Research Report: sslang

Leo Qiao (flq2101)

December 26, 2021

1 Introduction

This research report gives an overview of my research progress during 2021 Fall as a member of the Type System team within the sslang research group. The team is headed by John Hui and consists of Xijiao Li and Leo Qiao (me). For this semester, the team mainly focused on planning and implementing the following 2 features for sslang:

- type inference
- typeclass

While Xijiao and I studied both topics, Xijiao primarily worked on type inference and I primarily worked on typeclass. As a result, this report will discuss our progress on typeclass implementation. All code changes to implement typeclass are currently recorded in the "typeclass-instantiation" branch (PR: "Typeclass instantiation") [1]

2 Motivation

The current version of sslang does not support ad-hoc polymorphism. In other words, there cannot be a function that can exhibit different behaviors (or, use different function body) when different argument types are applied. Ad-hoc polymorphism is a feature that could add expressiveness to sslang programs.

3 Other implementations of ad-hoc polymorphism

3.1 Haskell: typeclass

A popular language that has ad-hoc polymorphism is Haskell. Haskell supports ad-hoc polymorphism through a type system construct called typeclasses. The following Haskell code demonstrates the usage of typeclasses:

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

elem :: Eq a => a -> [a] -> Bool
elem y []     = False
elem y (x:xs) = (x == y) || elem y xs
```

The class block defines the Eq typeclass, which acts like a protocol that its instances need to conform to. The function signatures inside the class block denote the types of the methods that the instances should have. In the type signature of elem, there is a typeclass constraint that demands type variable a to be an instance of the Eq typeclass.

To make a type become an instance of the Eq typeclass, one write the following code:

[1] PR: https://github.com/ssm-lang/sslang/pull/49

```
data Foo = Bar | Baz

instance Eq Foo where
  Bar == Bar = True
  Baz == Baz = True
  Bar == Baz = False
  Baz == Bar = False

  x \= y = not (x == y)
```

3.2 Swift: protocol

Another popular language with ad-hoc polymorphism is Swift. It does so with its protocol construct, which has a lot of similarities as Haskell typeclass.

A protocol that simulates a Monad looks like the following:

```
protocol Monad {
    typealias F
    typealias U
    class func bind<M : Monad where M.U == U>(Self, F -> M) -> M
    class func `return`(F) -> Self
}
```

Notice that it has not only associated methods, but also associated types.

One common way to make a type conform to a protocol is through the extension block. The following code makes Array conform to the Monad typeclass:

```
extension Array : Monad {
    typealias F = T
    typealias U = Array<()>
    static func bind<M : Monad where M.U == U>(m : Array, _ f : F -> M) -> M {
        var ret = Array<M.F>()
        for a in m {
            ret += f(a) as M.F[]
        }
        return ret as M
    }
    static func `return`(f : F) -> Array {
        return [f]
    }
}
```

4 sslang typeclass

The team has decided on implementing ad-hoc polymorphism with a Haskell-like typeclass construct. For the first milestone of adding typeclasses, the sslang typeclass should have the following specifications:

- single parameter
- no parent typeclass
- support associated methods
- no default method implementations

4.1 Example usage

The following code demonstrates the usage of the sslang typeclass. Note that the syntax is still subject to change.

```
class Collapsible a
  collapse: a -> Bool

instance Collapsible Bool
  collapse(x) =
    x

instance Collapsible Int
  collapse(x) =
    match x
      | 0 = False
      | otherwise = True

collapseToggle[Collapsible a](x: a, led: &Bool) -> () =
  led <- collapse x

useCollapseToggle(led: &Bool) -> () =
  led <- collapseToggle (1, led)
```

4.2 Syntax

4.2.1 class

The syntax for class definition is similar to Haskell. However, the where keyword is omitted and the class block uses either explicit curly-bracket or indentation-based layout. The parser CFG is given below:

```
-- | Single class definition.
defClass                              --> ClassDef
  : 'class' id id '{' methodDecls '}'    { ClassDef $2 $3 $5 }

