École Polytechnique

INF564 : Compilation

examen 2024 (X2021)

Jean-Christophe Filliâtre

18 mars 2024 — 14h00–17h00

The test lasts 3 hours. Handwritten or printed course notes are the only documents allowed. The questions are independent, in the sense that it is not necessary to have answered the previous questions in order to deal with a question. On the other hand, questions can call on definitions or results introduced in previous questions. Unless explicitly stated otherwise, all answers must be justified.
Feel free to answer in French or English.
Figures 2–3 are grouped together at the end of the subject on page 7. Suggestion: detach the last sheet.

Throughout this subject, we consider a small fragment of the Python language (different from that of the project), whose abstract syntax is given in figure 2. This fragment includes the value `None`, Booleans (`False` and `True`), integers and "lists". Expressions are limited to constants (c), variables (x), primitive operations (op), function calls (f) and a conditional expression. The latter, e_1 `if` e_2 `else` e_3, evaluates e_2 then returns the value of e_1 if e_2 is true and the value of e_3 otherwise. A program is a sequence of function definitions (d), followed by the evaluation of a single expression whose value is displayed with `print`. The body of a function is a sequence of assignments $x = e$, followed by a `return` instruction. Each function can refer to previously defined functions or to the function currently being defined (recursive function). Here's an example of a program in this fragment, written in concrete Python syntax, which calculates and displays a list of five integers:

```
def aux(s):
    a = s[0]
    b = s[1]
    return [a+b] + s
def myst(n, s):
    return s if n==0 else myst(n-1, aux(s))
print(myst(3, [1]+[0]))
```

A big-step operational semantic is shown in figure 3. It expresses call-by-value. A value is denoted v. The value of variables is given by an environment V, i.e. a function that maps variable names to values. The semantics includes the definition of primitive operations (in the table at the bottom of the figure). For each operation op, its semantics is given by a function $[\![op]\!]$ defined on values. This function can be partial, i.e. not defined over all values. For example, the expression `1+None` has no value. The *num* operation is used to interpret a Boolean as an integer in certain operations.

Question 1 What is the list calculated and displayed by the above program? (No justification is required.)

Question 2 Implement a function `rev` that, when applied to a list $[v_0, v_1, \ldots, v_{n-1}]$, returns the list $[v_{n-1}, \ldots, v_1, v_0]$, with $n \geq 0$. This function must be able to work with $n = 0$, even though it is not possible to construct an empty list in our fragment. You can introduce an auxiliary function.

Question 3 For each of the following expressions, give the derivation of its evaluation in an empty environment, when it exists, or justify that there is none. (Here we take the context of the two functions `aux` and `myst` given as examples at the beginning of the subject.)

- `myst(0, [1]+[0])`
- `aux([0])`
- `myst(-1, [1]+[0])`

Question 4 We propose to program an interpreter of our language, in Java or in OCaml (your choice), following the big-step semantics of figure 3. Specify the Java/OCaml types and data structures used. Give the type of each Java/OCaml function involved in the interpreter. *You are not asked to write the code for these functions.* Recall that in Java (resp. OCaml) we obtain dictionaries whose keys are strings with `HashMap<String, ...>` (resp. `module StrMap = Map.Make(String)`).

Static Typing. Although Python is a dynamically-typed language, we propose here to perform a little static typing on our language, with the dual aim of rejecting inconsistent programs and executing certain programs more efficiently. We give ourselves four kinds `none`, `bool`, `int`, and `list`, to represent respectively a `None` value, a Boolean value, an integer value or a list. A type τ is then a set of kinds, i.e.

$$\tau \subseteq \{\texttt{none}, \texttt{bool}, \texttt{int}, \texttt{list}\},$$

with the following interpretation: if an expression of type τ evaluates to a value, then this value will necessarily be of one of the kinds of τ. In particular, a type can be the empty set $\emptyset$ (the expression cannot have a value) or the set $\{\texttt{none}, \texttt{bool}, \texttt{int}, \texttt{list}\}$ (the value can be any).

To type an expression, we give ourselves a context made up of two environments: an environment Γ giving the type of the variables (a function from variables to types) and an environment Δ giving function types. (Primitive operations are seen as functions for typing.) The type of a function, noted σ, is of the form

$$\tau_0 \times \cdots \times \tau_{n-1} \to \tau$$

where n is the number of function parameters. The environment Δ is a set of pairs (f, σ) where f is the name of a function and σ is a function type. For a single function, there may be several different types in the Δ environment. The typing judgment for an expression is noted $\Delta, \Gamma \vdash e : \tau$. Figure 1 gives typing rules for expressions, as well as an environment Δ_{op} giving the types of primitive operations. Note that the *add* operation has two different types in Δ_{op}, which is consistent with its two interpretations in figure 3.

