COSC 4780/5010 - Principles of Programming Languages
Finley McIlwaine
March 1, 2022

Midterm

Due before midnight (11:59 PM MST) on March 11, 2022

The midterm is two written-response questions followed by a large-scale programming project in which you will implement a type checker for a small fragment of the C++ language. For your submission on WyoCourses, submit a PDF named `midterm.pdf` containing your written answers and the URL to your generated midterm project repository.

1 Questions

1. (10 Points) Given the inference rules from section 2.2, build a proof tree concluding with the judgement:

$$\varnothing \vdash_s int\ x;\ x = x + 1; \Rightarrow x : int$$

Recall that:

- $\varnothing$ is the empty environment.
- $\vdash_s$ is the typing relation for statements. It maps environments and statements to updated environments.
- $\vdash_e$ is the typing relation for expressions. It maps environments and expressions to types.

2. (20 Points) In C++, `for` loops typically have the form

$$\texttt{for}(initialization;\ condition;\ change)\ statement$$

where the *initialization* part declares, defines, or initializes a variable. If a variable is initialized or declared, it should only be in scope inside the loop body. For example:

```
for (int i = 1; i < 101; i++)
{
    printInt(i);
}
double i = 3.14; // Legal, no longer in loop.
```

Write an inference rule for type checking this kind of `for` loop that extends the statement typing relation ($\vdash_s$). The *initialization* part can be an arbitrary statement. The *condition* expression should be required to have type *bool*. Nothing should be required for the type of the *change* expression.

2 Project

Your task is to develop a type checker for a fragment of the C++ programming language. The type checker should be built as a stage in the compiler pipeline, where the lexing and parsing is done by the BNFC generated lexer and parser. The LBNF grammar file is included in the starting repository, so you don't need to worry about constructing the grammar yourself. All you need to do is write the type checking logic. As usual, you can choose between Haskell and Java for an implementation language.

2.1 Project Layout

Begin the project by accepting the assignment on GitHub at this link: `https://classroom.github.com/a/5tcE0VZM`. Clone the repository that is generated for you and include the GitHub link in your submission on WyoCourses. The repository includes `haskell` and `java` directories containing the starter code for the implementation language you choose. Both of these are already in a state where the parsing is complete and the abstract syntax tree is ready to be type checked, but feel free to rearrange things and break things out into different modules/classes/files.

Both directories also contain a `Makefile` that you should use to build and test the compiler. Here is a description of what each of the `make` rules do:

- `make bnfc`: Generate the parser and lexer described by the `CPP.cf` grammar file using BNFC.
- `make parse`: Run the generated parser, which expects input on standard input. Good for sanity checking program syntax if you want to.
- `make compile`: Run your compiler, which also expects its input on standard input. Good for testing your type checker for a given chunk of source code.
- `make test`: Run your compiler on the test suite. Once all of these tests pass, you can feel confident in your submission.
- `make test_grad`: Run the graduate-level tests along with the previous tests. These require a modified type system to be implemented in your compiler that is described later.

The test suite is a custom Haskell program that runs your compiler on each of the test programs in the test directory and checks for success/failure, so email Finley if you have any issues running it.

2.2 Type System Specification

What follows is the formal type system that your type checker needs to implement. Reviewing the test files to see which programs are expected to pass and fail type checking in this system may also help you get an understanding of the type system. See below for descriptions of the notation and meta-expressions.

$$\frac{\varnothing \vdash_d d_1 \ldots d_n \Rightarrow \Gamma \quad \Gamma \vdash_p d_1 \ldots d_n}{\varnothing \vdash_p d_1 \ldots d_n}\ \textsc{PInit}$$

$$\frac{\Gamma \vdash_p d_1 \quad \Gamma \vdash_p d_2 \ldots d_n}{\Gamma \vdash_p d_1 \ldots d_n}\ \textsc{PDefs} \qquad\qquad \frac{}{\Gamma \vdash_p \ominus}\ \textsc{PEDefs}$$

$$\frac{\Gamma \triangleright \varnothing \vdash_d T_1\ id_1 \ldots T_n\ id_n \Rightarrow \Gamma' \quad \Gamma' \vdash_s stms \Rightarrow \Gamma'' \quad id == \text{“}main\text{”} || returnsCheck(stms, T)}{\Gamma \vdash_p T\ id(T_1\ id_1, \ldots, T_n\ id_n)\ \{\ stms\ \}}\ \textsc{PDef}$$