-- | Class method declatation
methodDecls                           --> [(VarId, TypeAnn)]
  : id ':' typ ';' methodDecls          { ($1, $3) : $5 }
  | id ':' typ                          { [($1, $3)] }
```

4.2.2 instance

The syntax for instance definition is also similar to Haskell but without the where keyword. The parser CFG is given below:

```
-- | Single instance definition
defInst                               --> InstDef
  : 'instance' id id '{' methodDefs '}' { InstDef $2 $3 $5 }

-- | Instance method definitions
methodDefs                            --> [Definition]
  : defLet ';' methodDefs               { $1 : $3 }
  | defLet                              { [$1] }
```

4.2.3 typeclass constraint

The syntax for typeclass constraint in a function type signature is currently still under discussion. We aim to consolidate this syntax in the beginning of next semester.

5 Important sslang code changes

To incorporate the typeclass construct, there are some major changes to the rest of the sslang codebase.

5.1 Ast.Program

The user-defined classes and instances exist separately from the rest of the program. As a result, the Ast.Program definition has been changed to the following:

```
newtype Program = Program [Top]
  deriving (Eq, Show)

data Top = TopDef Definition
         | TopClass ClassDef
         | TopInst InstDef
         deriving (Eq, Show)

data ClassDef = ClassDef
  { className    :: TClassId
  , classTVar    :: TVarId
  , classMethods :: [(VarId, TypAnn)]
  }
  deriving (Eq, Show)

data InstDef = InstDef
  { instClassName :: TClassId
  , instTCon      :: TConId
  , instMethods   :: [Definition]
  }
  deriving (Eq, Show)
```

5.2 IR.IR.Program

The user-defined classes and instances need to be stored within the IR. As a result, the IR.IR.Program definition has been changed to the following:

```
data Program t = Program
  { programEntry    :: VarId
  , programDefs     :: [(VarId, Expr t)]
  , programTypes    :: [(TConId, TypeDef t)]
  , programClasses  :: [ClassDef t]
  , programInsts    :: [InstDef t]
  }
  deriving (Eq, Show)

data ClassDef t = ClassDef
  { className    :: ClassId
  , classTVar    :: t
  , classMethods :: [(VarId, t)]
  }
  deriving (Eq, Show)

data InstDef t = InstDef
  { instConstraint :: InstConstraint t
  , instMethods    :: [(VarId, Expr t)]
  }
  deriving (Eq, Show)
```

6 Implementation status

To implement typeclass in sslang, we are using the dictionary-passing technique [2]. Currently, on the typeclass−instantiation branch, the following has been implemented and tested:

- instantiation of classes
- instantiation of instances

All tests are passing. The implementation details can be found in ClassInstantiation.hs.

6.1 class instantiation

The example is taken from a test case in ClassInstantiationSpec.hs. Note that the test cases are manually written IR.

Written in sslang, the pre-instantiation program is the following:

```
class Collapsible a
  collapse: a -> Int
```

The post-instantiation program is:

```
type Collapsible a = Collapsible { collapse: a -> Int }
```

6.2 instance instantiation

The example is taken from another test case in ClassInstantiationSpec.hs.

The pre-instantiation program is:

```
class Collapsible a
  collapse: a -> Int

instance Collapsible Int
  collapse(x: Int) =
    match x
      0 -> 0
      _ -> 1
```

The post-instantiation program is:

```
type Collapsible a = Collapsible { collapse: a -> Int }

Collapsible_Int = Collapsible (fun x {
  match x
    0 -> 0
    _ -> 1
})
```

7 Next steps

In the next semester, the main goal is add single-parameter typeclass to the sslang "main" branch. In order to do so, the following problem will need to be solved.

- type inference pass that is aware of typeclasses: this is necessary in order to let typeclasses flow through the entire compiler pipeline. We currently have it in the lexer, parser and lower-ast passes.
- syntax for typeclass constraint: this is necessary to start implementing typeclass-constrained functions

[2] https://okmij.org/ftp/Computation/typeclass.html

- integration with the rest of sslang: the typeclass construct needed a lot of changes in the current codebase. There are also a lot of moving pieces (many PRs) in sslang in general, so some synchronization in the codebase is crucial.
- ADTs and pattern-matching: these are used in the dictionary-passing technique, so we need them for typeclasses to work in sslang.