1 Introduction

The second project is to implement a type checker for LangF, which checks whether or not a parse tree is statically correct and produces a typed abstract syntax tree (AST) representation of the program. The abstract syntax tree includes information about the binding sites of identifiers and about the types of variables and expressions.

The project seed code will provide an ML-ULex based scanner, an ML-Antlr based parser (but you may also use your scanner and parser from Project 1), and modules for implementing the abstract-syntax-tree representation.

A companion document (The LangF Type System) describes the formal LangF type system; the goal of Project 2 is to implement a checker for that formal system. In this document, we cross-reference sections of the “The LangF Type System” using the notation TS[$n$], where $n$ is the section number.

2 Sample code

We will seed your repositories with sample code in a directory called proj2. The sample code includes a solution to Project 1, as well as a substantial amount of infrastructure to support type checking. This code is incomplete, but does compile. You should read this code carefully (especially the signatures, which are well documented), so as to avoid reimplementing mechanisms that are already provided. There are two directories that are of particular importance:

- The ast directory contains the definitions for Abstract Syntax Tree (AST) representation of LangF programs. You will not have to make any modifications to the code in this directory, but you will need to understand the data structures and operations that are provided.
- The type-checker directory contains the modules that implement the type checker. Some of the type checker code is provided, but other parts are missing. Look for statements of the form “raise Fail "YOUR CODE HERE".” You should make sure to understand the working of the Context module (defined in typechecker/context.sml), since it provides

---

1Remember, a specification is a description of a property (yes/no question; true/false statement). It does not define (though it may suggest) an implementation for deciding whether or not the property holds. A significant component of this project is to develop the skills needed to produce an implementation from a specification.
the main mechanism for managing the typing context (i.e., the environments, source-location, and error streams).

3 Checking Patterns

One of the syntactic restrictions discussed in TS[§3], is that the patterns of a case expression must be exhaustive and useful. Because of the syntactic restrictions that LangF places on patterns, this check is fairly straightforward.

Consider the following generic case expression

```haskell
case exp of
  \{ p_1 \Rightarrow scope_1 \}
  \ldots
  \{ p_n \Rightarrow scope_n \}
end
```

and assume that it is type correct (we do not need to do the redundancy check when there is a type error in the argument). If the type of exp is not a datatype, then any rule past the first is useless (or redundant), because of the fact that the pattern $p_1$ can only be a variable, wildcard, or tuple of variables/wildcards, and, thus $p_1$ matches any value of the correct type.

The more common case is when the type of exp is a datatype. In this case, the patterns $p_1, \ldots, p_n$ must be either variables, wildcards, or constructor patterns. If some pattern $p_i$ ($i < n$) is a variable or wildcard, then the patterns $p_{i+1}, \ldots, p_n$ are useless. If all of the patterns are constructor patterns, the constructors must be unique (otherwise the second occurrence of a constructor is useless) and must cover all of the constructors of the datatype (otherwise the pattern is not exhaustive).

We have provided a utility module, called Coverage, that implements the above logic. You should use it when processing the rules of a case expression.

4 Building the Typed AST

In addition to checking the program for type correctness, your program is also responsible for converting the parse tree representation of the program into the typed abstract-syntax tree (AST) representation. The AST is a further simplification of the program representation.

• Unlike the parse tree, the AST does not include source location information.

• AST type variables, type constructors, data constructors, and variables in the AST representation are implemented the types TyVar.t, TyCon.t, DataCon.t, and Var.t (respectively). As discussed in TS[§4], semantic type variables, type constructors, data constructors, and variables are used to distinguish different binding occurrences, which otherwise have the same syntactic names. Each of the provided modules includes a new function for creating unique identifiers.

• Variables bound in simple patterns include their type.

• Wild cards are replaced with unique variables.
• There is no type declaration form; all type abbreviations will have been expanded during type checking and conversion to the abstract syntax tree.