To type an application $f(e_0, \ldots, e_{n-1})$, we make the union of all the types that can be obtained with the types of f given by Δ and compatible with the types τ_i of the actual parameters e_i. In particular, the result may be the empty type $\emptyset$ if Δ contains no compatible function type.

Question 5 In an empty environment Γ and an environment Δ containing only primitive operations, give

1. an expression of type $\emptyset$;
2. an expression of type $\{\texttt{bool}, \texttt{int}, \texttt{list}\}$.

$$\frac{}{\Delta, \Gamma \vdash \texttt{None} : \texttt{none}} \qquad \frac{}{\Delta, \Gamma \vdash b : \texttt{bool}} \qquad \frac{}{\Delta, \Gamma \vdash n : \texttt{int}} \qquad \frac{x \in \mathrm{dom}(\Gamma)}{\Delta, \Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Delta, \Gamma \vdash e_1 : \tau_1 \quad \Delta, \Gamma \vdash e_2 : \tau_2 \quad \Delta, \Gamma \vdash e_3 : \tau_3}{\Delta, \Gamma \vdash e_1 \texttt{ if } e_2 \texttt{ else } e_3 : \tau_1 \cup \tau_3}$$

$$\frac{\forall\ 0 \leq i < n,\ \Delta, \Gamma \vdash e_i : \tau_i}{\Delta, \Gamma \vdash f(e_0, \ldots, e_{n-1}) : \Delta(f)(\tau_0, \ldots, \tau_{n-1})} \quad \text{with } \Delta(f)(\tau_0, \ldots, \tau_{n-1}) \stackrel{\text{def}}{=} \bigcup_{\substack{f : \tau'_0 \times \cdots \tau'_{n-1} \to \tau \in \Delta \\ \forall 0 \leq i < n,\ \tau_i \cap \tau'_i \neq \emptyset}} \tau$$

Environment Δ_{op} :

operator	type
add	$\{\texttt{list}\} \times \{\texttt{list}\} \to \{\texttt{list}\}$
add	$\{\texttt{bool}, \texttt{int}\} \times \{\texttt{bool}, \texttt{int}\} \to \{\texttt{int}\}$
sub	$\{\texttt{bool}, \texttt{int}\} \times \{\texttt{bool}, \texttt{int}\} \to \{\texttt{int}\}$
len	$\{\texttt{list}\} \to \{\texttt{int}\}$
mk	$\{\texttt{none}, \texttt{bool}, \texttt{int}, \texttt{list}\} \to \{\texttt{list}\}$
get	$\{\texttt{list}\} \times \{\texttt{bool}, \texttt{int}\} \to \{\texttt{none}, \texttt{bool}, \texttt{int}, \texttt{list}\}$
eq	$\{\texttt{none}, \texttt{bool}, \texttt{int}, \texttt{list}\} \times \{\texttt{none}, \texttt{bool}, \texttt{int}, \texttt{list}\} \to \{\texttt{bool}\}$

Figure 1: Static typing of expressions.

Question 6 For this question, the following environments are used:

$$\Delta \stackrel{\text{def}}{=} \Delta_{op} \cup \{(f, \{\texttt{int}\} \times \{\texttt{list}\} \to \{\texttt{list}\}), (f, \{\texttt{int}\} \times \{\texttt{none}\} \to \{\texttt{none}\})\}$$
$$\Gamma \stackrel{\text{def}}{=} \{x \mapsto \{\texttt{list}\}\}$$

For each of the following expressions, give its typing derivation in Δ, Γ.

1. $f(1, x)$
2. $f(1, x\texttt{[0]})$
3. $f(\texttt{None}, \texttt{None})$

Question 7 What are the conditions over Δ, Γ and e for the existence of a type τ such that $\Delta, \Gamma \vdash e : \tau$?

Question 8 Show that, for an environment Δ, Γ and an expression e, there exists at most one type τ such that $\Delta, \Gamma \vdash e : \tau$.