$$\frac{\Gamma \vdash_d d_1 \Rightarrow \Gamma' \quad \Gamma' \vdash_d d_2 \dots d_n \Rightarrow \Gamma''}{\Gamma \vdash_d d_1 \dots d_n \Rightarrow \Gamma''}\,\text{DDefs} \qquad \frac{}{\Gamma \vdash_d \varnothing \Rightarrow \Gamma}\,\text{DEDefs}$$

$$\frac{\Gamma \vdash_d (T_1 \to \dots \to T_n \to T)\ id \Rightarrow \Gamma'}{\Gamma \vdash_d T\ id(T_1\ id_1, \dots, T_n\ id_n)\ \{\ \dots\ \} \Rightarrow \Gamma'}\,\text{DFun}$$

$$\frac{id_1 \notin dom(head(\Gamma)) \quad \dots \quad id_n \notin dom(head(\Gamma, id_1 : T, \dots, id_{n-1} : T))}{\Gamma \vdash_d T\ id_1 \dots id_n \Rightarrow \Gamma, id_1 : T, \dots, id_n : T}\,\text{DDecl}$$

$$\frac{\Gamma \vdash_d T_1\ id_1 \Rightarrow \Gamma_1 \quad \dots \quad \Gamma_{n-1} \vdash_d T_n\ id_n \Rightarrow \Gamma_n}{\Gamma \vdash_d T_1\ id_1 \dots T_n\ id_n \Rightarrow \Gamma_n}\,\text{DParams}$$

$$\frac{\Gamma \vdash_s s_1 \Rightarrow \Gamma' \quad \Gamma' \vdash_s s_2 \dots s_n \Rightarrow \Gamma''}{\Gamma \vdash_s s_1 \dots s_n \Rightarrow \Gamma''}\,\text{SSeq} \qquad \frac{}{\Gamma \vdash_s \ominus \Rightarrow \Gamma}\,\text{SESeq}$$

$$\frac{\Gamma \vdash_e e : T}{\Gamma \vdash_s e; \Rightarrow \Gamma}\,\text{SExp} \qquad \frac{\Gamma \vdash_d T\ id_1 \dots id_n \Rightarrow \Gamma'}{\Gamma \vdash_s T\ id_1 \dots id_n; \Rightarrow \Gamma'}\,\text{SDecl}$$

$$\frac{\Gamma \vdash_d T\ id \Rightarrow \Gamma' \quad \Gamma' \vdash_e e : T}{\Gamma \vdash_s T\ id = e; \Rightarrow \Gamma'}\,\text{SInit} \qquad \frac{\Gamma \vdash_e e : returnType(head(\Gamma))}{\Gamma \vdash_s return\ e; \Rightarrow \Gamma'}\,\text{SReturn}$$

$$\frac{returnType(head(\Gamma)) \equiv void}{\Gamma \vdash_s return; \Rightarrow \Gamma}\,\text{SReturnVoid} \qquad \frac{\Gamma \vdash_e e : bool \quad \Gamma \triangleright \varnothing \vdash_s stm \Rightarrow \Gamma \triangleright \Gamma'}{\Gamma \vdash_s while(e)\ stm \Rightarrow \Gamma}\,\text{SWhile}$$

$$\frac{\Gamma \triangleright \varnothing \vdash_s s_1 \dots s_n \Rightarrow \Gamma \triangleright \Gamma'}{\Gamma \vdash_s \{s_1 \dots s_n\} \Rightarrow \Gamma}\,\text{SBlock}$$

$$\frac{\Gamma \vdash_e e : bool \quad \Gamma \triangleright \varnothing \vdash_s stm1 \Rightarrow \Gamma \triangleright \Gamma' \quad \Gamma \triangleright \varnothing \vdash_s stm2 \Rightarrow \Gamma \triangleright \Gamma''}{\Gamma \vdash_s if(e)\ stm1\ else\ stm2 \Rightarrow \Gamma}\,\text{SIfElse}$$

$$\frac{e \in \mathbb{B}}{\Gamma \vdash_e e : bool}\,\text{EBool} \qquad \frac{e \in \mathbb{Z}}{\Gamma \vdash_e e : int}\,\text{EInt}$$

