Compiler Construction
Summer Term 24
May 16
Lübke/Schupp

Type Checker

Deadline : 06.06.24 08:00 AM (2024-06-06T08:00:00+02:00)

How to complete a project successfully?

Follow the rules as described in the Lecture!

GitLab at TUHH[1] is used as the submission system in this course. Your GitLab credentials are the same as for StudIP. If you cannot log into GitLab with your StudIP login and password, contact the teaching assistant immediately. All your code must be pushed to your group repository provided to you by the teaching assistant. Remember, that each group member needs to demonstrate their contributions through code commits. You can update your solution until the deadline. The latest commit before the deadline is considered by default. If you want your late submission to be graded for penalized maximal number of possible points, the **whole** group must send an email to the teaching assistant **before** the interview day.

How to get additional information?

You are encouraged to discuss past and present project sheets with the teaching assistants. Either approach the teaching assistant during the exercise session, or visit us during the weekly office hours. We are also available through e-mail or on the StudIP forum. We try to reply as quickly as possible and in general, you should get a reply the next weekday, but we cannot guarantee this.

[1] https://collaborating.tuhh.de/

Type Checker

Static type systems are some of the most successful and widely accepted formal tools to help write correct programs. The common interpretation of types in programming languages is a combination of the following:

a) A type is a set of values.

b) The values of a type share an implementation at run-time.

Further, type systems are often categorized in two dimensions: **static** vs. **dynamic**, and **strong** vs. **weak**. A **static** type system is executed at **compile time**, i.e., before the program is run, e.g., during compilation. A **dynamic** type system on the other hand is executed at **run time**, i.e., when the program is executed. One advantage of **static** typing is that it checks that a program is **type safe**, i.e., will not generate any type errors during execution, where a type error is a data flow between a type-wise incompatible value source and sink (`int x = "foo";`). Once a program is proven to be type safe, certain checks can be eliminated, thereby gaining performance. The advantage of **dynamic** typing is often said to be that incorrectly typed can still be executed during development, and that programs do not have to be annotated with types, thereby being less obscured. However, modern programming languages (Haskell, OCaml,. . .) with static typing use type inference and therefore do not require many type annotations anymore. **Strong** versus **weak** generally refers to how easy it is to claim a value is of a different type or, similarly, how close the type of a source has to match the type of the sink. No generally accepted formal definitions of the terms exist. This exercise is based on course materials taught by Aarne Ranta[2] modified for our purposes.

Tasks

Your task is to write a type checker for a simple, C-like language. Work with the (enclosed) "CPP.cf" file and read through the explanations below. Do **not** work with your own grammar from the previous task!

Task 1 Implement the base part of the type checker. The base does not include while, do-while, for, and if statements, binary arithmetic operators `+`, `-`, `*`, `/`, and arithmetic comparison operators `<`, `>`, `<=`, `>=`. Work on this task in groups.

[2] http://www.cse.chalmers.se/edu/year/2018/course/DAT151

Task 2 Look up the file `ASSIGNMENTS` in your GitLab repository and implement the parts of the language corresponding to your number:

(1) do-while-loops, `+`, and `<`

(2) while-loops, `-`, and `>`

(3) for-loops, `*`, and `<=`

(4) if-statements, `/`, and `>=`

Work on your assignments **individually**. The commit with the implementation must be done by the group member assigned to the corresponding language construct.

Summary

The objective of this project is to write a type checker for a fragment of the C++ programming language. The type checker should check the program. At type checking failure, a type error should be reported. The recommended implementation is via a BNF grammar processed by the BNF Converter (BNFC) tool. The syntax tree created by the parser should then be processed further by a program using the skeleton generated by BNFC. The fragment of C++ covered is smaller than in Exercise 1. You should use the grammar `CPP.cf` that came in the zip file with this PDF.

