CS 164 : Programming Languages and Compilers

PA4: A Type-Checker for the TYPEDFUN Language

Professor: Koushik Sen, prepared by Alex Reinking *Due: 3/21/17*

1. Introduction

In this homework assignment, you will implement the type checker for the TYPEDFUN programming language, described below. You have been supplied with a grammar, the type checking rules, and extensive documentation for the Atrai tree transformation API, which you will use to complete the type checker.

Unlike previous assignments, this type checker will not return a value (like true or false), but will return the tree with the types annotated. We have provided you with a typed version of LET, called TYPEDLET for you to use as a reference.

Additionally, we have extended the Atrai API with functions for cleanly handling groups of children. **Be sure to check the updated slides** in `docs/atrai-slides.pdf` to get up to date.

1.1. Scope and Collaboration.

As before, you may share knowledge of the workings of Atrai with your classmates, as well as discuss general strategies for implementation. However, all code must be written independently, and **you must list all of your collaborators** in your `README`.

1.2. Setup.

You will write your interpreter in Java. The staff has supplied a skeleton for you to complete on GitHub. While logged into your GitHub student account, visit `https://tinyurl.com/cs164s17pa4b`. It will redirect you to GitHub classroom. Accept the assignment, then run `git clone https://github.com/cs164spring17/pa4-<username>.git`, where `<username>` is your GitHub username. The cloned repository will contain all the necessary files for the assignment.

1.3. Software.

To develop locally, you will need the following software installed:

- JDK 1.8 (or newer). Available from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
- Apache Maven 3.3.9 (or newer). Available from `https://maven.apache.org/download.cgi`
- JUnit 4. This dependency will be installed automatically by Maven, but the documentation is available here: `http://junit.org/junit4/`
- (optional) A Java IDE such as IntelliJ IDEA (free community edition; ultimate edition free for students) that can manage Maven projects.

If you are running Mac or Linux, we suggest installing these tools from your package manager, for example Homebrew on Mac, or Apt on Debian/Ubuntu. Your mileage may vary with Chocolatey on Windows or Apt on the Windows Subsystem for Linux (WSL). We recommend, if possible, using an IDE on Windows. Be sure you are familiar with the new features in Java 8, including lambdas and static imports. A fairly comprehensive overview of the new features is available here: `https://leanpub.com/whatsnewinjava8/read`

1.4. **Testing.** This assignment uses Maven to build and run the tests. To do so, simply:

(1) Open a terminal and `cd` into your git repository
(2) Run `mvn clean test`. You should be in the same directory as `pom.xml`

This will take care of building all the source files beneath `src/main/java`, generating the ANTLR sources from `src/main/antlr4`, and executing all the tests beneath `src/test/java`.

Your code will be tested for correctness, meaning strict adherence to the operational semantics. You have been given 15 of the 50 tests that will be used to grade your code.

1.5. **Documentation.** In addition to tests, we have throughly annotated the APIs with Javadoc strings. The compiled javadocs are located in the `docs/javadocs` folder of the git repository. There is also a set of *slides* describing the Atrai API that is located in `docs/slides/atrai-slides.pdf`.

1.6. **Staff Solution.** To help you test your programs, we have set up a website that lets you run a TYPEDFUN program with the staff's code. The URL of the website is: `http://alexreinking.com/cs164/index.php`.

1.7. **Running your code.** You can run your own code by either adding tests to `src/test/java/atrai/interpreters/TYPEDFUN/TypedFunCheckerPublicTest.java` and running `mvn clean test` or by running `mvn -DskipTests clean package` and executing the following command:

```
java -jar target/atrai-1.0.jar [parse|interpret]
atrai.interpreters.TYPEDFUN.TypedFunChecker <file name>
```

This command reads a TYPEDFUN program from a file. Then, if you pass `parse` as the first argument, it will dump the untyped tree using the APIs you have been provided. If you instead pass `interpret` as the first argument, it will print the result of parsing and interpreting the whole program. You can also run the provided TYPEDLET typechecker by replacing `TYPEDFUN.TypedFunChecker` with `TYPED.TypedLetInterpreter` in the command.

