Written exam
MPRI 2-6, year 2022–2023

28 November 2022

Duration: 2 hours

All written documents (printed course slides, personal notes, books, etc.) are allowed.
The use of connected electronic devices (computers, phones, tablets, smartwatches, buds, etc.) is not allowed.
The questions are written in English. You can answer either in English or in French, to your preference.
It will not be answered any question during the exam. In case of an ambiguity or an error in the definitions or the questions, it is part of the exam to correct them and answer to the best of your abilities.
In case you cannot answer a question, it is often possible to assume its results and continue with the next questions.
Questions marked with $(*)$ are more difficult.

Analysis of dynamic programs with *eval*

This exam studies the static analysis of dynamic code, that is, programs where the whole source code is not available statically at the begining of the analysis; some of the code is discovered dynamically. We study a common case where the program builds strings that are then interpreted at run-time as code, and executed. This is implemented, for instance, as the *eval* statement in JavaScript, or the *exec* statement in Python.

We consider the following language based on the numeric language used in the course, extended with strings manipulation and eval:

$$
\begin{array}{rcll}
\textit{Stmt} & ::= & {}^{(\ell_i)}\ x \leftarrow \textit{NExp};\ {}^{(\ell_x)} & \textit{(numeric assignment into } x \in \mathcal{N}\textit{)} \\
& | & {}^{(\ell_i)}\ s \leftarrow \textit{SExp};\ {}^{(\ell_x)} & \textit{(string assignment into } s \in \mathcal{S}\textit{)} \\
& | & {}^{(\ell_i)}\ \textbf{if}\ (\textit{BExp})\ \textbf{then}\ \{\ {}^{(\ell_1)}\ \textit{Stmt}\ \};\ {}^{(\ell_x)} & \textit{(conditional)} \\
& | & {}^{(\ell_i)}\ \textbf{while}\ (\textit{BExp})\ \textbf{do}\ \{\ {}^{(\ell_1)}\ \textit{Stmt}\ \};\ {}^{(\ell_x)} & \textit{(loop)} \\
& | & {}^{(\ell_i)}\ \textit{Stmt}\ {}^{(\ell_1)}\ \textit{Stmt}\ {}^{(\ell_x)} & \textit{(sequence)} \\
& | & {}^{(\ell_i)}\ \textbf{eval}(\textit{SExp});\ {}^{(\ell_x)} & \textit{(execute string as code)}
\end{array}
$$

We assume a finite, fixed set $\mathcal{N}$ of integer-valued variables and a finite, fixed set $\mathcal{S}$ of string-valued variables. We denote by Σ the set of characters. Strings are sequences of characters of arbitrary length in Σ^*, and $\epsilon \in \Sigma^*$ denotes the empty string. Statements include: numeric and string assignments, as well as the conditional **if** $(\cdot)$ **then** $\{\cdot\}$, the loop **while** $(\cdot)$ **do** $\{\cdot\}$, the sequence and, finally, **eval**. As in the course, statements are labelled with a control point before ℓ_i and a control point after ℓ_x and, in some cases, an intermediate label ℓ_1.
The language of expressions is as follows:

$$
\begin{array}{rcll}
\textit{NExp} & ::= & x & \textit{(variable } x \in \mathcal{N}\textit{)} \\
& | & i & \textit{(constant } i \in \mathbb{Z}\textit{)} \\
& | & ? & \textit{(random integer)} \\
& | & \textit{NExp} \circ \textit{NExp} & \textit{(operator } \circ \in \{+,-,\ldots\}\textit{)}
\end{array}
\qquad
\begin{array}{rcll}
\textit{SExp} & ::= & s & \textit{(variable } s \in \mathcal{S}\textit{)} \\
& | & \text{“}\sigma\text{”} & \textit{(constant } \sigma \in \Sigma^*\textit{)} \\
& | & ? & \textit{(random string)} \\
& | & \textit{SExp} \cdot \textit{SExp} & \textit{(concatenation)}
\end{array}
$$

$$
\begin{array}{rcll}
\textit{BExp} & ::= & \textit{NExp} \circ \textit{NExp} & \textit{(numeric comparison } \circ \in \{\leq,<,=,\ldots\}\textit{)} \\
& | & \textit{SExp} \circ \textit{SExp} & \textit{(string comparison } \circ \in \{=,\neq\}\textit{)} \\
& | & \textbf{startsWith}(\textit{SExp},\textit{SExp}) & \textit{(string prefix test)} \\
& | & ? & \textit{(random Boolean)} \\
& | & \neg \textit{BExp} & \textit{(Boolean negation)}
\end{array}
$$

