Representation of Simply-Typed λ-Calculus

Khizar Siddiqui

November 29, 2020

Contents

1 Introduction

This documentation was created for Assignment 2 of 3MI3, it's purpose is to help understand the fundamentals behind how (and why) my program runs. Lambda calculus is a formal system for expressing computation based on function abstraction and application. We will attempt to create a typed interpretation of the lambda calculus (also known as Simply-Typed λ-Calculus) using Scala and Ruby.

2 Representation

Our representation adds on to the standard untyped λ calculus expressions **(variable, abstraction, application)** with some additional terms **(zero, successor, isZero, true, false, test)** as well as a way to represent these terms with a type.

2.1 Scala

For Scala, we will classify our main term STTerm and type STType as a trait and our typed terms and custom types as classes that extend this trait. Our variable term will only take an integer input since a variable will be represented by an integer, and True/False/Zero are all objects that need no parameters but the rest of the terms will take inputs that match the type term as well as the type of the custom types that we create.

```scala
sealed trait STTerm
case class STVar(index: Int) extends STTerm {
    override def toString() = index.toString()
}
case class STApp(t1: STTerm, t2: STTerm) extends STTerm {
    override def toString() = "(" + t1.toString() + ") (" + t2.toString() + ")"
}
case class STAbs(t: STType, t1: STTerm) extends STTerm {
    override def toString() = "lambda " + t.toString + " . " + t1.toString()
}
case object STZero extends STTerm {
    override def toString() = "0"
}
case class STSuc(t: STTerm) extends STTerm {
    override def toString() = "S " + t.toString()
}
case class STIsZero(t: STTerm) extends STTerm {
    override def toString() = t.toString() + " ?"
}
case object STTrue extends STTerm {
    override def toString() = "True"
}
case object STFalse extends STTerm {
    override def toString() = "False"
}
case class STTest(t1: STTerm, t2: STTerm, t3: STTerm) extends STTerm {
    override def toString() = "test " + t1.toString() + " " + t2.toString() + " " + t3.toString()
}

sealed trait STType
case object STNat extends STType {
    override def toString() = "nat"
}
case object STBool extends STType {
    override def toString() = "bool"
}
case class STFun(dom: STType, codom: STType) extends STType {
    override def toString() = "(" + dom.toString + ") -> (" + codom.toString + ")"
}
```

2.2 Ruby

For Ruby, we define our main term STTerm and type STType as classes and our typed terms and custom types as classes that extend this main class. Our term parameters are initialized with "attr_reader" to allow accessing. Moreover, for individual terms and types, we define "==" functions for comparisons between two terms/types.

```
class STTerm
end

class STVar < STTerm
    attr_reader :index

    def initialize(i)
        unless i.is_a?(Integer)
          throw "Constructing a lambda term out of non-lambda terms"
        end
        @index = i
    end

    def ==(r); r.is_a?(ULVar) && r.index == @index end
end

class STApp < STTerm
    attr_reader :t1
    attr_reader :t2

    def initialize(t1, t2)
        unless t1.is_a?(STTerm) && t2.is_a?(STTerm)
          throw "Constructing a lambda term out of non-lambda terms"
        end
        @t1 = t1
        @t2 = t2
    end

    def ==(r); r.is_a?(STApp) && r.t1 == @t1 && r.t2 == @t2 end
end

class STAbs < STTerm
    attr_reader :t
    attr_reader :t1

    def initialize(t, t1)
        unless t.is_a?(STType) && t1.is_a?(STTerm)
            throw "Constructing a lambda term out of non-lambda terms"
        end
        @t = t
        @t1 = t1
    end

    def ==(r); r.is_a?(STAbs) && r.t == @t && r.t1 == @t1 end
end

class STZero < STTerm
    def ==(r); r.is_a?(STZero) end
```

```ruby
end

class STSuc < STTerm
    attr_reader :t

    def initialize(t)
        unless t.is_a?(STTerm)
            throw "Constructing a lambda term out of non-lambda terms"
        end
        @t = t
    end

    def ==(r); r.is_a?(STSuc) && r.t == @t end
end

class STIsZero < STTerm
    attr_reader :t

    def initialize(t)
        unless t.is_a?(STTerm)
            throw "Constructing a lambda term out of non-lambda terms"
        end
        @t = t
    end

    def ==(r); r.is_a?(STIsZero) && r.t == @t end
end

class STTrue < STTerm
    def ==(r); r.is_a?(STTrue) end
end

class STFalse < STTerm
    def ==(r); r.is_a?(STFalse) end
end

class STTest < STTerm
    attr_reader :t1
    attr_reader :t2
    attr_reader :t3

    def initialize(t1, t2, t3)
        unless t1.is_a?(STTerm) && t2.is_a?(STTerm) && t3.is_a?(STTerm)
            throw "Constructing a lambda term out of non-lambda terms"
        end
        @t1 = t1
        @t2 = t2
        @t3 = t3
    end

    def ==(r); r.is_a?(STTest) && r.t1 == @t1 && r.t2 == @t2 && r.t3 == @t3 end
end

class STType end
```

```ruby
class STNat < STType
    def ==(type); type.is_a?(STNat) end
    def to_s; "nat" end
end

class STBool < STType
    def ==(type); type.is_a?(STBool) end
    def to_s; "bool" end
end

class STFun < STType
    attr_reader :dom
    attr_reader :codom

    def initialize(dom, codom)
        unless dom.is_a?(STType) && dom.is_a?(STType)
            throw "Constructing a type out of non-types"
        end
        @dom = dom; @codom = codom
    end

    def ==(type); type.is_a?(STFun) && type.dom == @dom && type.codom == @codom end
    def to_s; "(" + dom.to_s + ") -> (" + codom.to_s + ")" end
end
```