During debugging, you might find it helpful to pass the flag `-Ddebug1=true` or additionally `-Ddebug2=true` to the `java` command. You can also set these in the JVM flags area of your IDE's run-command dialog. These flags produce extra debugging output. Passing the `debug1` flag shows traces of the interpreter execution when rules match. Additionally passing `debug2` will show output for rules that don't match, helping you find bugs when a rule fails to match.

1.8. **Submission.** To submit, simply push to origin with `git push origin`. You must tag a commit with `pa4final` using the `git tag` command so we know which commit to grade. If you do not tag a commit, we will use the latest before the deadline, which could potentially be worse than a better, but slightly late, submission.

2. Files and Directories

This is a standard Maven project, so you will find the following files:

- `docs/javadocs` - This is where the pre-compiled Javadocs are.
- `docs/slides` - This folder contains the relevant lecture notes and slides on Atrai.

- `pom.xml` - This is the Maven project file. There should be no need to modify it.
- `src/main/antlr4` - This is where all the grammar files are located, including the grammar for TYPEDFUN.
- `src/main/java` - This is where all the source is. It contains several packages:
  - `atrai.interpreters` - This package contains the example interpreters and some common utilities. You will place your code in `TYPEDFUN/TypedFunChecker.java`. You may add more files to this directory.
  - `atrai.core` - This package contains a library for implementing tree transformations. All the other interpreters are built on top of this. Read the Javadocs and look at the example interpreters to understand its use.
  - `atrai.antlr` - This package contains generic utilities for processing ANTLR parse trees.
- `src/test/java/atrai/interpreters/TYPEDFUN/TypedFunCheckerPublicTest.java` - This file contains 15 test cases. The rest of the tests will go here during grading, so expect this file to be overwritten. Feel free to add more tests to this file.

The project should compile as-is, but will always reject inputs, and will not pass any test cases. We will overwrite every existing file, except for `TypedFunChecker.java` and any new files you add in that same directory.

3. TYPEDFUN vs. FUN

TYPEDFUN has essentially the same syntax as FUN, but type annotations are added after declarations, function parameters, and functions (to indicate return types). This is demonstrated in the following factorial function.

```
let fact : (int) -> int = fun(x : int) : int =
  let prod : int = 1
   in { while x > 1 do {
          prod = prod * x;
          x = x - 1
        }; prod }
in fact(4)
```

The objective of the type checker is to add type information to every `expr` subtree that appears in the parse tree. The way this will be done is by wrapping every expression in a new `typed` node. The general form is `(%FUN <id> typed <expr> <type>%)`. Here, the `<id>` is the same as the `<id>` of its child. The type is an instance of either `PrimitiveTypeValue` or `FunctionTypeValue`, both of which are provided to you. The expected output for the `prod = prod * x` part of the above code snippet is shown in listing 3.1. Similarly, the type of the `fun(x : int): int = ...` expression would be given as `(INT)->INT`.

4. Atrai Update

In the previous assignment, we introduced the `iterate()` function to process the children of a node. Since then, we have improved the interface. It is now possible to use `@*_` in a pattern and `$*n` in a replacement to capture or insert multiple children simultaneously. As an example, you can reverse the order of execution in a sequence in FUN with the following code:

```
(%FUN 29 expr
  (%FUN 30 iden prod%) =
  (%FUN 31 expr
    (%FUN 32 expr
      (%FUN 33 iden prod%)
    %)
    *
    (%FUN 34 expr
      (%FUN 35 iden x%)
    %)
  %)
%)
```

A. Original untyped tree

```
(%FUN 29 typed
  (%FUN 29 expr
    (%FUN 30 iden prod%) =
    (%FUN 31 typed
      (%FUN 31 expr
        (%FUN 33 typed
          (%FUN 33 iden prod%) INT
        %)
        *
        (%FUN 35 typed
          (%FUN 35 iden x%) INT
        %)
      %)
      INT
    %)
  %)
  INT
%)
```