Numeric expressions *NExp* use the classic arithmetic operators, constants, variables, and the random choice “?” in $\mathbb{Z}$. String expressions *SExp* also use variables, constants “σ”, and the random choice “?” in Σ^*, and feature a single operator: string concatenation “$\cdot$”. Boolean expressions *BExp* allow comparing two numbers as well as two strings. $\textbf{startsWith}(\sigma,\sigma')$ is true if and only if σ' is a prefix of σ, i.e., $\exists \sigma'' \in \Sigma^* : \sigma = \sigma' \cdot \sigma''$. “?” is the non-deterministic choice (random Boolean) and $\neg$ is the Boolean negation.

Informally, **eval** accepts a string expression, evaluates it into a string value, interprets the string as a statement in the syntax of *Stmt*, and executes the statement immediately. Here are some examples programs using **eval**:

$$P_1 \stackrel{\text{def}}{=} s \leftarrow \text{“}x\text{”};\ t \leftarrow \text{“12”};\ \textbf{eval}(s \cdot \text{“} \leftarrow \text{”} \cdot t \cdot \text{“};\text{”});$$

$$P_2 \stackrel{\text{def}}{=} s \leftarrow \text{“100”};\ \textbf{if}\ (?)\ \textbf{then}\ \{\ s \leftarrow \text{“200”};\ \};\ \textbf{eval}(\text{“while}\ \ (x < \text{”} \cdot s \cdot \text{“)}\ \ \{y \leftarrow y+1;\ x \leftarrow x+1;\ \};\text{”});$$

$$P_3 \stackrel{\text{def}}{=} s \leftarrow \epsilon;\ \textbf{while}\ (?)\ \textbf{do}\ \{\ s \leftarrow s \cdot \text{“}x \leftarrow x \times 2;\text{”};\ \};\ x \leftarrow 1;\ \textbf{eval}(s);$$

$$P_4 \stackrel{\text{def}}{=} s \leftarrow \text{“?”};\ \textbf{eval}(\text{“eval(\textbackslash“}x \leftarrow \text{”} \cdot s \cdot \text{“};\text{\textbackslash”);”});$$

$$P_5 \stackrel{\text{def}}{=} s \leftarrow \text{“eval(}s\text{);”};\ \textbf{eval}(s);$$

P_1 constructs the assignment $x \leftarrow 12$ and executes it; at the end of P_1, x equals 12. P_2 shows that an **eval** statement can be called with different strings depending on a non-deterministic choice. It can also execute complex loops: in P_2, incrementing x and y until x equals 100 or 200. P_3 shows an example constructing a string with an arbitrary number of assignments, computing some power of two. P_4 shows that the string evaluated by **eval** can, itself, contain an **eval** statement. It assigns a random value to x. Finally, P_5 loops executing non-terminating recursive calls to **eval**.

The first part of the exam studies the concrete semantics, while the second part studies string abstractions. These parts are independent, and come together to build the final analysis in part three.

1 Concrete Semantics

We denote by $\mathcal{L}$ an (infinite) set of labels usable by all programs. A control-flow graph $(e, x, A) \in \mathcal{CFG} \stackrel{\text{def}}{=} \mathcal{L} \times \mathcal{L} \times \mathcal{P}(\mathcal{L} \times \mathit{Cmd} \times \mathcal{L})$ is defined by an entry-point $e \in \mathcal{L}$, an exit point $x \in \mathcal{L}$, and a set $A \subseteq \mathcal{L} \times \mathit{Cmd} \times \mathcal{L}$ of arcs decorated by commands in the following language, including assignments, filtering by Boolean expressions, and eval:

$$\mathit{Cmd} ::= x \leftarrow \mathit{NExp} \mid s \leftarrow \mathit{SExp} \mid \mathit{BExp} \mid \textbf{eval}(\mathit{SExp})$$

Any statement $P \in \mathit{Stmt}$ can be converted into a control-flow graph.