• The let declaration form has no type constraint, since the representation of AST variables includes the variable’s type.

• The AST representation of expressions is split into two levels, the type exp_rep datatype that has a constructor per syntactic form and the exp datatype that pairs a syntactic form with its type.

• Polymorphic data constructors in patterns are applied to their type arguments.

We have already introduced one form in the abstract syntax tree representation: the semantic types from TS[§4]. Similarly, we have already introduced one judgement for translating a parse tree representation form into an abstract syntax tree representation form: the judgement for type checking types from TS[§6.3]:

\[ E \vdash \text{typ} \triangleright \tau \quad \text{in the environment} \ E, \text{the parse tree type} \ \text{typ} \text{ is well-formed and translates to the AST type} \ \tau. \]

We can imagine other judgements that combine type checking with translation to the AST:

\[ E \vdash \text{exp} \triangleright \tau; \ e \quad \text{in the environment} \ E, \text{the parse tree expression} \ \text{Exp} \text{ has the abstract syntax tree type} \ \tau \text{ and translates to the abstract syntax tree expression} \ e. \]

\[ E \vdash \text{def} \triangleright \ E'; \ d \quad \text{in the environment} \ E, \text{the parse tree declaration} \ \text{Decl} \text{ returns the environment} \ E' \text{ and translates to the abstract syntax tree declaration} \ d. \]

\[ E \vdash \text{prog} \triangleright p \quad \text{the parse tree program} \ \text{Prog} \text{ is statically correct and translates to the abstract syntax tree program} \ p. \]

The inference rules for these judgements will be very similar to those given in TS[§6], except they will construct an appropriate output abstract syntax tree form.

5 Hints

• This project is probably the most difficult and substantial of the projects, so budget your time appropriately and start early.

• Study the sample code carefully. You will need to be familiar with the types and operations in the ParseTree, Type, and AST modules.

• The provided Context module provides a type that groups the environments along with location information and the error stream.

• We have provided a template for the structure of the type checker split across four modules. Each typing judgement can be implemented as a function. Each function for a judgement will have a case for each typing rule with that judgement as the conclusion.
• Work in stages. First implement a simple binding checker that only checks that the program has no unbound variables (but does not check types and does not produce an abstract syntax tree). You can use a “bogus” return value to get the types right at this stage without having to write the code that constructs the AST. This binding checker will establish the basic structure of the implementation. Next, implement a type checker that checks for unbound variables and checks types (but does not produce an abstract syntax tree). This type checker will require extending the simple binding checker, but will very closely match the typing rules from TS[§6]. Next, implement a full type checker that checks for unbound variables, checks types, and produces an abstract syntax tree. This full type checker will require additional information to be carried in the environment and to be returned by each type checking function. Finally, extend the full type checker to additionally check the syntactic restrictions of TS[§3].

• Work on error reporting last. Detecting errors and producing good error messages can be difficult; it is more important for your type checker to work on good programs than for it to “work” on bad programs. Again, work in stages. First implement a type checker that stops after detecting the first error. Then implement a type checker that continues after detecting an error.

• The representation of semantic types includes a special constructor ErrorTy that can be used when an error in the program prevents computing the type of an expression, etc. The ErrorTy will test equal to any other type (using the TypeUtil.same function), which can help avoid cascading errors.

6 Submission

We will collect the projects at 23:59 (Chicago time) on November 5, 2020 from the SVN repositories, so make sure that you have committed your final version before then.

Important note: You are expected to submit code that compiles and that is well documented. Remember that points for project code are assigned 30% for coding style (documentation, choice of variable names, and program structure), and 70% for correctness. Code that does not compile will not receive any points for correctness.

7 Document history

October 30, 2020 Extended due date to November 5, 2020.

October 22, 2020 Fix grammatical issue.

October 21, 2020 Original version.

---

2 Once you add error checking, you may want to return bogus expressions and patterns when you encounter an error.