B. Type-annotated untyped tree

LISTING 3.1. Example for adding types to a parse tree. The result of your transformer should match 3.1b

```
transformer.addTransformer("(%FUN @_ expr { (%FUN @_ exprseq @*_%) }%)",
  (c, E) -> {
    Object[] children = (Object[])c[3];
    Collections.reverse(Arrays.asList(children));
    c[3] = children;
  }, "(%FUN $1 expr { (%FUN $2 exprseq $*3%) }%)");
```

This transformer would convert `(%FUN 1 expr { (%FUN 2 exprseq 1 ; 2 ; 3 %)} %)` to `(%FUN 1 expr { (%FUN 2 exprseq 3 ; 2 ; 1 %)} %)`. Note that the new operators may only appear once in any given subtree. They greedily match *all* remaining children starting at their position, which could be zero.

5. TYPEDFUN SYNTAX

You will implement the TYPEDFUN language, which has arithmetic and logical expressions, assignment, branching, loops, and functions. The grammar for TYPEDFUN is reproduced below.

```
1 prog: expr
2
3 expr: expr '(' expr? ')'
4     | expr ('*' | '/' | '+' | '-') expr
5     | expr ('=='|'!='|'>'|'<'| '>=' | '<=') expr
6     | iden '=' expr
7     | 'let' decllist 'in' expr
8     | 'if' expr 'then' expr 'else' expr
```

```
9       | 'while' expr 'do' expr
10      | 'fun' '(' iden ':' type ')' ':' type '=' expr
11      | 'fun' '(' ')' ':' type '=' expr
12      | 'print' expr
13      | 'null'
14      | num
15      | iden
16      | string
17      | '{' exprseq '}'
18      | '(' expr ')'
19
20  decllist: decl (',' decl)*;
21  decl: iden ':' type '=' expr;
22  exprseq: expr (';' expr)*;
23
24  type: ID
25      | '(' ')' '->' type
26      | '(' type ')' '->' type
27
28  num:[0-9]+;
29  iden: [a-zA-Z_][a-zA-Z0-9_]*;
30  string: "[^"]*";
```

6. Typing Rules for TYPEDFUN

Since the operational semantics to TYPEDFUN are identical to those for FUN by simply ignoring the type annotations, we present only the typing inference rules here. Here, O is a typing context, and the types in the language are the *value* types INT , BOOL , STRING , NULL , and the function types $() \to T$ and $(T_1) \to T_2$, where T, T_1, T_2 all refer to other types.

6.1. **Literals.** The literals are all immediately their own types. Referring to an identifier looks up the established type in the current context.

$$\text{NUM}\frac{}{O \vdash num : \mathsf{INT}} \qquad \text{STRING}\frac{}{O \vdash string : \mathsf{STRING}}$$

$$\text{NULL}\frac{}{O \vdash \texttt{null} : \mathsf{NULL}} \qquad \text{IDEN}\frac{T = O(id)}{O \vdash id : T}$$

6.2. **Simple Expressions and Sequencing.** Comparisons and arithmetic always result in `int` , while concatenation always ends in a `string` . Unlike FUN, we only allow ints to be concatenated to strings, and strings to be concatenated to each other.

$$\text{LOGIC}\frac{O \vdash e_1 : \mathsf{INT} \qquad O \vdash e_2 : \mathsf{INT} \qquad \mathsf{op} \in \{==, !=, <, >, <=, >=\}}{O \vdash e_1 \ \mathsf{op} \ e_2 : \mathsf{BOOL}}$$

$$\text{ARITH}\frac{O \vdash e_1 : \mathsf{INT} \qquad O \vdash e_2 : \mathsf{INT} \qquad \mathsf{op} \in \{+, -, *, /\}}{O \vdash e_1 \ \mathsf{op} \ e_2 : \mathsf{INT}}$$

$$\text{CONCAT-L}\frac{O \vdash e_1 : \mathsf{STRING} \qquad O \vdash e_2 : \mathsf{INT}}{O \vdash e_1 + e_2 : \mathsf{STRING}}$$

$$\text{CONCAT-R}\frac{O \vdash e_1 : \ \mathsf{INT} \qquad O \vdash e_2 : \ \mathsf{STRING}}{O \vdash e_1 + e_2 : \ \mathsf{STRING}}$$

$$\text{CONCAT-S}\frac{O \vdash e_1 : \ \mathsf{STRING} \qquad O \vdash e_2 : \ \mathsf{STRING}}{O \vdash e_1 + e_2 : \ \mathsf{STRING}}$$