Question 1.1.
Define the conversion function $CFG : \mathit{Stmt} \to \mathcal{CFG}$ from statements to control-flow graphs. It is defined by induction on the structure of statements (similarly to the way we derived transition relations or equation systems in the course).

A concrete state $(\rho, \mu) \in \mathcal{E} \stackrel{\text{def}}{=} (\mathcal{N} \to \mathbb{Z}) \times (\mathcal{S} \to \Sigma^*)$ is a pair of functions, assigning a numeric value to each numeric variable in $\mathcal{N}$ and a string value to each string variable in $\mathcal{S}$.

We first consider, in the following two questions, programs *without* **eval**.

Question 1.2.
Give the definition of the concrete evaluation $\mathsf{S}[\![\, \mathit{SExp} \,]\!] : \mathcal{E} \to \mathcal{P}(\Sigma^*)$ for string expressions: it returns the set of values an expression can evaluate to in a concrete environment. We assume that $\mathsf{N}[\![\, \mathit{NExp} \,]\!] : \mathcal{E} \to \mathcal{P}(\mathbb{Z})$ is similarly defined for numeric expressions (it is not asked as part of the exam).
*Give the concrete semantics $\mathsf{C}[\![\, \mathit{Cmd} \,]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ of (non-**eval**) commands that, given a set of states before the command, returns the states reachable after the command.*

Given a control-flow-graph (e, x, A) without **eval** and a set $I \subseteq \mathcal{E}$ of environments at its entry point e, we denote as $\mathsf{G}[\![\, (e, x, A) \,]\!]\, I : \mathcal{L} \to \mathcal{P}(\mathcal{E})$ its semantics, that gives the reachable environments at each label.

Question 1.3.
Give the definition of $\mathsf{G}[\![\, (e, x, A) \,]\!]\, I$ as a fixpoint. Justify that the fixpoint indeed exists.

We now add back **eval** into our language. It remains to provide the definition of $\mathsf{C}[\![\, \textbf{eval}(\mathit{SExp}) \,]\!]$ to complete our concrete semantics. We assume given a function $\mathit{parse} : \Sigma^* \to \mathit{Stmt} \cup \{\omega\}$ that, given a string, outputs the statement it represents, or ω if it is not the string representation of any statement in the language (i.e., ω denotes a parsing error, which halts the program). All the labels used in $\mathit{parse}(\sigma)$ are fresh labels (i.e., previously unused) in $\mathcal{L}$.

Question 1.4.(∗)
Express the semantics $\mathsf{C}[\![\, \mathbf{eval}(\mathit{SExp})\,]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ *of* **eval**, *using* $\mathsf{S}[\![\, \mathit{SExp}\,]\!]$, *parse, and* $\mathsf{G}[\![\ ]\!]$.
Prove that $\mathsf{G}[\![\ ]\!]$ *and* $\mathsf{C}[\![\ ]\!]$ *remain well-defined, despite their cyclic dependencies, and the fact that a program can trigger unbounded nested calls to* **eval** *(example* P_5*). You can use use a fixpoint for this.*
Compute the concrete semantics of examples P_1*,* P_4*, and* P_5 *using your definitions; detail each step carefully.*

2 String Abstractions

We construct several domains to abstract strings. We start with non-relational domains of the form $\mathcal{D}^\sharp \stackrel{\text{def}}{=} \mathcal{S} \to \mathcal{B}^\sharp$ for different abstractions $\mathcal{B}^\sharp$ of sets of strings $\mathcal{P}(\Sigma^*)$. Our first domain is a variation on the constant domain, which maintains up to k different strings (where k is a fixed parameter), or reverts to $\top$ to represent all possible strings:

$$\mathcal{B}^\sharp_{\leq k} \stackrel{\text{def}}{=} \{\, S \subseteq \Sigma^* \mid |S| \leq k \,\} \cup \{\top\}$$

