

Coursework 1

Due: Tuesday 10 March, 5.00pm

For this coursework you are asked to implement the semantics of various extensions of IMP in Haskell. You are given predefined functions in `coursework_1.hs`, and you may re-use any code provided in this unit, including the solutions to tutorial exercises. Your solutions should consist of a single Haskell file, which uses the specified function names, and is accepted by the compiler without errors. Non-working partial solutions may be included as comments, and will be considered for partial marks. Submission is through Moodle, by the above deadline. This coursework is an individual assignment.

Assignment 1 (65%): Programming languages such as Java and C include *post-increment* and *post-decrement* operators on variables. An expression of the form `v++` returns the value stored in variable `v`, but has the side-effect of incrementing the stored value by 1. In this exercise we give a formal semantics to an extension of IMP with these operators. The syntax of arithmetic expressions becomes:

$$A ::= n \mid v \mid A + A \mid A * A \mid A - A \mid v++ \mid v--$$

The syntax of Boolean expressions and commands is as for IMP. The interpretation functions have the following type, and the semantic domains are as follows:

$$\begin{array}{ll} \mathcal{A}[-] : Aexp \rightarrow \mathbf{Arit} & \mathbf{Arit} = \mathbf{ST} \rightarrow (\mathbf{ST} \times \mathbb{Z}) \\ \mathcal{B}[-] : Bexp \rightarrow \mathbf{Bool} & \mathbf{Bool} = \mathbf{ST} \rightarrow (\mathbf{ST} \times \mathbb{B}) \\ \mathcal{C}[-] : Comm \rightarrow \mathbf{Comm} & \mathbf{Comm} = \mathbf{ST} \rightarrow \mathbf{ST} \end{array}$$

Note that the domains for arithmetic and Boolean expressions have been amended so that expressions return not only a value but also a new state. The semantics of *post-increment* and *post-decrement* are as follows. In accordance with the domain **Arit**, the interpretation returns a new state t as well as a value n .

$$\mathcal{A}[v++](s) = (t, n) \quad \text{where} \quad t = s[v \mapsto n + 1] \quad \text{and} \quad n = s(v)$$

$$\mathcal{A}[v--](s) = (t, n) \quad \text{where} \quad t = s[v \mapsto n - 1] \quad \text{and} \quad n = s(v)$$

The semantics of all binary operators, both for A and for B , is to evaluate the left argument before the right argument. For example:

$$\mathcal{A}[A + B](s) = (u, a + b) \quad \text{where} \quad \mathcal{A}[A](s) = (t, a) \quad \text{and} \quad \mathcal{A}[B](t) = (u, b)$$

While the semantics of commands remains has not changed from that of IMP, the evaluation function $\mathcal{C}[-]$ still needs to be adapted to the new semantics of $\mathcal{A}[-]$ and $\mathcal{B}[-]$. The changes are the following:

$$\mathcal{C}[v := A](s) = t[v \mapsto a] \quad \text{where} \quad \mathcal{A}[A](s) = (t, a)$$

$$\mathcal{C}[\text{if } B \text{ then } C_1 \text{ else } C_2](s) = \begin{cases} \mathcal{C}[C_1](t) & \text{if } b = \text{true} \\ \mathcal{C}[C_2](t) & \text{otherwise} \end{cases} \quad \text{where} \quad \mathcal{B}[B](s) = (t, b)$$

$$\mathcal{C}[\text{while } B \text{ do } C](s) = \begin{cases} \mathcal{C}[\text{while } B \text{ do } C](u) & \text{if } b = \text{true} \\ t & \text{otherwise} \end{cases}$$

where $\mathcal{B}[B](s) = (t, b)$ and $\mathcal{C}[C](t) = u$

The function `evalC` in `coursework_1.hs` implements the above semantics. You are also given a sample `factorial` program (in comments) to test your answers to this assignment. It implements the following algorithm:

$$\text{factorial } x = \begin{cases} y = 1; \\ \text{while } 1 \leq x \text{ do } y = y \times (x--); \\ \text{return } y \end{cases}$$