The sequencing construction needs all of its components to be well-typed, but only returns the last one.

$$\text{SEQ}\frac{O \vdash e_1 : T_1 \qquad \cdots \qquad O \vdash e_n : T_n}{O \vdash \{e_1; \cdots ; e_n\} : T_n}$$

Assignment is only valid when the already-known variable type and expression type match.

$$\text{ASSN}\frac{T = O(id) \qquad O \vdash e : T}{O \vdash id = e : T}$$

6.3. **Print.** Print evaluates to its argument, and so matches its type. We don't model the side effect in the type system since it does not affect execution.

$$\text{PRINT}\frac{O \vdash e : T}{O \vdash \ \texttt{print}\ e : T}$$

6.4. **Lets and Branching.** Here, we define the rules for LET inductively by reducing the many-declarations case to the one-declaration case. One could also write a full explicit rule as was done for FUN.

$$\text{LET-BASE}\frac{O \vdash e_1 : T_1 \qquad [id = T_1]O \vdash e_2 : T_2}{O \vdash \ \texttt{let}\ id : T_1 = e_1\ \texttt{in}\ e_2 : T_2}$$

$$\text{LET-INDUCT}\frac{O \vdash \ \texttt{let}\ id_1 : T_1 = e_1\ \texttt{in}\ (\ \texttt{let}\ id_2 : T_2 = e_2, \ldots, id_n : T_n = e_n\ \texttt{in}\ e) : T}{O \vdash \ \texttt{let}\ id_1 : T_1 = e_1, id_2 : T_2 = e_2, \ldots, id_n : T_n = e_n\ \texttt{in}\ e : T}$$

Conditional branches only work with a boolean condition, and require the types of their branches to match.

$$\text{IF}\frac{O \vdash e_1 : \ \mathsf{BOOL} \qquad O \vdash e_2 : T \qquad O \vdash e_3 : T}{O \vdash \ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T}$$

While-loops always evaluate to null, but still require their conditions to be booleans and their bodies to be well-typed.

$$\text{WHILE}\frac{O \vdash e_1 : \ \mathsf{BOOL} \qquad O \vdash e_2 : T}{O \vdash \ \texttt{while}\ e_1\ \texttt{do}\ e_2 : \ \mathsf{NULL}}$$

6.5. **Functions.** The function declarations are both kinds of literals, but are not considered values. They require that their bodies typecheck under the assumption that their arguments are well-typed.

$$\text{FUN}\frac{[id = T_1]O \vdash e : T_2}{O \vdash \ \texttt{fun}\ (id : T_1) : T_2 = e : (T_1) \to T_2}$$

$$\text{FUN-NA}\frac{O \vdash e : T}{O \vdash \ \texttt{fun}\ () : T = e : () \to T}$$

To apply a function, we need to be sure that the arguments type-check. The whole expression results in the return type of the function being called.

$$\text{APP}\frac{O \vdash e_2 : T_2 \qquad O \vdash e_1 : (T_2) \to T_1}{O \vdash e_1(e_2) : T_1}$$

$$\text{APP-NA}\frac{O \vdash e_1 : () \to T}{O \vdash e_1() : T}$$

7. Grading - 60 points

The rubric is as follows:

- 50 points. One point per automated test, including the 15 provided tests.
- 5 points. General code cleanliness and style. Clear variable names, consistent spacing and brace conventions, no monolithic functions, etc.
- 5 points. README containing a description of any challenges you faced while completing this assignment.