Question 2.1.
Give a partial order on $\mathcal{B}^\sharp_{\leq k}$.
Provide a Galois connection between $\mathcal{P}(\Sigma^*)$ *and* $\mathcal{B}^\sharp_{\leq k}$*, and prove that it is indeed a Galois connection.*
Provide optimal abstract versions for string operators on $\mathcal{B}^\sharp_{\leq k}$*, including:* $\cup$*,* $\cap$*, string constants, random ?, concatenation* $\cdot$*, equality and disequality testing, and* **startsWith***. Prove the optimality.*
State whether each operator is exact or not, providing a proof of your claim.

We now lift the restriction on the size of sets to get the following abstraction of $\mathcal{P}(\Sigma^*)$:

$$\mathcal{B}^\sharp_{f} \stackrel{\text{def}}{=} \{\, S \subseteq \Sigma^* \mid |S| \text{ is finite } \} \cup \{\top\}$$

Question 2.2.
Give a partial order on $\mathcal{B}^\sharp_{f}$*, and prove that there is no Galois connection between* $\mathcal{B}^\sharp_{f}$ *and* $\mathcal{P}(\Sigma^*)$.
Discuss how the operators from $\mathcal{B}^\sharp_{\leq k}$ *can be adapted to this new domain.*
Discuss the need for a widening, and propose one if needed.

We then propose an abstraction $\mathcal{B}^\sharp_{p}$ that abstracts a set of strings as their longest common prefix (possibly ϵ).

Question 2.3.
Provide a partial order on $\mathcal{B}^\sharp_{p}$ *and a Galois connection between* $\mathcal{P}(\Sigma^*)$ *and* $\mathcal{B}^\sharp_{p}$ *(provide a proof).*
Propose abstract versions of $\cup$*,* $\cap$*, concatenation, and* **startsWith***; discuss their optimality and exactness.*
Discuss the need for a widening and a narrowing; propose one if needed.

Our last domain is a *relational* version $\mathcal{D}^\sharp_r \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{S} \times \mathcal{S}) \cup \{\bot\}$ of the prefix string abstraction. A set of string environments is abstracted as either $\bot$, or a relation $r^\sharp \in \mathcal{P}(\mathcal{S} \times \mathcal{S})$ where $(s,t) \in r^\sharp$ means that, in each environment, the value of variable s is a prefix of that of t.

Question 2.4.(∗)
Propose a Galois connection between $\mathcal{P}(\mathcal{S} \to \Sigma^*)$ *and* $\mathcal{D}^\sharp_r$.
Propose sound abstractions of $\cup$*,* $\cap$*,* $\mathsf{C}[\![\, s \leftarrow t \cdot u\,]\!]$*, and* $\mathsf{C}[\![\, s = t\,]\!]$.
Show that applying a transitive closure to $r^\sharp \in \mathcal{P}(\mathcal{S} \times \mathcal{S})$ *is sound, and important to ensure the precision of the abstract* $\cup$ *(an example is sufficient, no proof of optimality is required).*

Question 2.5.
Propose a reduced product between the constant string domain $\mathcal{S} \to \mathcal{B}^\sharp_{\leq k}$ *and the relational prefix domain* $\mathcal{D}^\sharp_r$.

String domains used in practice are generally non-relational domains that abstract sets of strings using regular expressions and finite state automata. However, we do not consider their construction in this exam as it is quite complex.

3 Abstract Analysis with eval

We assume that we are given an abstract domain $\mathcal{D}^\sharp$ for $\mathcal{P}(\mathcal{E})$, and the corresponding sound abstractions $\mathsf{C}[\![\, x \leftarrow \mathit{NExp} \,]\!]^\sharp$, $\mathsf{C}[\![\, s \leftarrow \mathit{SExp} \,]\!]^\sharp$, $\mathsf{C}[\![\, \mathit{BExp} \,]\!]^\sharp$, and $\cup^\sharp$, as well as a widening $\triangledown$ (for instance, $\mathcal{D}^\sharp$ can combine a numeric domain, such as intervals, with a string domain from Part 2).