$$\frac{e \in \mathbb{R}}{\Gamma \vdash_e e : double}\,\text{EDouble} \qquad \frac{e \in \mathcal{S}}{\Gamma \vdash_e e : string}\,\text{EString}$$

$$\frac{id : T \in \Gamma}{\Gamma \vdash_e id : T}\,\text{EId}$$

$$\frac{id : T_1 \to \cdots \to T_n \to T \in \Gamma \quad \Gamma \vdash_e e_1 : T_1 \quad \cdots \quad \Gamma \vdash_e e_n : T_n}{\Gamma \vdash_e id(e_1, \ldots, e_n) : T}\ \text{EApp}$$

$$\frac{\Gamma \vdash_e id : T \quad T \in \{int, double\}}{\Gamma \vdash_e id{+}{+} : T}\ \text{EPIncr} \qquad \frac{\Gamma \vdash_e id : T \quad T \in \{int, double\}}{\Gamma \vdash_e id{-}{-} : T}\ \text{EPDecr}$$

$$\frac{\Gamma \vdash_e id : T \quad T \in \{int, double\}}{\Gamma \vdash_e {+}{+} id : T}\ \text{EIncr} \qquad \frac{\Gamma \vdash_e id : T \quad T \in \{int, double\}}{\Gamma \vdash_e {-}{-} id : T}\ \text{EDecr}$$

$$\frac{\Gamma \vdash_e e_1 : T \quad \Gamma \vdash_e e_2 : T \quad T \in \{int, double\}}{\Gamma \vdash_e e_1 * e_2 : T}\ \text{ETimes} \qquad \frac{\Gamma \vdash_e e_1 : T \quad \Gamma \vdash_e e_2 : T \quad T \in \{int, double\}}{\Gamma \vdash_e e_1 / e_2 : T}\ \text{EDiv}$$

$$\frac{\Gamma \vdash_e e_1 : T \quad \Gamma \vdash_e e_2 : T \quad T \in \{int, double, string\}}{\Gamma \vdash_e e_1 + e_2 : T}\ \text{EPlus} \qquad \frac{\Gamma \vdash_e e_1 : T \quad \Gamma \vdash_e e_2 : T \quad T \in \{int, double\}}{\Gamma \vdash_e e_1 - e_2 : T}\ \text{EMinus}$$

$$\frac{\Gamma \vdash_e e_1 : T \quad \Gamma \vdash_e e_2 : T \quad T \in \{bool, int, double, string\}}{\Gamma \vdash_e e_1 \circledcirc e_2 : bool}\ \text{EComp}$$

where $\circledcirc \in \{<, >, <=, >=, ==, !=\}$

$$\frac{\Gamma \vdash_e e_1 : bool \quad \Gamma \vdash_e e_2 : bool}{\Gamma \vdash_e e_1 \&\& e_2 : bool}\ \text{EAnd} \qquad \frac{\Gamma \vdash_e e_1 : bool \quad \Gamma \vdash_e e_2 : bool}{\Gamma \vdash_e e_1 || e_2 : bool}\ \text{EOr}$$

$$\frac{id : T \in \Gamma \quad \Gamma \vdash_e e : T}{\Gamma \vdash_e id = e : T}\ \text{EAss}$$

Some reiterations on notation:

- $\vdash_p$ is the typing relation for programs. If a pair (Γ, p) is in the relation then we write $\Gamma \vdash_p p$ and say that the program p is well-typed in the environment Γ.
- $\vdash_d$ is the typing relation for definitions/declarations. If a triple (Γ, d, Γ') is in the relation then we write $\Gamma \vdash_d d \Rightarrow \Gamma'$ and say that the definition d evaluates to the environment Γ' in the initial environment Γ.
- $\vdash_s$ is the typing relation for statements. If a triple (Γ, stm, Γ') is in the relation then we write $\Gamma \vdash_s stm \Rightarrow \Gamma'$ and say that the statement stm evaluates to the environment Γ' in the initial environment Γ.
- $\vdash_e$ is the typing relation for expressions. If a triple (Γ, e, T) is in the relation then we write $\Gamma \vdash_e e : T$ and say that the expressions e has type T in environment Γ.