3 Typechecking

Now that we have defined our terms, we can create a function typecheck that checks if our term obeys the type rules. This is done by checking if the parameters match the typerules and the parameters of those and so on (using recursion). If at any point, typerules do not match, we note this and come back to our main function to return false for the term being properly typed (both languages use different ways to check this). While typechecking, we keep track of variable types through the use of an environment (I have gone with a List implementation).

3.1 Scala

For Scala, we define a new function typecheck and a function typeOf within it. typecheck calls the typeOf with the empty environment and typefo pattern matches the parameter with different STTerms and recurses until either failure of typerules or reaching the end. I also use the implementation of Either[] in this function to keep track of failed typerules (represented by None). If we encounter None at some point, the function simply returns False otherwise True

```
def typecheck(term: STTerm): Boolean = {
    def typeOf(env: List[STType], exp: STTerm): Option[STType] = exp match {
        case STVar(index) => (env.length > index) match {
            case true => Some(env(index))
            case false => None
        }
        case STAbs(t, t1) => {
            val newenv = List(t) ++ env
            val newtype = typeOf(newenv, t1)
            newtype match {
                case Some(value) => Some(STFun(t, value))
                case None => None
            }
        }
        case STApp(t1, t2) => typeOf(env, t1) match {
            case Some(value) => value match {
                case STBool => None
                case STNat => None
                case STFun(dom, codom) => typeOf(env, t2) match {
                    case Some(value) if (value == dom) => Some(codom)
                    case Some(value) => None
                    case None => None
                }
            }
            case None => None
        }
        case STZero => Some(STNat)
        case STSuc(t) => typeOf(env, t) match {
            case Some(value) if (value == STNat) => Some(STNat)
            case Some(value) => None
            case None => None
        }
        case STIsZero(t) => typeOf(env, t) match {
            case Some(value) if (value == STNat) => Some(STBool)
            case Some(value) => None
            case None => None
        }
        case STTrue => Some(STBool)
```

```
        case STFalse => Some(STBool)
        case STTest(t1, t2, t3) => typeOf(env, t1) match {
            case Some(value1) if (value1 == STBool) => typeOf(env, t2) match {
                case Some(value2) => typeOf(env, t3) match {
                    case Some(value3) if (value2 == value3) => Some(value2)
                    case Some(value) => None
                    case None => None
                }
                case Some(value) => None
                case None => None
            }
            case Some(value) => None
            case None => None
        }
    }

    typeOf(List[STType](), term) match {
        case Some(value) => true
        case None => false
    }
}
```