- Complete the data type `Aexp` for `Aexp` given in `coursework_1.hs`, using the constructors `Incr` and `Decr` for *post-increment* and *post-decrement*. Uncomment `factorial` and `runfactorial`, which should pass type-checking.
- Give type signatures to the functions `evalA` and `evalB` matching the given semantic domains.
- Complete the evaluation functions `evalA` and `evalB`, implementing $\mathcal{A}[-]$ and $\mathcal{B}[-]$. Use `runFactorial` to test your code.
- The *pre-increment* and *pre-decrement* operators `++v` and `--v` first change the value of the stored variable `v` by 1, and then return the new value. Implement *pre-increment* and *pre-decrement*: add suitable constructors `PreIncr` and `PreDecr` to `Aexp`, and add the corresponding cases to `evalA`.

Assignment 2 (25%): The Boolean operators `(&&:)` and `(&|:)` we have asked you to implement in the previous assignment are the *eager* operators: they evaluate both arguments. Not all programming languages have eager Boolean operators: for example, C and Haskell do not. Instead, the standard implementation of the Boolean operators is as *short-circuit* operators, which only evaluate their second argument when needed. For example, an expression `(a || b)` in Haskell evaluates `a`, returns `True` (without inspecting `b`) if the result is `True`, and otherwise returns the evaluation of `b`.

- Add syntax for short-circuit Boolean operators to your implementation. Use the constructor names `(&&:)` and `(&|:)`.
- Implement a suitable semantics of `(&&:)` and `(&|:)` in your function `evalB`. If the first argument of `(&&:)` evaluates to `False`, the second argument should not affect the state; and similarly for `(&|:)` and `True`.
- Define expressions `b1 :: Bexp`, `b2 :: Bexp`, and `st :: State` such that the following two expressions return a different state:

```
evalB (b1 &&: b2) st
evalB (b1 &|: b2) st
```

Give expressions $b3 :: \text{Bexp}$ and $b4 :: \text{Bexp}$ that are identical up to replacing eager operators by the corresponding short-circuit ones (in either direction), such that $b3$ and $b4$ evaluate to distinct values in the state st .

Assignment 3 (10%): For the last part of this coursework we will extend your implementation with a command `print A` that outputs the value of an arithmetic expression A . **Note:** this assignment makes the final 10% of your mark hard-earned, and represents considerably more than 10% of the total effort.

Output will be modeled by sequences of integers:

$$\text{Out} = \mathbb{Z}^*$$

We will change the semantic domain for commands to keep track of output:

$$\mathcal{C}[-] : \text{Comm} \rightarrow \text{CommOut} \quad \text{CommOut} = \text{ST} \rightarrow (\text{ST} \times \text{Out})$$

You are given the semantics of the following three commands, where ε is the empty sequence and (\cdot) concatenates two sequences:

$$\mathcal{C}[\text{print } A](s) = (t, a) \quad \text{where} \quad \mathcal{A}[A](s) = (t, a)$$

$$\mathcal{C}[v := A](s) = (t[v \mapsto a], \varepsilon) \quad \text{where} \quad \mathcal{A}[A](s) = (t, a)$$

$$\mathcal{C}[C_1 ; C_2](s) = (u, \alpha \cdot \beta) \quad \text{where} \quad \mathcal{C}[C_1](s) = (t, \alpha) \quad \text{and} \quad \mathcal{C}[C_2](t) = (u, \beta)$$

(Note that in the case for `print A` the value a is an integer in one case, and a sequence of a single integer in the other case.)

- We will use a list of integers to model output. Give a type synonym `Output` for lists of integers.
- Amend the type of `evalC` to reflect the new semantic domain. (To pass type-checking, you should return the definitions to `undefined`, and comment out `runFactorial`.)
- Add a suitable constructor named `Print` to the syntax of commands `Comm`.
- Re-implement the function `evalC` to reflect the new semantics of commands. Give a suitable interpretation to the three remaining commands `skip`, `if-then-else`, and `while-do`.