- $\ominus$ is used to represent empty lists of syntactic constructs, such as the empty list of definitions or the empty list of statements.
- $\varnothing$ is the empty context/environment.
- The symbol $\triangleright$ is the *environment extension operator*. The environment is a stack of contexts. If we had a stack of two contexts with Γ_1 on the bottom and Γ_2 on the top, we would write $\Gamma_1 \triangleright \Gamma_2$.
- We write $\Gamma, id : T$ to add the judgement $id : T$ (identifier id has type T) to the top context of the environment Γ.
- The symbol T represents any primitive type in the language $(void, bool, int, double, string)$, whereas we write $T_1 \to \cdots \to T_n$ to represent the type of a function whose arguments have types $T_1, \ldots, T_{n-1}$ and whose return type is T_n.
- The symbol Γ is often used to represent both environments and contexts. In general, if it is used with the $\triangleright$ operator, it represents a context. If it used free-standing, it represents an arbitrary environment.

Definitions of meta-expressions:

- The meta-expression $head(\Gamma)$ selects the top context from the environment Γ.
- The meta-expression $dom(\Gamma)$ is the set of identifiers that have type information in the context Γ.
- The meta-expression $returnType(\Gamma)$ is the expected return type of the current scope.
- The meta-expression $returnsCheck(stms, T)$ returns *true* if T is *void* or if there is a return statement anywhere along the main branch of execution in a list of statements representing a function body, and $false$ otherwise. It occurs in the PDEF rule to require that all non-*void* functions (besides "main"!) have a return statement to pass type checking. The definition of "main branch of execution" is the set of statements in a function whose execution does not depend on a condition–so statements not occurring within an if-else statement or a loop.
- The meta-variables id, d, stm represent an arbitrary identifier, definition/declaration, and statement, respectively. Anytime the meta-expression id is used as a subexpression (in the unary operators), your type checker should enforce that the expression is actually an identifier.

We will talk at length about this type system in class, but here is a very rough outline of how a type checker implementing this system may work:

- Do a first pass over every function definition in the program and insert the function identifiers into the environment with their corresponding types. This is what is shown on the left side premise of the PINIT rule, and carried out by the DFUN rule (among others).
- With the resulting environment from above, check that every individual function definition is valid. Note that an empty program (with no function definitions) is technically valid. This is what is shown on the right side premise of the PINIT rule, and carried out by the PFUN rule (among others).

It is recommended to implement this with a top-down approach. Start by checking whole programs and enforcing, for example, that the same function identifier is not declared twice. Once that's done, try checking parameter lists and, for example, making sure no two parameters to a single function are named the same. The supplied tests are also roughly in order of increasing difficulty, so you may find the development experience easier if you aim to pass the first few tests, then the next, etc.

2.3 Type Checker and Compiler Specification

Your type checker should run immediately after parsing in the compilation pipeline. The compiler should accept input on standard input (it already does in the starter code). Your type checker should exit with a non-zero exit code and output a message describing the type error to standard output if type checking fails, otherwise exit with a zero code and output a message indicating success.

2.4 Grading

Your submission will be tested with the following command from the directory of whichever implementation language you choose:

```
make test
```

The tentative score output by the testing suite is *tentative* in the sense that points may be deducted or added at the grader's discretion. For example, the test suite does not check for output messages but if your type checker is outputting nonsensical messages, points will be deducted. Your submission will earn roughly 2 points per passed test case (**100 points** total).

3 Extra Credit (Required for 5010)

For **15 points** extra credit (or for 15 points normal credit for 5010 students): Implement `for` loops, as described in question 2 of section 1, and implicit type conversions in your type checker.

3.1 For Loops

Implementing `for` loops will require you to add another rule to the `CPP.cf` grammar file to include the `for` loop syntax. Then you need to implement the typing rules in your type checker. If you have answered question 2 in section 1, it should be a pretty straightforward translation. However, unlike that question, we require that you restrict the type of statements that can occur in the *initialization* part of a `for` loop to be either an expression statement, declaration, or initialization. This is not difficult to enforce, but makes things a lot simpler from a verification perspective and brings our language closer to the C++ standard!. However, differing from the C++ standard, we do allow redeclaration of a variable that is declared in the *initialization* part in a block statement loop body.