Question 9 With regard to typing, is there a difference between the following two environments?

$$\Delta = \{(f, \{\texttt{bool}\} \times \{\texttt{bool}\} \to \{\texttt{int}\}); (f, \{\texttt{bool}\} \times \{\texttt{int}\} \to \{\texttt{int}\})\}$$
$$\text{and } \Delta' = \{(f, \{\texttt{bool}\} \times \{\texttt{bool}, \texttt{int}\} \to \{\texttt{int}\})\}$$

Same question with the following two environments:

$$\Delta = \{(f, \{\texttt{bool}\} \times \{\texttt{bool}\} \to \{\texttt{int}\}); (f, \{\texttt{bool}\} \times \{\texttt{int}\} \to \{\texttt{list}\})\}$$
$$\text{and } \Delta' = \{(f, \{\texttt{bool}\} \times \{\texttt{bool}, \texttt{int}\} \to \{\texttt{int}, \texttt{list}\})\}$$

Typing a function. To type a function definition, we introduce the judgment $\Delta \vdash f : \sigma$ which means "in environment Δ, the definition of function f admits the function type σ". We propose the following rule for this judgment:

$$\frac{\begin{array}{c} f(x_0, \ldots, x_{n-1}) \stackrel{\text{def}}{=} x_n = e_n; \ldots; x_{m-1} = e_{m-1}; \texttt{return}\ e \\ \Gamma_n \stackrel{\text{def}}{=} \{x_0 \mapsto \tau_0; \ldots; x_{n-1} \mapsto \tau_{n-1}\} \\ \forall\ n \leq i < m,\ \Delta, \Gamma_i \vdash e_i : \tau_i \quad \Gamma_{i+1} \stackrel{\text{def}}{=} \Gamma_i[x_i \mapsto \tau_i] \\ \Delta, \Gamma_m \vdash e : \tau \end{array}}{\Delta \vdash f : \tau_0 \times \cdots \times \tau_{n-1} \to \tau} \tag{1}$$

(Note its similarity to the semantics rule.) In particular, this rule allows a recursive definition to be typed, by showing $\Delta \vdash f : \sigma$ for an environment Δ containing one or more types for f.

Question 10 For the `aux` function at the beginning of the subject, show that we have

$$\Delta_{op} \vdash \texttt{aux} : \{\texttt{list}\} \to \{\texttt{list}\}.$$

Give the complete typing derivation.

Question 11 Propose at least two different types σ such that, for each, we have

$$\Delta_{op} \cup \{(\texttt{aux} : \{\texttt{list}\} \to \{\texttt{list}\}); (\texttt{myst}, \sigma)\} \vdash \texttt{myst} : \sigma$$

for the `myst` function at the beginning of the subject. (No justification is requested, *i.e.*, we don't ask for the typing derivations, but only for the two types σ.)

Question 12 Propose a type for the function

```
def loop(x):
    return loop(x)
```

that is as informative as possible.

Question 13 We would like to show the type safety property in the following sense: if $V, e \twoheadrightarrow v$ and $\Delta, \Gamma \vdash e : \tau$, then $T(v) \in \tau$ where function T gives the type of a semantic value and is unsurprisingly defined as

$$\begin{array}{rcl} T(\texttt{None}) & = & \texttt{none} \\ T(b) & = & \texttt{bool} \\ T(n) & = & \texttt{int} \\ T([v_0, \ldots, v_{n-1}]) & = & \texttt{list} \end{array}$$

Give necessary conditions on V and Δ, Γ for type safety to be possible. *But we don't ask to show type safety.*

Question 14 Our typing rule for function is not algorithmic: it only allows us to check the definition of a function f with respect to Δ, not to find a type for f. Propose an algorithm to infer a set of types for a function whose definition we have.

Parsing. We wish to perform a syntactic analysis of our small language. A grammar for a subset of the expressions is as follows:

$$\begin{array}{rcl} E & ::= & \texttt{id} \\ & | & E \texttt{ if } E \texttt{ else } E \\ & | & E \texttt{ [} E \texttt{]} \end{array}$$

Question 15 Show that this grammar is ambiguous.

