Oblig 4: The `AeroScript` Type System

Deadline: 20 November 2025
Version 1.01

1 Introduction

With the structure and functionality of `AeroScript` taking shape, your fourth assignment will focus on ensuring the robustness of your language. We have designed a static type system for `AeroScript` and it is your task to implement type checking across the entire AST. This step is crucial for catching errors early and ensuring that `AeroScript` programs run efficiently and safely in high-stakes drone operations. Dive into the specifications and elevate `AeroScript` to meet the rigorous demands of type safety. Off we go!

2 The Objective

In this assignment, you will implement a static type checker on the `AeroScript` AST that you have defined in the previous assignments. Here, *static* simply means that the type checking stage is its own, isolated stage that should be performed *after* parsing, but *before* execution. If a given program does not adhere to the typing rules, any further execution should be halted and an appropriate error message should be printed.

3 Syntax

For easy reference, you can find the full concrete syntax of `AeroScript` in Figure 1. There are no changes to the concrete syntax in Assignment 3.

$$
\begin{aligned}
& n \in \mathbb{R}^{+} \\
& id \in \text{NAME} \\
\textit{program} &:= (\textit{mode})^{+} \\
\textit{mode} &:= \text{`}\to\text{`}^{?}\ \textit{id}\ \text{`\{`}\ (\textit{statement})^{*}\ \text{`\}`}\ (\text{`}\to\text{`}\ \textit{id})^{?} \\
\textit{statement} &:= \textit{action} \mid \textit{reaction} \\
\textit{reaction} &:= \text{`on`}\ \textit{event}\ \text{`}\to\text{`}\ \textit{id} \\
\textit{event} &:= \text{`obstacle`} \mid \text{`low battery`} \mid \text{`message`}\ \text{`[`}\ \textit{id}\ \text{`]`} \\
\textit{action} &:= (\textit{acDock} \mid \textit{acMove} \mid \textit{acTurn} \mid \textit{acAscend} \mid \textit{acDescend}) \\
&\quad (\text{`for`}\ \textit{expression}\ \text{`seconds`} \mid \text{`at speed`}\ \textit{expression})^{?} \\
\textit{acDock} &:= \text{`return to base`} \\
\textit{acMove} &:= \text{`move`}\ (\text{`to`}\ \text{`point`}\ \textit{point} \mid \text{`by`}\ n) \\
\textit{acTurn} &:= \text{`turn`}\ (\text{`right`} \mid \text{`left`})^{?}\ \text{`by`}\ \textit{expression} \\
\textit{acAscend} &:= \text{`ascend by`}\ \textit{expression} \\
\textit{acDescend} &:= \text{`descend by`}\ \textit{expression} \mid \text{`descend to ground`} \\
\textit{expression} &:= \text{`}-\text{-`}\textit{expression} \mid \textit{expression} \oplus \textit{expression} \mid \text{`random`}\ \textit{range}^{?} \\
&\quad \mid \text{`point`}\ \textit{point} \mid n \mid \text{`(`}\ \textit{expression}\ \text{`)`} \\
\textit{range} &:= \text{`[`}\ \textit{expression}\ \text{`,`}\ \textit{expression}\ \text{`]`} \\
\textit{point} &:= \text{`(`}\ \textit{expression}\ \text{`,`}\ \textit{expression}\ \text{`)`} \\
\oplus &:= \text{`+`} \mid \text{`-`} \mid \text{`*`}
\end{aligned}
$$

Figure 1. The syntax of `AeroScript`. The star ($\odot^{*}$) denotes a sequence containing 0 or more occurrences. and the question mark ($\odot^{?}$) denotes choice (zero or one occurrence), the plus ($\odot^{+}$) denotes a sequence containing one or more occurrences. NAME is the set of all possible identifiers.