Method

In the type checker, the recommended procedure is two passes:

a) build a symbol table with all function types

b) type check the code by using this symbol table

As a very first step, run `bnfc` on the provided *CPP.cf* file (without the `-m` switch because the makefile is provided by us). Apart from generating a parser, this also generates a *skeleton* you can use as a starting point. For C++ the skeleton consists of *SKELETON.H* and *SKELETON.C*, for Haskell it is called *SkelCPP.hs*, and both work slightly differently. Rename the generated file to something more suitable (e.g., *TypeChecker*, this is especially important for C++ because the makefile deletes the skeleton files so they don't interfere with your actual code).

The C++ code uses the *Visitor Pattern*[3] to traverse the AST. For each of the `visit*` functions there is a corresponding syntax element, sub-trees are visited using the `accept` function. The Haskell code relies on the algebraic data type defined for the AST, and only top-level syntax elements have their own `trans*` function (kind of similar to the C++ `visit*` functions). For traversing sub-trees, pattern matching is used, which showcases why Haskell is so well-suited for such tasks – the code directly reflects the syntax specification! Even if you're planning to work with C++, generating the Haskell skeleton and

[3]https://en.wikipedia.org/wiki/Visitor_pattern

having a brief look at it is worth the time (at least if you know at least a little Haskell), because it may give some insight into how the visitor pattern is used.

Again, we recommend a gradual (i.e., feature-by-feature) approach to building the type checker. It is also a good idea to define your own test cases for debugging.

The type checker

Types

The five built-in types

- `int`
- `double`
- `bool`
- `void`
- `exception`

and user-defined structure types are taken into account. Every expression has one of these types. The `exception` type is special because there are no literals for it; you can think of it as a `struct` with no fields. Yet, `structs` can inherit from it to create user-defined exceptions.

Types of functions in the symbol table can be represented in any way that stores their argument and return types. For instance, the function header

```
int f (double x, bool b)
```

can create a symbol table entry

```
f |-> ([double, bool], int)
```

mapping the name f to a pair whose first component is the list of argument types and the second component is the return type.

Programs

A program is a sequence of definitions. A program may also contain comments and preprocessor directives, which are just ignored by the parser (see below). A program must have a function `main` that takes no arguments and returns an `int`. But this need not be checked in the type checker.

Definitions

a) A function definition has a type, a name, an argument list, and a body. Example:

```
int foo(double x, int y)
{
  return y + 9 ;
}
```

b) A structure definition consists of a name and a list of declarations. Example:

```
struct data {
  int i;
  double d;
} ;
```

Optionally, a structure can inherit fields from other structures:

```
struct data_ex : exception {
  int i;
  double d;
};

struct more : data_ex {
  bool b;
};
```

Notice that the semicolon is obligatory in the end of a structure definition.

Typing rules

The same function or structure name may be used in at most one function or structure definition respectively. All return statements in a function body must return an expression whose type is the return type of the function or nothing if the function is `void`. You don't need to check that there actually is a return statement (you can do this optionally). All fields within one structure must have different names. Structures can only inherit from `exception` or other structures. Fields from the inherited structure and the newly defined structure are treated equally. This includes that a structure definition may not redeclare/overwrite fields of its parent structure.

Argument lists, declarations, and function bodies

An argument list is a comma-separated list of argument declarations. It is enclosed in parentheses `(` and `)`. An argument declaration has a type and an identifier, for instance

```
int x
```

Notice that argument declarations with multiple variables (`int x, y`) are not included. A declaration that occurs as a statement (as shown below), can also have multiple variables. But it must have at least one variable. A function body is a list of statements enclosed in curly brackets { and }.

Typing rules

An argument list may use each name only once. An argument cannot be of type `void`.

Statements

Any expression followed by a semicolon ; can be used as a statement. Any declaration followed by a semicolon ; can be used as a statement. Declarations consist of a type and one or more variables with an optional initializing expression, e.g.:

```
int i ;
double x, y = 5, z ;
```

Typing. The initializing expression must have the declared type. Variables cannot be of type `void`.