Additionally, we need to extract the set of strings an expression evaluates to in an abstract environment. We use *regular expression* as a way to convey this information independently from the choice of domain. Thus, we assume the existence of an abstract evaluation $\mathsf{S}[\![\, \mathit{SExp} \,]\!]^\sharp : \mathcal{D}^\sharp \to \mathit{re}$, where regular expressions re are defined classically as:

$$\mathit{re} ::= \sigma \in \Sigma^* \mid (\mathit{re}|\mathit{re}) \mid \mathit{re} \cdot \mathit{re} \mid \mathit{re}*$$

using constant strings "σ", choice "$|$", concatenation "$\cdot$", and repetition "$*$".

We first construct a function $\mathit{parse}^\sharp : \mathit{re} \to \mathit{Stmt}$ that returns a statement over-approximating the possible statements denoted by a regular expression. We observe that *parse* (from Part 1) can handle constant strings, while "$|$" and "$*$" can be modeled using, respectively, **if** $(\cdot)$ **then** $\{\cdot\}$ and **while** $(\cdot)$ **do** $\{\cdot\}$ statements. Observe also that, as a sound fallback, we can always revert to a non-deterministic program that exhibits all possible behaviors.

Question 3.1.
Propose a definition of $\mathit{parse}^\sharp$ by induction on the structure of regular expressions, trying to be as precise as possible (you don't need to specify the labels). In case you use a program exhibiting all behaviors, give its definition.
Illustrate your function on examples P_1, P_2, and P_3: give a regular expression modeling the (concrete) arguments of each **eval***, and the result of your $\mathit{parse}^\sharp$ function on these regular expressions.*

We assume given an iterator algorithm $\mathsf{G}[\![\, (e,x,A) \,]\!]^\sharp X^\sharp : \mathcal{L} \to \mathcal{D}^\sharp$ that implements a static analysis, starting in abstract state $X^\sharp \in \mathcal{D}^\sharp$ at entry point e and calling abstract functions $\mathsf{C}[\![\, \mathit{Cmd} \,]\!]^\sharp$ on arcs in A until stabilization with widening (such iterators were discussed in the course).
It only remains to provide a sound abstraction $\mathsf{C}[\![\, \mathbf{eval}(\mathit{SExp}) \,]\!]^\sharp$ of **eval** to complete our analysis.

Question 3.2.$(*)$
Propose an abstraction $\mathsf{C}[\![\, \mathbf{eval}(\mathit{SExp}) \,]\!]^\sharp$ using the functions $\mathsf{S}[\![\, \mathit{SExp} \,]\!]^\sharp$, $\mathit{parse}^\sharp$, and $\mathsf{G}[\![\, (e,x,A) \,]\!]^\sharp$ above.
Prove the soundness of your definition, and use it to prove the soundness of the whole analysis of programs with eval.
Propose a solution to ensure that the analysis always terminates with a sound result, even in the case of unbounded nested calls to **eval***. Provide a proof that your solution indeed ensures termination.*

Question 3.3.
Demonstrate your analysis on examples P_1, P_2, and P_5 using the interval domain for numeric variables and the constant string domain $\mathcal{B}^\sharp_{\leq 2}$ from Part 2 for string variables.

Notes. This exam is inspired from the article:

- Vincenzo Arceri and Isabella Mastroeni. Analyzing Dynamic Code: A Sound Abstract Interpreter for evil eval. In ACM Transactions on Privacy and Security, vol. 24, issue 2, May 2021.

See also:

- Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the Eval that Men Do. In ISSTA '12, July 15-20, 2012.
- Vincenzo Arceri, Isabella Mastroeni, and Sunyi Xu. Static Analysis for ECMAScript String Manipulation Programs. In Appl. Sci. 2020, 10(10), 3525.