4 The Typing Rules

In order for an `AeroScript` program to be well-typed, a program should follow the rules defined here. All rules are written in one of the following two ways (called judgement rules):

$$\vdash_{prod} term : \tau \ ,$$

or

$$\vdash_{prod} term \ ,$$

where *prod* are the production of the syntactic categories, *term* the given terms of the concrete syntax and τ the set of types. The former is used for syntactic categories that "return" a typed values (expressions etc.) while the latter is used to syntactic categories that do not return values (statements etc.). The typing terms are defines as

$$\tau := \textsc{Num} \mid \textsc{Range} \mid \textsc{Point} \ .$$

Figure 2 shows the rules for *point*s, *range*s, and *expression*s. Points and ranges types to the types Point and Range, respectively, under the condition that both their sub-terms types to Num. Simple numerals can always trivially be inferred to have numeral type Num. Additionally, a `random` expression will trivially be of type Num as long as either no range is given, or the type Range. Finally, any pair of parentheses containing an expression will have the same type (denoted by τ) as the contained expression. Remember back in A1 we defined that:

> when binary operators or negation are applied to scalars (i.e., integers), all is well; applying addition and subtraction to two points is also well-defined. Further, multiplying a point (or *vector*) by a scalar simply multiplies each component of the point by the scalar, and negating a point just negates each component of the point. However, any other operation on vectors is *undefined*.

Let us start with unary negation by defining it to be typing it to the same as its sub-expression[1]. However, the above description only defines it to work for Num and Point, so we can restrict it with a side-condition ensuring that Range is not well-typed for unary negation. Binary operators are left for you to define according to the above description; we will detail this later.

$$\frac{\vdash_{exp} e_1 : \textsc{Num} \quad \vdash_{exp} e_2 : \textsc{Num}}{\vdash_{point} (e_1, e_2) : \textsc{Point}} \qquad \frac{\vdash_{exp} e_1 : \textsc{Num} \quad \vdash_{exp} e_2 : \textsc{Num}}{\vdash_{range} [e_1, e_2] : \textsc{Range}} \qquad \frac{}{\vdash_{exp} n : \textsc{Num}}$$

$$\frac{\vdash_{exp} e : \tau}{\vdash_{exp} \texttt{--}e : \tau}\, \tau \neq \textsc{Range} \qquad \frac{\vdash_{exp} e_1 : ? \quad \vdash_{exp} e_2 : ?}{\vdash_{exp} e_1 \oplus e_2 : ?}$$

$$\frac{}{\vdash_{exp} \texttt{random} : \textsc{Num}} \qquad \frac{\vdash_{range} range : \textsc{Range}}{\vdash_{exp} \texttt{random}\ range : \textsc{Num}} \qquad \frac{\vdash_{point} point : \textsc{Point}}{\vdash_{exp} \texttt{point}\ point : \textsc{Point}} \qquad \frac{\vdash_{exp} e : \tau}{\vdash_{exp} (e) : \tau}$$

Figure 2. The typing rules of `AeroScript` expressions, points, and ranges. Note that binary operators are left as an exercise for the reader.

For statements, you should use these typing rules to enforce the types of the contained expressions as shown in Figure 3. Note that we have collapsed some of the categories to make a term match just one rule. All rules expect for `move to` terms, states that sub-expressions must type to Num; in `move to` the sub-expression must type to Point.

Finally, all *mode*s in all *program*s must only contain well-typed statements.

5 Examples

This section aims to provide some concrete examples of both well-typed and ill-typed statements. In the first example, we correctly ascend by an expression of type Num—however, we subsequently move to an ill-typed point containing another point in its second component:

[1] We call it for ad-hoc polymorphism when operators can be used for several different types.

$$\frac{}{\vdash_{stms} \texttt{return to bases}} \qquad \frac{\vdash_{exp} e : \textsc{Num}}{\vdash_{stms} \texttt{move by } e} \qquad \frac{\vdash_{exp} e : \textsc{Point}}{\vdash_{stms} \texttt{move to } e}$$

$$\frac{\vdash e : \textsc{Num}}{\vdash_{stms} \texttt{turn left by } e} \qquad \frac{\vdash e : \textsc{Num}}{\vdash_{stms} \texttt{turn right by } e} \qquad \frac{\vdash e : \textsc{Num}}{\vdash_{stms} \texttt{for } e \texttt{ s}} \qquad \frac{\vdash e : \textsc{Num}}{\vdash_{stms} \texttt{at speed } e}$$

$$\frac{\vdash e : \textsc{Num}}{\vdash_{stms} \texttt{ascend by } e} \qquad \frac{\vdash e : \textsc{Num}}{\vdash_{stms} \texttt{descend by } e} \qquad \frac{\vdash e : \textsc{Num}}{\vdash_{stms} \texttt{descend to ground}}$$

Figure 3. The typing rules of relevant `AeroScript` statements.

```
TakeOffLand {
  ascend by 20
  on low battery -> EmergencyLanding
  move to point (50, point (10, 20))
} -> ScanArea
```

