École Polytechnique

CSC_52064 : Compilation

examen 2025 (X2022)

Jean-Christophe Filliâtre

17 mars 2025 — 14h00–17h00

The test lasts 3 hours. Handwritten or printed course notes are the only documents allowed. Most 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 1–3 are grouped together at the end of the subject on page 5.

Suggestion: detach the last sheet.

Throughout this subject, we consider a small imperative language called WHILE, whose abstract syntax is given in figure 1. A program is limited to a single statement. A statement (noted s) is an assignment, a display with `print`, a conditional, a `while` loop or a block (a sequence of statements, possibly empty). A variable (noted x) contains an integer and assignments are limited to nonnegative constants and subtractions of variables. Here is an example:

```
{ a=0  b=1
  while n<>0 do { t=0  t=t-b  b=a-t  a=b-a  t=1  n=n-t }
  print a }
```

Semantics. We equip WHILE with a small-step operational semantics, taking the form of a binary relation noted $\to$ between two configurations. A configuration, noted $\langle E, s\rangle$, is an environment E and a statement s. An environment is a partial function from variables to integers. Its domain, that is the set of variables for which E is defined, is noted $\mathrm{dom}(E)$. The relation $\langle E, s\rangle \to \langle E', s'\rangle$ reads as follows: In the environment E, the statement s successfully executes one step of computation and reaches environment E' and statement s'.

Figure 2 defines the relation $\to$ with a set of inference rules. In these rules, notation $E \oplus \{x \mapsto n\}$ stands for the function E' defined by

$$\begin{array}{rcl} E'(x) & = & n, \\ E'(y) & = & E(y) \ \text{ for } y \in \mathrm{dom}(E) \text{ and } y \neq x. \end{array}$$

In particular, we have $\mathrm{dom}(E') = \mathrm{dom}(E) \cup \{x\}$. Note carefully how the rules defining $\to$ manipulate environments. If `fib` stands for the example program above, we have the following execution,

$$\langle \{\mathtt{n} \mapsto 10\}, \mathtt{fib}\rangle \to^\star \langle \{\mathtt{a} \mapsto 55, \mathtt{n} \mapsto 0, \mathtt{b} \mapsto 89, \mathtt{t} \mapsto 1, \}, \mathtt{\{\ \}}\rangle$$

(and the program printed 55).

A configuration C is said to be irreducible if there is no configuration C' such that $C \to C'$. We say that the execution of a configuration *blocks* if it reaches, after some execution steps, an irreducible configuration $\langle E, s\rangle$ with $s \neq \mathtt{\{\ \}}$. A trivial example of blocking configuration is $\langle \emptyset, \mathtt{print}\ x\rangle$ since the variable x is not defined in the empty environment. An execution that does not block terminates on the statement `{ }` or never terminates.

Question 1 Give the execution steps of the program

```
{ x=1 if x=0 then y=1 else z=2 print y }
```

in an empty initial environment.

Question 2 Propose a program that, in an environment E defining at least the variables a and b, with $E(b) \geq 0$, prints the value of $E(a) \times E(b)$ and terminates. How can you relax the hypothesis $E(b) \geq 0$? (Do not give the code this time, but only the idea.)

Parsing. We wish to implement a parser for the language WHILE.

Question 3 Discuss the problem of the constant `0` which appears in the syntax of the constructions `if` and `while` but also in an assignment of the form x `= 0`. Show precisely how this can be taken into account in both lexical and syntactic analysis. Try to suggest two different solutions.

Question 4 Give a grammar for the language WHILE in the syntax of either CUP or Menhir (your choice). It must be able to parse a file containing a single statement. Declare the tokens and the start symbol. Declare associativity rules and priorities, if needed. *Do not include the semantic actions.*

Static Analysis of Definitions and Uses. For a program not to block, it is sufficient that any variable used by a statement (to the right of an assignment or as an argument of a `print`, `if` or `while` statement) must be present in the environment at the time of execution. This variable may have been present in the environment since the start of execution, or it may have been introduced into the environment by a previously executed assignment. For instance, the program

```
{ if x = 0 then y = 1 else y = 2
  print y }
```

successfully executes in an initial environment that defines only `x`.

We propose to statically determine, for a statement s, a set of variables noted $use(s)$ sufficient for its correct execution, in the following sense: for any environment E such that $use(s) \subseteq \mathrm{dom}(E)$, the execution of $\langle E, s\rangle$ does not block. Of course, it would be sufficient to take for $use(s)$ all the variables that appear in s, but this would be a very crude solution. For the program above, for example, the set $\{\mathtt{x}\}$ is sufficient.

