Univ. Grenoble Alpes **UFR IM^{2}AG**
MoSIG 1 **Part I** **Year 2021-2022**

Programming Language Semantics and Compiler Design

—

Final Exam of Thursday 9 December

- **Duration**: 3h.
- 5 sheets of A4 paper are authorized.
- Any electronic device is forbidden.
- The grading scale is indicative.
- **The care of your submission will be taken into account.**
- Exercises are **independent**.
- If you don't know how to answer to some question, you may assume the result and proceed with the next question.
- The maximal grade is obtained with 20 points.

- **Submit each part on a separate answer sheet (negative point otherwise).**
- **Care will be taken into account (-1 point in case of lack of care).**
- **Unreadable parts will be ignored.**

Exercise 1 — Structural Operational Semantics - modeling a debugger (10 points)

We consider language **While** and its structural operational semantics, as defined in the course, and relying on semantic functions $\mathcal{A}$ and $\mathcal{B}$ for arithmetic and Boolean expressions. In particular, we use the set **Vars** of variables appearing in the program. We want to model the use of a debugger that executes in parallel with the debugged programs. The debugger offers means to the programmer to interact with the program, modify it, and observe the state of the program. We consider only the notion of watch point, that is point in the execution that are related to the reading or the writing of some variables of interest (i.e., that are watched). We recall that structural operational semantics implicitly defines a notion of atomic step, which is achieved by either the execution of an assignment, the execution of the `skip` statement, or the evaluation of Boolean conditions, or the rewriting of iterative `while` b `do` S `od` statements.

We consider that the debugger offers the following commands:

- `step` to let the program performs one execution step;
- `play` to resume the execution;
- `add` $-$ `read`(x) to add variable $x \in$ **Vars** to the watched read variables;
- `add` $-$ `write`(x) to add variable $x \in$ **Vars** to the watched write variables;
- `rem` $-$ `read`(x) to remove variable $x \in$ **Vars** from the watched read variables;
- `rem` $-$ `write`(x) to remove variable $x \in$ **Vars** from the watched write variables;
- `set`(x, z) to set variable $x \in$ **Vars** to some value $z \in \mathbb{Z}$.

Note, if one of the above commands is applied to a variable that is not part of the program (i.e., not in **Vars**), the command should have no effect.

The debugger can be in two modes:

- `in`, which is the interactive mode;
- `nin`, which is the non-interactive mode.

We denote by **Modes** the set of modes of the debugger. In interactive mode, the execution is suspended and the debugger waits for input from the user. In non-interactive mode, the program runs normally (without interaction) and the execution is suspended only if a watch point is triggered.

In addition, at runtime, the debugger maintains two sets of variables that correspond to the variables that are watched for read and write statements. If the execution of the program reads (resp. writes) a variable that is a watched read (resp. write) variable, the execution is suspended and the mode becomes interactive.

In interactive mode, only an input user debugger command can make the debugged program evolve and we assume that it is the user action of inputting a debugger command modifies the configuration. We do not model such configuration change though.

In the following questions, you may use some function *Used* : **Aexp** $\cup$ **Bexp** $\rightarrow \mathbf{2}^{\mathbf{Vars}}$ that returns the set of used variables in an arithmetic or Boolean expression.

1. **(0,5 point)** Define formally the set **Commands** of commands that can be executed by the debugger.
2. **(0,5 point)** Define formally the set **Watchpoints** of all watchpoints that are possible in the configurations of the debugged programs.
3. **(1,5 points)** Compared to structural operational semantics, the configurations of the debugged programs additionally contain the debugger mode, some active watchpoints, and a command. Note, the input command can be empty, which we denote by $-$.
   Define the set of (possible) configurations of the debugged programs. Indicate non-final and final configurations.
4. **(0,75 point)** Recall the signature of $\Rightarrow$, the transition relation between configurations.
5. **(0,75 point)** Command `play` changes the mode of the debugger from interactive to non-interactive. After executing, the input command becomes empty ($-$). Other elements in the configuration are not changed. Give the corresponding semantic rule.
6. **(2 points)** Command `step` is executed in interactive mode. The effect of the command depends on the first statement of the debugged program. Command `step` essentially allows executing one execution step of the program, similarly to the atomic step of structural operational semantics of non-debugged programs. Give the semantic rules when the command input to the debugger is `step`.
7. **(2 points)** Give the semantic rules for adding and removing watchpoints.
8. **(2 points)** In non-interactive mode, the program is suspended (and the debugger switches to interactive mode) only when a watchpoint is triggered. Give the semantic rules corresponding to the non-interactive mode of the debugger.

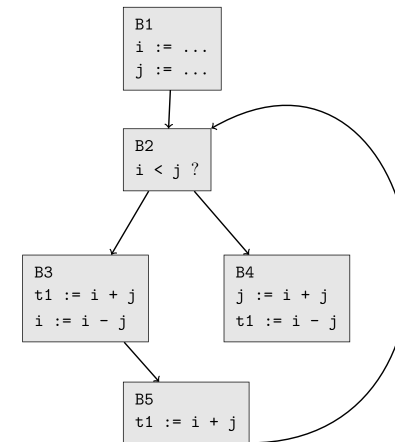

Figure 1: Initial control-flow graph

Exercise 2 — Optimization: available expressions (3 points)

We consider the control-flow graph in Figure 1. We consider expressions $e_1 = i + j$ and $e_2 = i - j$ and the set of expressions $\Sigma = \{e_1, e_2\}$.

1. Compute the sets Gen(b) and Kill(b) for each basic block b.
2. Compute the sets In(b) and Out(b) for each basic block b.
3. Suppress redundant computations.