In this next example, we attempt to move to a (valid) point, but at a speed given by another (in and of itself well-typed) point:

```
ScanArea {
  move to point (100, 100) at speed point (25, 25)
  on low battery -> EmergencyLanding
} -> EndMission
```

In the third example, we have switched it around; here, we attempt to move to a numeral value:

```
TakeOff {
  ascend by 20
  on low battery -> EmergencyLanding
  move to 30
} -> Explore
```

In order to drive the point home, the following snipped contains a `random` expression with an ill-typed range:

```
Explore {
  move to point (random[0, point (10, 15)], random[0, random]) at speed 10
  on low battery -> EmergencyLanding
  on message [end_tour] -> ReturnToBase
}
```

6 Typing rules for binary operators

Remember, we did not finish all the rules. Specifically, the rule for binary operators were just left as this

$$\frac{\vdash_{exp} e_1 :? \quad \vdash_{exp} e_2 :?}{\vdash_{exp} e_1 \oplus e_2 :?}$$

It is your task define the missing typing rules such that they fit the adapted description from A1 as

> when binary operators or negation are applied to scalars (i.e., integers), all is well-typed ; applying addition and subtraction to two points is also well-typed . Further, multiplying a point (or *vector*) by a scalar simply multiplies each component of the point by the scalar, and negating a point just negates each component of the point ; hence these operations are also well-typed . However, any other operation on vectors is ill-typed .

Note, that I here use plural for rules. There are many ways of defining this, but it is easiest to define more rules. Some will be very on case-basis and even exchanging $\oplus$ for specific binary operators.

We use the following LaTeX code for defining the typing rules. You are welcome to use this in your report. Note that this is not a valid rule.[2]

```
\usepackage{xspace}

\newcommand{\tpint}{\textsc{Num}\xspace}
\newcommand{\tppoint}{\textsc{Point}\xspace}
\newcommand{\tprange}{\textsc{Range}\xspace}

$$
    \frac
      {\vdash_{exp} e_1 : \tprange \quad \vdash_{exp} e_2 : \tau}
      {\vdash_{exp} e_1 \oplus e_2 : \tpint}
      \mathrm{Side-condition}
$$
```

7 Your task

You are tasked with implementing the type system for `AeroScript`. In particular, you are expected to do the following:

1. Define the typing rules for binary operators.
2. Implement static type checking for `AeroScript` based on the rules in Figures 2 and 3 and your defined rules.
    - There is no template code for this assignment. Instead you are used the typed calculator (and the slides) to understand how to extend your code.
3. Implement `Junit` tests for your solution. Hint: make a test that passes and one that throws error for each of the case (you can set up the class for the Junit test like in the previous assignment).
4. Write a report (4-5 pages) where you discuss the challenges you faced and choices you made during implementation. The report should cover the entire work (that is also the most important part of assignments 1, 2, and 3), but should have a focus on this work. Be sure to read the report guide on the course page beforehand.
    - Also define the needed typing rules. Why did you define as you did? How does it fit with your implementation?

Small Note on the Java Code The reasoning behind using variables for expressions (e.g. `define=ID`) is to make it easier to retrieve the any information, in turn making it more consistent. Especially when working with multiple elements (expressions, etc.), we should retrieve data using indices (like `ctx.expression(0)`). If you change the order of expressions in the grammar, the logic of the program would no longer be consistent. In summary, using variables makes our interpreter more robust against re-arrangement of syntactic constructs in the grammar.

You only need to extend with a `runtime/TypeCheck.java` to implement the type checker.

If you want to give a typing error, you can add a `error/TypeError.java` and call:
`throw new TypeError("This is a good description of the typing error.")`

Integration with interpreter from A3 You need to integrate the type checker with the interpreter from A3. You need to extend `Main.java` with calling the type checker at the correct time.

[2] The xspace package is a "magic" space package.
`xspace` ensures that there always is a space if the following is a letter and no space to punctuation.

8 What to Submit

You should hand in a `zip` file containing your report (as a PDF) along with the project containing your implementation. Make sure to clean (run `./gradlew clean`) any unnecessary garbarge files before zipping. If your zip-file is very large, you likely did not clean.

9 Where to Submit

In IN2031 we use Devilry, not Canvas, for assignments. Go to

https://devilry.ifi.uio.no

and log in with your UiO username and password. When you are logged in, it should be fairly obvious how to upload the report and code.