sigma = ff : \frac{(S_2, \sigma) \to \sigma'}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \to \sigma'}$$

$$\text{if } \mathcal{B}[b]\sigma = tt : \frac{(S, \sigma) \to \sigma', (\texttt{while } b \texttt{ do } S, \sigma') \to \sigma''}{(\texttt{while } b \texttt{ do } S, \sigma) \to \sigma''}$$

$$\text{if } \mathcal{B}[b]\sigma = ff : (\texttt{while } b \texttt{ do } S, \sigma) \to \sigma$$

(denot sem.) Arithmetic:
$\mathcal{A} : expr \mapsto State \to \mathbb{Z}$

$$\mathcal{A}[v]\sigma = value(v)$$

$$\mathcal{A}[x]\sigma = \sigma(x)$$

$$\mathcal{A}[e_1 + e_2]\sigma = \mathcal{A}[e_1] + \mathcal{A}[e_2].$$

(denot. sem.) Boolean:

$\mathcal{B} : bexpr \mapsto State \to \mathbb{B}$

Reduction semantics for Mini-ML (call-by-value) $e \xrightarrow{v} v$ where e an expression and v a value following the following abstract syntax:

$$\begin{array}{rcll} v & ::= & c & \text{constant} \\ & | & op & \text{primitive} \\ & | & \texttt{fun } x \to e & \text{function} \\ & | & (v, v) & \text{pair} \end{array}$$

$$\frac{}{c \xrightarrow{v} c} \qquad \frac{}{op \xrightarrow{v} op} \qquad \frac{}{(\texttt{fun } x \to e) \xrightarrow{v} (\texttt{fun } x \to e)}$$

$$\frac{e_1 \xrightarrow{v} v_1 \quad e_2 \xrightarrow{v} v_2}{(e_1, e_2) \xrightarrow{v} (v_1, v_2)}$$

$$\frac{e_1 \xrightarrow{v} v_1 \quad e_2[x \leftarrow v_1] \xrightarrow{v} v}{\texttt{let } x = e_1 \texttt{ in } e_2 \xrightarrow{v} v}$$

$$\frac{e_1 \xrightarrow{v} (\texttt{fun } x \to e) \quad e_2 \xrightarrow{v} v_2 \quad e[x \leftarrow v_2] \xrightarrow{v} v}{e_1\ e_2 \xrightarrow{v} v}$$

$$\frac{e_1 \xrightarrow{v} + \quad e_2 \xrightarrow{v} (n_1, n_2) \quad n = n_1 + n_2}{e_1\ e_2 \xrightarrow{v} n}$$

$$\frac{e_1 \xrightarrow{v} fst \quad e_2 \xrightarrow{v} (v_1, v_2)}{e_1\ e_2 \xrightarrow{v} v_1}$$

Dataflow Analysis (liveness)

- 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.

TARGET18 ISA

Arithmetical and logical instructions have 2 or 3 operands: (also sub, xor, ...)

add3i r1 r0 3 *; r1 <− r0+3*
add2i r1 15 *; r1 <− r1+15*
add3 r1 r2 r3 *; r1 <− r2+r3*
add2 r1 r2 *; r1 <− r1+r2*

Conditional jumps can be made with:

loop:
sub2i r0 1 *; last written register r0*
jumpif nz loop *; if r0 different from 0, jump*

or

loop:
cmpi r0 0 *; compare r0 to 0*
jumpif neq loop *; if r0 different from 0, jump*

(*nz* is non zero, *neq* not equal, you also have $sle = \leq$ and $lt = <$)
Read from memory ($r_2 \leftarrow Mem[r_1]$):

setctr a0 r1 *; r1 is supposed to contain an address*
readse a0 xxx r2 *; xxx the number of bits to read*

Write to memory : same with `writese`.
Other useful instruction: `lea` (load effective address from a label).

3 address code generation (smt only)

`newTemp` : $() \to \mathbb{N}$ and `newLabel` : $() \to \mathbb{N}$.

x = e	`dr <- GenCodeExpr(e)`
	`#a code to compute e has been generated`
	`find loc the location for var x`
	`code.add(instructionLET(loc,dr))`

S1; S2	`#concat codes`
	`GenCodeSmt(S1)`
	`GenCodeSmt(S2)`

`if` b `then` $S1$ `else` $S2$	`lelse,lendif <-newLabels()`
	`t1 <- GenCodeExpr(b)`
	`#if the condition is false, jump to else`
	`code.add(InstructionCondJUMP(lelse, t1, "eq", 0))`
	`GenCodeSmt(S1) #then`
	`code.add(InstructionJUMP(lendif))`
	`code.addLabel(lelse)`
	`GenCodeSmt(S2) #else`
	`code.addLabel(lendif)`

`while` b `do` S `done`	`ltest,lendwhile <-newLabels()`
	`code.addLabel(ltest)`
	`t1 <- GenCodeExpr(b)`
	`code.add(InstructionCondJUMP(lendwhile, t1, "eq", 0))`
	`GenCodeSmt(S)                    #execute S`
	`code.add(InstructionJUMP(ltest))   #and jump to the test`
	`code.addLabel(lendwhile)           #else it is done.`