Question 16 This grammar is fed to a tool such as CUP or Menhir, as follows:

```
expr:
| IDENT                    { ... }
| expr IF expr ELSE expr    { ... }
| expr LBR expr RBR         { ... }
```

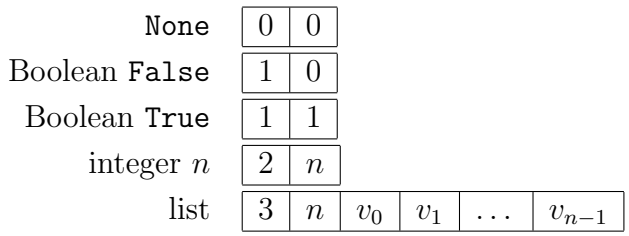
(Semantic actions don't interest us here and are omitted, as are terminal symbol declarations). The CUP or Menhir tool reports two conflicts. What kind of conflicts are they? Propose a solution to resolve these conflicts, by adding suitable priority and/or associativity.

Question 17 Another grammar is proposed for this language:

$$\begin{array}{rcl} E & ::= & A \\ & | & A \texttt{ if } E \texttt{ else } E \\ A & ::= & \texttt{ident} \\ & | & A \texttt{ [} E \texttt{]} \end{array}$$

Show that this grammar is SLR(1). (Reminder: SLR(1) means that there is no conflict anymore in the deterministic LR(0) automaton when reduction of non-terminal symbol X is only performed for lookahead symbols in $\textsc{follow}(X)$.)

Compiling to x86-64 assembly. We propose to compile our little language to x86-64 assembler. (A cheat sheet is given in the appendix.) We assume that the integers in our language are limited to signed 64-bit integers. We adopt the same representation as in the project, where any value is the address of a memory block allocated on the heap, whose size is a multiple of 64 bits, with the following form:



The first word is an integer indicating the kind of value.

Question 18 Discuss this choice of representation. In particular, could `None` be represented by the null pointer? If so, would there be any interest in doing so? And could we represent an integer directly by its value?

Question 19 Explain how the static typing described in the previous questions can be used locally to produce more efficient assembly code. Give examples.

Compiling a small function. We consider the following Python function

```
def mult(x, y, z):
  return z if x==0 else mult(dec(x), y, add(y, z))
```

where `dec` (not shown) implements the function $n \mapsto n - 1$ for an integer n and `add` (not shown) implements the addition of two integers. We assume that function `mult` is given the type

$$\{\texttt{int}\} \times \{\texttt{int}\} \times \{\texttt{int}\} \to \{\texttt{int}\}$$

that is, we only use it with integer parameters and we get a result that is an integer. We intend to compile function `mult` to some optimized x86-64 assembly code.

Question 20 Give some RTL code for function `mult`.

Question 21 Give some optimized assembly code for function `mult`. In particular, if a tail call can be optimized, do it. You are free to proceed the way you want, *i.e.*, you do not have to follow the optimized code generation scheme described in the course. We assume that functions `dec` and `add` are available, and that they follow the x86-64 calling conventions (but you don't have to align the stack before calling them).

$$
\begin{array}{rcll}
e & ::= & c & \textit{constant} \\
& | & x & \textit{variable} \\
& | & op(e,\ldots,e) & \textit{operation} \\
& | & f(e,\ldots,e) & \textit{application} \\
& | & e \texttt{ if } e \texttt{ else } e & \textit{conditional}
\end{array}
\qquad
\begin{array}{rcl}
c & ::= & \texttt{None} \\
& | & b \quad b \in \{\texttt{True},\texttt{False}\} \\
& | & n \quad n \in \mathbb{Z} \\
d & ::= & f(x,\ldots,x) \stackrel{\text{def}}{=} x = e; \ldots; x = e; \texttt{return } e \\
p & ::= & d \ldots d \texttt{ print}(e)
\end{array}
$$

Figure 2: Abstract Syntax.

$$
\begin{array}{rrcl}
\text{value} & v & ::= & c \\
& & | & [v,\ldots,v] \\
\text{environment} & V & ::= & x \mapsto v
\end{array}
\qquad
\begin{array}{rcl}
num(\texttt{False}) & \stackrel{\text{def}}{=} & 0 \\
num(\texttt{True}) & \stackrel{\text{def}}{=} & 1 \\
num(n) & \stackrel{\text{def}}{=} & n
\end{array}
$$

$$
\frac{}{V, c \twoheadrightarrow c}
\qquad
\frac{x \in \mathrm{dom}(V)}{V, x \twoheadrightarrow V(x)}
\qquad
\frac{V, e_2 \twoheadrightarrow v_2 \quad v_2 \notin \{\texttt{None},\texttt{False},0,[]\} \quad V, e_1 \twoheadrightarrow v_1}{V, e_1 \texttt{ if } e_2 \texttt{ else } e_3 \twoheadrightarrow v_1}
$$