Statements returning an expression, for example

```
return i + 9 ;
```

Typing. The type of the returned expression must be the same as the return type of the function in which it occurs. In case of void functions, the return statement has no argument.

While loops, with an expression in parentheses followed by a statement, for example:

```
while (i < 10) ++i ;
```

Do-while loops, with an expression in parentheses after the loop body, for example:

```
do ++i; while (i < 10) ;
```

Typing. The expression must have type `bool`.

For loops, three expressions in parentheses followed by a statement.

```
for ( i = 0 ; i != 10 ; ++i ) k = k + i ;
```

Typing. The second expression must have type `bool`.

Conditionals: `if` with an expression in parentheses followed by a statement, `else`, and another statement. Example:

```
if (x > 0) return x ; else return y ;
```

Typing. The expression must have type bool.

(No else-less if statements.)

Try-catch: `try` followed by a statement, and by `catch` with an argument declaration in parentheses and a statement. Example:

```
try
  foo();
catch(exception e)
  bar(e);
```

Typing. The type of the argument declaration must be an exception type. `exception` is an exception type, and any structure that inherits from an exception type is also an exception type. The scope of the declared variable is limited to the statement following the `catch`.

Blocks: any list of statements (including empty list) between curly brackets. For instance,

```
{
  int i = 2 ;
  {
  }
  i++ ;
}
```

Typing rules

A variable may only be declared once on the same block level. The parameters of a function have the same level as the top-level block in the body.

Expressions

The following table gives the expressions, their precedence levels, and associativity types. The arguments in a function call can be expressions of any level. Otherwise, the subexpressions are assumed to be one precedence level above of the main expression.

Note. The table is not guaranteed to be exactly the same as in the C++ standard.

level	expression forms	assoc	explanation	type
15	literal		atomic expressions	literal type
15	identifier		variable	declared type
14	`f(e,...,e)`	left	function call	return type
14	`e.id`	left	structure projection	type of the field `id`
14	`e++`, `e--`	left	in/decrement	operand type
13	`++e`, `--e`	right	in/decrement	operand type
13	`+e`, `-e`	right	unary plus/minus	operand type
12	`e*e`, `e/e`	left	multiplication, division	operand type
11	`e+e`, `e-e`	left	addition, subtraction	operand type
9	`e<=>e`	left	three-way comparison	integer
9	`e<e`, `e>e`, `e>=e`, `e<=e`	left	comparison	bool
8	`e==e`, `e!=e`	left	(in)equality	bool
4	`e&&e`	left	conjunction	bool
3	`e||e`	left	disjunction	bool
2	`e=e`	right	assignment	type of LHS
2	`e ? e : e`	right	conditional	branch operand type
1	`throw e`	right	throw exception	operand type

Typing rules

Integer, double, and boolean literals have their usual types. Variables have the type declared in the nearest enclosing block. A variable must be declared before it is used in an expression. The arguments of a function call must have types corresponding to the argument types of the called function. The number of arguments must be the same as in the function declaration (thus the C++ default argument rule is not applied). Notice that only identifiers are used as functions. Structure projection applies to an expression of a structure type and returns its field content. Increments and decrements only apply to numeric variables or structure fields. Numeric operations apply to two numeric operands of the same type and return the type of the operands. Comparison operators apply to two numeric operands of the same type and return `bool` or `int` in case of tree-way comparison. (In)equality applies to two operands of any type and returns `bool`. Conjunction and disjunction apply to operands of type `bool` and return `bool`. Conditional applies to a `bool` condition, and two branch expressions of the same type and returns the type of the branch expressions. Assignments have the same type as the left-hand-side variable. Notice that only variables and structure projections are used as left-hand-sides and there are no qualified constants or template instantiations. The operand of a `throw` expression must be of an exception type. `exception` is an exception type, and any structure that inherits from an exception type is also an exception type.