In order to perform a more detailed analysis of $use(s)$, we are going to simultaneously compute a second set of variables, noted $def(s)$, corresponding to variables necessarily defined by the execution of s, in the following sense: for any execution $\langle E, s\rangle \to^\star \langle E', \mathtt{\{\ \}}\rangle$, then we have $def(s) \subseteq \mathrm{dom}(E')$. For the program above, for example, we have $def(s) = \{\mathtt{y}\}$ because every execution gives a value to the variable `y`.

We propose to compute the sets $def(s)$ and $use(s)$ by induction on the structure of s. Here is how to perform the calculation for the first three statements:

	$def(s)$	$use(s)$
x `=` n	$\{x\}$	$\emptyset$
x_1 `=` x_2`-`x_3	$\{x_1\}$	$\{x_2, x_3\}$
`print` x	$\emptyset$	$\{x\}$

Question 5 Complete the table above by proposing a computation of $def(s)$ and $use(s)$ for the remaining three statements of WHILE: conditional, loop, and block.

Question 6 Show the correctness of your definition of *use*(s). Make sure you clearly state and prove all the intermediate properties you need.

Question 7 Is an algorithm (yours or any other) able to compute sets *def*(s) and *use*(s) that are minimal for inclusion? If yes, justify. If no, explain why.

RTL Language. We are now considering an RTL-type language (for Register Transfer Language) whose abstract syntax is given in figure 3. An RTL program is a finite set of basic blocks. A basic block b is given by a label L, a sequence of instructions, and a jump. An instruction i is an assignment or display, as in the WHILE language. A jump j is either the termination of the program (`halt`), a conditional jump (`ifz` x L_1 L_2) or an unconditional jump (`goto` L). An RTL program is well-formed if

- any label mentioned by `ifz` or `goto` corresponds to a program block;
- there is no jump `goto` L to a block with only one predecessor (*i.e.* to a block other than the first block, whose only reference is this `goto` L jump).

Here is an example of a well-formed RTL program with four basic blocks:

```
L0: n=10  r=0  one=1  one=r-one  goto L1
L1: t=r-n  ifz t L2 L3
L2: halt
L3: r=r-one  print r  goto L1
```
(R_0)

Note that block `L1` has two predecessors.

An RTL program runs in an E environment (of the same type as for WHILE) and starts at its first block (`L0` in the example above). The execution of a block is the execution of its instructions, in sequence, with the same semantics as for WHILE. If the execution of an instruction blocks, the RTL program also blocks. Otherwise, we get a new environment E' and we make the jump. The `halt` jump terminates the RTL program. The conditional jump `ifz` x L_1 L_2 continues execution on block L_1 if $E'(x) = 0$ and on block L_2 otherwise. The unconditional jump `goto` L continues execution on block L. In the case of a jump, execution continues with the environment E'. For instance, the above program prints the integers from 1 to 10 and then terminates.

Question 8 What is the effect of the following program

```
L0: zero=0  one=1  one=zero-one  u=1  r=0  goto L1
L1: t=r-n  ifz t L6 L2
L2: v=u-zero  s=0  goto L3
L3: t=s-r  ifz t L5 L4
L4: t=zero-v  u=u-t  s=s-one  goto L3
L5: r=r-one  goto L1
L6: print u  halt
```

in an environment that defines a variable `n` with a nonnegative value? Provide a detailed justification.

Question 9 Propose an abstract syntax (in Java or OCaml) for RTL programs. Propose data structures and methods/functions (in Java or OCaml) to execute an RTL program from an empty environment. *You do not have to implement these methods/functions.*

Question 10 We wish to compute the set *use* of an RTL program, with the same meaning as for a WHILE program (*i.e.*, execution does not block on an environment that defines at least the variables of the set *use*). Propose an algorithm to compute this set.

Question 11 Propose an algorithm to compile the WHILE language to the RTL language. Make sure that the resulting RTL program is well-formed.

Compiling to x86-64 Assembly. We propose to compile the RTL language to x86-64 assembly. (A cheat sheet is given in the appendix). We assume that the integers in our language are limited to signed 64-bit integers. We assume that an assembly function `print` is given, obeying the calling conventions (its argument is in `%rdi` and it potentially overwrites any *caller-saved* register) but not requiring stack alignment when called.

As our RTL language is very simple, we can perform register allocation by graph coloring on an interference graph built directly from the RTL code. For the next two questions, we take the following RTL program as an example:

```
A: zero=0  s=0  one=1  n=10  goto B
B: t=zero-n  s=s-t  n=n-one  ifz n H B                    (R1)
H: print s  halt
```

Question 12 Give the interference graph for program (R_1). Try to suggest one or more preference edges (with justification).