$$
\frac{V, e_2 \twoheadrightarrow v_2 \quad v_2 \in \{\texttt{None},\texttt{False},0,[]\} \quad V, e_3 \twoheadrightarrow v_3}{V, e_1 \texttt{ if } e_2 \texttt{ else } e_3 \twoheadrightarrow v_3}
$$

$$
\frac{V, e_i \twoheadrightarrow v_i \quad [\![op]\!](v_0,\ldots,v_{n-1}) = v}{V, op(e_0,\ldots,e_{n-1}) \twoheadrightarrow v}
$$

$$
\frac{\begin{array}{c}
\forall\ 0 \le i < n,\ V, e_i \twoheadrightarrow v_i \\
f(x_0,\ldots,x_{n-1}) \stackrel{\text{def}}{=} x_n = e_n; \ldots; x_{m-1} = e_{m-1}; \texttt{return } e \\
V_n \stackrel{\text{def}}{=} \{x_0 \mapsto v_0; \ldots; x_{n-1} \mapsto v_{n-1}\} \\
\forall\ n \le i < m,\ V_i, e_i \twoheadrightarrow v_i \quad V_{i+1} \stackrel{\text{def}}{=} V_i[x_i \mapsto v_i] \\
V_m, e \twoheadrightarrow v
\end{array}}{V, f(e_0,\ldots,e_{n-1}) \twoheadrightarrow v}
$$

concrete syntaxe	*op*	semantics $[\![op]\!]$
e `+` e	*add*	$[\![add]\!]([v_0,\ldots,v_{n-1}],[v'_0,\ldots,v'_{m-1}]) \stackrel{\text{def}}{=} [v_0,\ldots,v_{n-1},v'_0,\ldots,v'_{m-1}]$ $[\![add]\!](v_0,v_1) \stackrel{\text{def}}{=} num(v_0) + num(v_1)$, otherwise
e `-` e	*sub*	$[\![sub]\!](v_0,v_1) \stackrel{\text{def}}{=} num(v_0) - num(v_1)$
`len(`e`)`	*len*	$[\![len]\!]([v_0,\ldots,v_{n-1}]) \stackrel{\text{def}}{=} n$
`[`e`]`	*mk*	$[\![mk]\!](v) \stackrel{\text{def}}{=} [v]$
e`[`e`]`	*get*	$[\![get]\!]([v_0,\ldots,v_{n-1}],i) \stackrel{\text{def}}{=} v_{num(i)}$ if $0 \le num(i) < n$
e `==` e	*eq*	$[\![eq]\!](v_0,v_1) \stackrel{\text{def}}{=}$ `True` if $v_0 = v_1$ or $num(v_0) = num(v_1)$ `False` otherwise

Figure 3: Big-step Operational Semantics.

Appendix: x86-64 cheat sheet

A fragment of the x86-64 instruction set is given here. You are free to use any other part of the x86-64 assembler. In the following, r_i designates a register, n an integer constant and L a label.

`mov` r_2`,` r_1	copies register r_2 into register r_1
`mov` `$`n`,` r_1	loads constant n into register r_1
`mov` `$`L`,` r_1	loads the address of label L into register r_1
`sub` r_2`,` r_1	computes $r_1 - r_2$ and stores it into r_1
`mov` n`(`r_2`),` r_1	loads r_1 with the value contained in memory at address $r_2 + n$
`mov` r_1`,` n`(`r_2`)`	writes in memory at address $r_2 + n$ the value of r_1
`push` r_1	pushes the value of r_1 on the stack
`pop` r_1	pops a value from the stack and stores it into register r_1
`test` r_2`,` r_1	sets the flags according to the value of r_1 AND r_2
`jz` L	jumps to address L if flags signal a zero value
`jmp` L	jumps to address L
`call` L	pushes the return address to the stack and jumps to address L
`ret`	pops an address from the stack and jumps there

Calling conventions:

- up to six arguments are passed via registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`;
- other arguments are passed on the stack, if any;
- the returned value is put in `%rax`;
- registers `%rbx`, `%rbp`, `%r12`, `%r13`, `%14` and `%r15` are *callee-saved*: they won't be clobbered by a `call`;
- the other registers are *caller-saved*: they may be clobbered by a `call`;
- `%rsp` is the stack pointer, `%rbp` the *frame pointer*.