Stage 3: Types and Type-checking

Due May 3, 2021 by 11:59pm **Points** 100 **Available** until Jun 5, 2021 at 11:59pm

This assignment was locked Jun 5, 2021 at 11:59pm.

Introduction

This project implements additional semantic checks on the AST generated in the previous lab. Now that we can construct a tree representation of input Crux source code, we are able to perform additional semantic checks. All popular computer languages implement some kind of type system. You are probably quite used to satisfying the Java type system by now. Indeed, hopefully, you are using it (alongside testing) to enforce design constraints on your code, and alert you to mistakes. Depending on the language, the type system can do type checks at runtime (Scheme, JavaScript, Python, Ruby), or compile time (Haskell, OCaml), or a mix of both. Crux sits squarely in the compile time camp, so a Crux compiler has to check that all Crux programs obey the rules of the Crux type system, before it emits machine code.

In the past we've used the ANTLR to recognize illegal syntax, and the symbol table to recognize illegal usage of variables and functions. Now we add a type system so that we can detect semantic errors such as the illegal use of expressions, invalid arguments to functions, returns incompatible with their function return type, etc. In many languages the type system embodies a powerful checker, which programmers use to enforce design constraints and catch errors.

Designing a Type-checking System

With a type system, we aim to prevent the programmer from accidentally using operations in an invalid manner. For example, it's nonsensical to add the number 5 with the boolean true. In industrial programming languages, the type system can be quite complicated, allowing for automatic conversions between types (called type coercion), operator and function overloading, multiple or single inheritance, covariant returns, etc. Crux's type system is very simple in comparison. It allows operations (`add`, `sub`, `mul`, `div`, `and`, `or`) only between "like" types. That is, ints operate with ints and booleans only with booleans, and return values must match the function return type exactly.

In spite of the simplicity of our system, we still practice good coding techniques developed for more complex systems. The research article Design Patterns for Teaching Type Checking in a Compiler Construction Course demonstrates how to use the Composite Pattern to represent a set of base types together with operations which can be performed on those types. The types package in lab implements this design pattern. The abstract base class `Type` defines a default implementation for each of the operations in the Crux language.

The Base Types

Crux has the types int and bool represented by `IntType` and `BoolType`. It also has the `VoidType`, which is present to distinguish functions that return a value from those which do not.

The Composite Types

On top of the base types, Crux allows the creation of arrays, represented by `ArrayType`, and addresses, represented by `AddressType`. The constructor for `ArrayType` takes another type argument to distinguish what type the array contains. Using this argument, we can make an array of one of the base types, or an array of arrays. The constructor for `AddressType` takes another type argument to distinguish what type of value the address references. Using this argument, we can make addresses over any other type, including arrays. More powerful languages use the composite technique to build up structures and classes.

The Function Type

A computer language is not very useful without the function/procedure abstraction. The Crux compiler has a `FuncType` which tracks both the arguments `TypeList` and return type of a function. Within the type system, the function type could be used to construct functions that return/accept arrays, or even functions that return/accept other functions. However, the Crux grammar prevents us from expressing these more complex programs, restricting us to only the base types.

The Error Type

Errors which might occur during type checking are represented with the `ErrorType`, which contains a field, `String message`, to convey the reason for the error.

The Operations

For each operation available in the Crux language, we implement a checking function in the Type base class. Most of these operations, such as the basic arithmetic operations, are immediately recognizable.

Method/Operation	Description
`Type add(Type that)`	Addition of two expressions.
`Type sub(Type that)`	Subtraction of two expressions.
`Type mul(Type that)`	Multiplication of two expressions.
`Type div(Type that)`	Division of two expressions.
`Type and(Type that)`	Logical and of two expressions.
`Type or(Type that)`	Logical or of two expressions.
`Type not()`	Negation of one expression.
`Type compare(Type that)`	Logical comparison of two expressions.
`Type deref()`	Obtain the value at a given address.
`Type index(Type that)`	Index into an array.
`Type call(Type args)`	A function call.
`Type assign(Type source)`	Assign an expression to a designator.

Equivalence

We also implement an additional method boolean `equivalent(Type that)`, which allows the type system to detect if two type objects are structurally equivalent. This operation comes in especially handy when checking that the TypeList of a function call matches the function's signature.

Laying Some Groundwork

In order to make use of the type system, we shall have to make some modifications to our existing codebase. One of these changes includes the addition of a type field to the Symbol class. When a function, variable, or array is declared we shall store the type information with the newly created symbol. The type field allows the type-checker to access the information when it sees an access of that symbol in the AST. Additionally, the `ParseTreeLower` needs some modification in order to attach or update the type of newly created symbols.

The Type Checker

We implement the `TypeChecker` in terms of a `NodeVisitor`. Its job is to walk the AST produced by the `ParseTreeLower`, and check for any of the type errors that may occur in the Crux language. It accomplishes this task by first descending down to the leafs of the tree and then propagating type information up to the root as it unwinds. The checker must therefore associate some type information with each node of the AST.

Java does not allow us to extend the AST nodes for the purpose of adding type information, nor are we able to change the argument type of the visitor methods. As a result of this restriction, we create a `HashMap<Node, Type> typeMap` which records the association for us. Because the association is external to the nodes in the AST, the TypeChecker manually manages the association.

Reporting Errors

The `TypeChecker` also contains a `ArrayList<String> errors` field for recording any type errors it encounters during its traversal of the AST. Each time the `TypeChecker` enters an association into the `typeMap` it checks for the `ErrorType` and records a message into `errors` when one is present. It is convenient to create a wrapper for the `typeMap.put()` method, which catches any type error messages.

Some of the nodes in the AST require additional checks. For example, both the `IfElseBranch` and `WhileLoop`, much check that their conditions are a `BoolType`. Return statements must verify that they are compatible with the function being defined. At least one check, that functions declaring a non-`void` return value actually return in all possible paths, requires significant thought. I strongly suggest writing helper and convenience methods.

What do I need to implement?

- Modify your `Symbol` classes to contain a field for storing the type.
- Modify your `ParseTreeLower` so that it attaches the type to the symbols when they are created.
- Supply the appropriate implementation for the operation methods of classes in the types package.
- Complete the implementation of the `TypeChecker` visitor, so that it passes only those programs which satisfy all of the Type Semantics section of the Crux specification.
- Add a type specification to each of the predefined functions.
- Enforce that the main function has the appropriate signature.

Submission

Please follow the submission guides in the Project Overview. Make sure you pass all the public test cases.

Submission

Submission Details

Grade: 100% (100 pts possible)
Graded Anonymously: no

Comments:
No Comments