Software Engineering

WS 2021/22, Assignment 03

Prof. Dr. Sven Apel
Annabelle Bergum
Sebastian Böhm
Christian Hechtl

Handout: 14.12.2021
Handin: 11.01.2022 23:59 CET

Organizational Section:

- The assignment must be accomplished by yourself. You are not allowed to collaborate with anyone. Plagiarism leads to failing the assignment.
- The deadline for the submission is fixed. A late submission leads to a desk reject of the assignment.
- There are two separate submissions for the two tasks:
  - The submission for task 1 must consist of a *ZIP* archive containing only the project folder (i.e., the folder included in the provided skeleton)
  - The submission for task 2 must consist of a *PDF* including your solutions, your name, and matriculation number. We only accept solutions created in LaTeXwith the template provided on the CMS
- Questions regarding the assignment can be asked in the forum or during tutorial sessions. Please don't share any parts that are specific to your solution, as we will have to count that as attempted plagiarism.

Task 1 [15 Points]

Your task is to implement a type checker for the simple programming language presented in the lecture and its variability-aware type-system. The implementation has to be done in Scala[1] using the project skeleton provided on the CMS.

Grading Your submission will be graded based on the following criteria:

- Your submission must be in the correct format, i.e., a *ZIP* archive with the same layout as the project skeleton (may result in 0 points if violated).
- Your submission must compile, i.e., the command `sbt compile` must succeed (may result in 0 points if violated).
- We run unit tests (the ones provided + additional tests) against your submission. Points will be deducted for each failing test.

Project Skeleton You must implement your solution based on the provided project skeleton. The skeleton has the following structure:

```
build.sbt
project
    build.properties
    plugins.sbt
src
    main
        scala
            Expression.scala
            TypeChecker.scala
    tests
        scala
            *Test.scala
```

The file `Expression.scala` contains an implementation of the programming language constructs and types. The type checker shall be implemented in the file `TypeChecker.scala`. We already provide classes for the various context

[1] `https://www.scala-lang.org/`

objects, as well as the basic structure of the type checker. Your only task is to implement the different type rules (marked with a `TODO`) and possibly some helper functions.

We also provide some basic tests which can be found in the folder `tests`. The file `build.sbt` and the files in the folder `project` contain a build script that we use to build your submission and run the tests.[2] You must not edit these files.

The build script should also enable you to import the project skeleton into any IDE with Scala support (e.g., IntelliJ with the Scala plugin). Scala can be quite sensitive regarding the used language version and things might break if you use the wrong one. For the assignment, we use *Scala 2.13.7* which is also specified in the build script. You can also run the tests included with the project skeleton using the build script. To run all tests execute the command `sbt test`.

SAT Solver For some type rules, you will need a SAT solver to check some preconditions. For this assignment, we use CafeSAT[3] which is a SAT solver implemented in Scala.

The project skeleton already imports all the relevant packages. Checking whether a formula is satisfiable is as simple as this (assuming `a, b, c` are formulas):

```
Solver.solveForSatisfiability((!a || b) && c) match {
  case None => print("Not satisfiable");
  case Some(_) => print("Satisfiable")
}
```

Note how you can use `!`, `||`, and `&&` for *not*, *or*, and *and*. The API does not support implication so you have to encode implication using the other operators. The primitives *true* and *false* are available as `Formulas.True` and `Formulas.False`. Mind the prepended `Formulas` as, otherwise, you might confuse them with the `True` and `False` literals from our programming language!

Type Errors If the type checker detects a type error in the expression it is currently checking it must throw a special exception called `TypeCheckingError`. The constructor of that class requires you to pass the (sub-) expression that is currently checked, as well as the two context objects, and a message describing the type error. This information is used by our tests to determine whether your implementation detected the error correctly. The message is not checked in the tests but can help you during debugging.

[2] `https://www.scala-sbt.org/1.x/docs/`
[3] `https://github.com/regb/cafesat`

Task 2 [10 Points]

We extend the simple programming language from the lecture with language constructs for *λ-abstractions*. Table 1 shows the abstract syntax and variant generation rules for the new constructs. A λ-abstraction e_1 encapsulates an expression e in an anonymous function (i.e., a function without a name) with one parameter x. This function can then be "called" with a second expression e_2 as its argument via the *function application* construct. λ-abstractions also introduce a new *function-type*. A function type consists of the type of the parameter $x : t_1$ and the type of the abstracted expression $e : t_2$ and is written as $t_1 \to t_2$.

	Abstract syntax	**Variant generation**
λ-abstraction:	$(\lambda x.\ e) \quad x \in Id, e \in VExpr$	$[\![(\lambda x.\ e)]\!]_\Psi = (\lambda x.\ [\![e]\!]_\Psi)$
Function application:	$e_1\ e_2 \quad e_1, e_2 \in VExpr$	$[\![e_1\ e_2]\!]_\Psi = [\![e_1]\!]_\Psi\ [\![e_2]\!]_\Psi$
Function type:	$t_1 \to t_2 \quad t_1, t_2 \in Type$	

Table 1: The new language constructs

Examples:

λ-abstraction:	$(\lambda x.\ x < 1) : \mathsf{Num} \to \mathsf{Bool}$
Function application:	$(\lambda x.\ x < 1)\ \mathsf{42} : \mathsf{Bool}$

The new language constructs come with the following new type rules for λ-abstractions (T-Abs) and function application (T-App):

$$\text{T-Abs}\ \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash (\lambda x.\ e) : t_1 \to t_2}$$

$$\text{T-App}\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2}$$

Your task is to make the new type rules T-Abs and T-App *variability-aware*. Use the notation from the lecture (chapter 6 Analysis, slide 73).