Question 13 Propose an assembly code for program (R_1).

Question 14 Propose an assembly code for program (R_0) on page 3.

Question 15 Generally speaking, what are the preference edges we can derive from an RTL program?

Question 16 Propose a general compilation scheme for the RTL language, *i.e.*, explain how each of its constructs (instruction or jump) can be compiled to x86-64 assembly.

$$
\begin{array}{rcll}
s & ::= & x\texttt{=}n & \textit{constant } n \in \mathbb{N} \\
& | & x\texttt{=}x\texttt{-}x & \textit{subtraction} \\
& | & \texttt{print}\ x & \textit{display} \\
& | & \texttt{if}\ x\ \texttt{=}\ \texttt{0}\ \texttt{then}\ s\ \texttt{else}\ s & \textit{conditional} \\
& | & \texttt{while}\ x\ \texttt{<>}\ \texttt{0}\ \texttt{do}\ s & \textit{loop} \\
& | & \texttt{\{}\, s \ldots s\, \texttt{\}} & \textit{block}
\end{array}
$$

Figure 1: Abstract Syntax of WHILE.

$$
\frac{}{\langle E, x\texttt{=}n\rangle \to \langle E \oplus \{x \mapsto n\}, \texttt{\{ \}}\rangle}
\qquad
\frac{x_2 \in \mathrm{dom}(E) \quad x_3 \in \mathrm{dom}(E)}{\langle E, x_1\texttt{=}x_2\texttt{-}x_3\rangle \to \langle E \oplus \{x_1 \mapsto E(x_2) - E(x_3)\}, \texttt{\{ \}}\rangle}
$$

$$
\frac{x \in \mathrm{dom}(E)}{\langle E, \texttt{print}\ x\rangle \to \langle E, \texttt{\{ \}}\rangle}
$$

$$
\frac{x \in \mathrm{dom}(E) \quad E(x) = 0}{\langle E, \texttt{if}\ x\ \texttt{=}\ \texttt{0}\ \texttt{then}\ s_1\ \texttt{else}\ s_2\rangle \to \langle E, s_1\rangle}
\qquad
\frac{x \in \mathrm{dom}(E) \quad E(x) \neq 0}{\langle E, \texttt{if}\ x\ \texttt{=}\ \texttt{0}\ \texttt{then}\ s_1\ \texttt{else}\ s_2\rangle \to \langle E, s_2\rangle}
$$

$$
\frac{x \in \mathrm{dom}(E) \quad E(x) = 0}{\langle E, \texttt{while}\ x\ \texttt{<>}\ \texttt{0}\ \texttt{do}\ s\rangle \to \langle E, \texttt{\{ \}}\rangle}
\qquad
\frac{x \in \mathrm{dom}(E) \quad E(x) \neq 0}{\langle E, \texttt{while}\ x\ \texttt{<>}\ \texttt{0}\ \texttt{do}\ s\rangle \to \langle E, \texttt{\{}\, s\ \texttt{while}\ x\ \texttt{<>}\ \texttt{0}\ \texttt{do}\ s\, \texttt{\}}\rangle}
$$

$$
\frac{}{\langle E, \texttt{\{\{ \}}\ s \ldots \texttt{\}}\rangle \to \langle E, \texttt{\{}\, s \ldots \texttt{\}}\rangle}
\qquad
\frac{\langle E, s_1\rangle \to \langle E_1, s_1'\rangle}{\langle E, \texttt{\{}\, s_1\ s_2 \ldots \texttt{\}}\rangle \to \langle E_1, \texttt{\{}\, s_1'\ s_2 \ldots \texttt{\}}\rangle}
$$

Figure 2: Operational Semantics of WHILE.

$$
\begin{array}{rcll}
p & ::= & b \ldots b & \textit{RTL program} \\
b & ::= & L : i \ldots i\ j & \textit{basic block} \\
i & ::= & x\texttt{=}n & \textit{constant } n \in \mathbb{N} \\
& | & x\texttt{=}x\texttt{-}x & \textit{subtraction} \\
& | & \texttt{print}\ x & \textit{display} \\
j & ::= & \texttt{halt} & \textit{end of program} \\
& | & \texttt{ifz}\ x\ L\ L & \textit{conditional jump} \\
& | & \texttt{goto}\ L & \textit{unconditional jump}
\end{array}
$$

Figure 3: RTL Language.

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
`neg` r_1	computes $-r_1$ 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 `%rsp`, `%rbp`, `%rbx`, `%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*.