3.2 Implicit Type Conversions

Read about implicit type conversions here. In our type system, type conversions should only be done *up* the type hierarchy specified by the following ordering on primitive types:

$$bool < int < double < string$$

Note the exclusion of the *void* type. There are no values in the *void* type and therefore no void value should ever be coerced to a value of another type.

Implement for loops by adding an appropriate rule to the `CPP.cf` grammar file and then handling them correctly in your type checker.

Implement implicit type conversions by adding an *internal* rule to the `CPP.cf` grammar file that adds an expression containing the expression being converted and the type being converted to. Because the rule is internal, syntax is not important. The primary way this feature changes your type checker is in the way it type checks expressions. An expression of a given type now always "fits" into places where an expression of a "greater" type is expected. For example, if $returnType(\Gamma) \equiv double$, then $\Gamma \vdash_s return\ true; \Rightarrow \Gamma$ is derivable. Note that the type checker should not coerce the type of variables being assigned to in assignment expressions; it should always try to convert the right side of an assignment to the type of the variable on the left side.

3.3 Testing

Test cases for these features are included in the `test/grad` directory in each of the implementation language directories. Take a look at those tests to get a better feel of what these features enable. Test your implementation by running the command:

```
make test_grad
```

in the directory of your implementation language. It will run the undergraduate and graduate test cases and output your tentative score.

4 More Extra Credit

For **10 points** more extra credit (graduate and undergraduate students), make your type checker an *annotating type checker*. As previously specified, our type checker can be considered an idempotent static verification algorithm in the sense that it doesn't output anything; it could be run multiple times on the abstract syntax tree and always have the same result. Moreover, If type checking passes, the next phases of our compiler will be forced to simply "take our word for it" that the program is indeed well-typed. This is fine in practice, but sometimes we want concrete *proof* that the program was actually well-typed. One way to get this proof is via an annotated abstract syntax tree, where each expression in the program is annotated with its inferred type. Change your type checker so that it returns such an annotated syntax tree when type checking succeeds.

This essentially changes every typing relation defined in section 2.2 to include another element which is the annotated syntax tree resulting from type checking an element in an environment. In practice, though, there are several ways you could do this. The easiest way, albeit perhaps the most ad-hoc, is to add an *internal* rule to the `CPP.cf` grammar file that adds an expression containing an expression and the type it has been annotated with. Then, your type checker should annotate every expression it encounters with its inferred type, and accumulate those annotated expressions in the resulting abstract syntax tree. Another way that saves you from adding another rule to the grammar file is to create another abstract syntax tree data type that uses an explicitly annotated expression data type. Another way that saves you from creating a whole other massive data type would be to add an internal annotation rule to the BNFC grammar that annotates expressions with an index into a list of explicitly annotated expressions. The possibilities are endless!

The ease with which you can implement this will largely depend on how you have implemented your type checker up to this point. This is especially true for graduate students, as the interaction between type conversion and type annotation can be tricky. The general algorithm for type inference combined with type conversion and type annotation is as follows (very roughly and omitting environments etc.):

```
inferExp(exp):
    annotatedSubExps := inferExp(subExps(exp))
    convertedType    := convertSubExps(annotatedSubExps)
    annotatedExp     := annotate(exp,annotatedSubExps,convertedType)
    return annotatedExp
```

Type checking for statements and definitions/declarations should also return the statements and definitions/declarations containing the annotated expressions, so that the entire resulting abstract syntax tree is fully annotated. For example, Figure 1 shows a source program that uses implicit type conversion followed by what my annotating type checker outputs for that program. The structure of annotated output implies things happen in the following order:

- The subexpression of the assignment (`true`) gets an inferred type of `bool` and is annotated accordingly.
- The type checker notices it is checking an assignment of a `bool` value to a variable of `int` type, and correctly adds an implicit conversion of the annotated `(true : bool)` expression to the `int` type.
- The entire expression is then annotated with a type of `int`.

```
int main()
{
    int x = true;
    return x;
}
```

```
int main()
{
    int x = (int ((true : bool)) : int);
    return (x : int);
}
```

Figure 1: Comparison of source program and annotated output

If you implement this feature, indicate so in your submission (in the README or on WyoCourses) and make your type checker output the full annotated syntax tree when type checking succeeds, using BNFC's generated `printTree` function or some custom `toString` function depending on your implementation. Since we can't easily automate the testing of this feature, it will be graded by examining the output of your type checker on the "success" test cases.