Literals

We include integer literals and floating point literals. There are also two boolean literals, `true` and `false`. Notice that the names true and false were not specified as literals in Exercise 1, so you probably treated them as identifiers.

Identifiers

An identifier is a letter followed by a list of letters, digits, and underscores.

Comments

There are three kinds of comments.

- anything between tokens `/*` and `*/`
- anything from token `//` to the end of a line or the file
- anything from token `#` to the end of a line or the file (preprocessor directive)

Comments cannot be nested.

Implicit type conversion

In this task you are required to implement implicit type conversion from `int` to `double`. This means that whenever a `double` expression is expected, an `int` expression can appear, for example, `1 + 1.0` is a valid `double` expression.

Format

Input and output

The type checker must be a program called `compiler`, which reads the standard input and prints its output to the standard output. The output at success must be just `OK`. The output at failure is a `TYPE ERROR` or a `SYNTAX ERROR`, depending on the phase at which the error occurs. The exit code must be 0 or 1 in the case of success of failure respectively. These error messages should also give some useful explanation, but we do not specify their format.

The easiest way to produce the proper format is to use the (enclosed) ready-made templates.

Example of success

Source file

```
int main ()
{
  return 0 ;
}
```

Running the type checker

```
% ./compiler < good.cc
OK
```

Example of failure

Source file

```
int main ()
{
  return 0.0;
}
```

Running the type checker

```
% ./compiler < bad.cc
TYPE ERROR
return expression: expected int, found double
```

Success criteria

Your type checker must pass the testsuite available on INGInious[4] accessible from the TUHH network or through the TUHH VPN. You can log in with you usual TUHH user account (the one you also use for StudIP).

After you log in, if you haven't done so already, click the *Course List* and register for the *Compiler Construction* course. When you're registered, click *My Courses* in the side bar, select *Compiler Construction*, and then the *Type*

[4] https://inginious.sts.tuhh.de

Checker task. Use the *Choose File* button to select the file you want to submit and then click *Submit*. Your submission must be a ZIP archive containing your sources and the grammar file excluding the automatically generated BNFC files and the compiled compiler. The Haskell and C++ submissions must be compilable with `stack build` and `clang++-9` respectively.

When all tests are run you'll see a red or green box (depending on whether your submission already passes all tests). In this box, there are two sections, *debug info* and *test output*. Most of the time, you can ignore the *debug info*, but please always include it when you encounter problems with INGInious and ask for help. The *test output* will tell you how many tests your submission already passed, and gives hints on how to pass any remaining tests.

Compilation together with passing all tests takes a few minutes and may take longer depending on the server load. Please, be patient and do not resubmit the same solution to the same problem. If you experience timeouts, please contact the teaching assistant (please include debug info if shown).

Note that the test suite is built in a gradual manner starting from the simplest cases and basic language constructs ending at more complicated ones. The hint shown upon a test failure relates to the language construct in focus and is relevant only assuming that all the previous ones work correctly. The tests for individual subtasks are based on the base, hence we suggest that you start working on Task 2 only after completing Task 1.

Submission

Submit your solution via GitLab[5]. If you have any problems getting the test program to run, or if you think that there is an error in the test suite, contact the teaching assistant (please include debug info if shown).

Grading

You can earn up to 30 points in this project phase.

5 points are awarded for the readability of you code. This includes but is not limited to: sensible naming of variables/functions/..., logical structure (division into modules or sections in the file(s)), length of functions, depth of nesting of functions, avoidance of code duplication, the use of comments, as well as the formatting. In essence: make sure someone else can quickly understand your code without any further explanations.

15 points are awarded for the group's portion of the work (i.e., task 1), split evenly between definitions, statements, and expressions (5 points each).

10 points are awarded for the individual portion of the work (i.e., task 2), split evenly between statements and expressions (5 points each).

[5] https://collaborating.tuhh.de