3.2 Ruby

In Ruby, instead of the Either[] implementation, we use Nil to represent our failed typerules. Each term contains the typeOf function that has been passed the environment. The main typecheck function resides in the STTerm class since every term has the same code for the initial ”passing an empty environment” part and we can be less redundant this way:

```
class STTerm
    def typecheck
        (typeOf(Array.new)) ? true : false
    end
end

class STVar < STTerm
    def typeOf(env)
        (env.length > @index) ? env[@index] : nil
    end
end

class STApp < STTerm
    def typeOf(env)
        val1 = @t1.typeOf(env)
        val2 = @t2.typeOf(env)
        if (val1.is_a?(STFun))
            if (val2 == val1.dom)
                val1.codom
            else
                nil
            end
        else
            nil
        end
    end
end

class STAbs < STTerm
    def typeOf(env)
        newenv = env.unshift(@t)
        (@t1.typeOf(newenv)) ? STFun.new(@t, @t1.typeOf(newenv)) : nil
    end
end

class STZero < STTerm
    def typeOf(env); STNat.new end
end

class STSuc < STTerm
    def typeOf(env)
        (@t.typeOf(env) == STNat.new) ? STNat.new : nil
    end
end

class STIsZero < STTerm
    def typeOf(env)
        (@t.typeOf(env) == STNat.new) ? STBool.new : nil
```

```ruby
    end
end

class STTrue < STTerm
    def typeOf(env); STBool.new end
end

class STFalse < STTerm
    def typeOf(env); STBool.new end
end

class STTest < STTerm
    def typeOf(env)
        type1 = @t1.typeOf(env)
        type2 = @t2.typeOf(env)
        type3 = @t3.typeOf(env)
        if (type1 == STBool.new) && (type2 == type3)
            type2
        else
            nil
        end
    end
end
```

For the sake of clear documentation, I have shown the classes with their respective typeOf functions within. In reality, you would simply add the function into the already created classes from earlier.

4 Translation to untyped λ calculus

We will now translate our Simply typed terms into Untyped terms. This is done by removing the type for Abstraction and using the λ definitions for our other terms (True/False/Zero etc.) as well as combining them with other terms occasionally using STApp.

4.1 Scala

In Scala, this is simply implemented by a single function that maps each STTerm with it's corresponding ULTerm or representation using ULTerms:

```
def eraseTypes(term: STTerm): ULTerm = term match {
    case STVar(index) => ULVar(index)
    case STAbs(t, t1) => ULAbs(eraseTypes(t1))
    case STApp(t1, t2) => ULApp(eraseTypes(t1), eraseTypes(t2))
    case STZero => ULAbs(ULAbs(ULVar(0)))
    case STSuc(t) => ULApp(ULAbs(ULAbs(ULAbs(ULApp(ULVar(1),ULApp(ULApp(ULVar(2),ULVar(1)),
                     ULVar(0)))))),eraseTypes(t))
    case STIsZero(t) => ULApp(ULAbs(ULApp(ULApp(ULVar(0),ULAbs(eraseTypes(STFalse))),
                        eraseTypes(STTrue))), eraseTypes(t))
    case STTrue => ULAbs(ULAbs(ULVar(1)))
    case STFalse => ULAbs(ULAbs(ULVar(0)))
    case STTest(t1, t2, t3) => ULApp(ULApp(ULApp(ULAbs(ULAbs(ULAbs(ULApp(ULApp(ULVar(2),
                               ULVar(1)), ULVar(0))))), eraseTypes(t1)), eraseTypes(t2)),
                               eraseTypes(t3))
}
```

4.2 Ruby

In Ruby, we can simply define our function within each term class that returns the respective ULTerm translation:

```
class STVar < STTerm
    def eraseTypes
        ULVar.new(@index)
    end
end

class STApp < STTerm
    def eraseTypes
        ULApp.new(@t1.eraseTypes, @t2.eraseTypes)
    end
end

class STAbs < STTerm
    def eraseTypes
        ULAbs.new(@t1.eraseTypes)
    end
end

class STZero < STTerm
    def eraseTypes
        ULAbs.new(ULAbs.new(ULVar.new(0)))
    end
end

class STSuc < STTerm
    def eraseTypes
        ULApp.new(ULAbs.new(ULAbs.new(ULAbs.new(ULApp.new(ULVar.new(1),ULApp.new(
        ULApp.new(ULVar.new(2),ULVar.new(1)),ULVar.new(0)))))), @t.eraseTypes)
    end
end

class STIsZero < STTerm
    def eraseTypes
        ULApp.new(ULAbs.new(ULApp.new(ULApp.new(ULVar.new(0), ULAbs.new(STFalse.eraseTypes)),
        STTrue.eraseTypes)), @t.eraseTypes)
    end
end

class STTrue < STTerm
    def eraseTypes
        ULAbs.new(ULAbs.new(ULVar.new(1)))
    end
end

class STFalse < STTerm
    def eraseTypes
        ULAbs.new(ULAbs.new(ULVar.new(0)))
    end
end
```

```ruby
class STTest < STTerm
    def eraseTypes
        ULApp.new(ULApp.new(ULApp.new(ULAbs.new(ULAbs.new(ULAbs.new(ULApp.new(ULApp.new(
        ULVar.new(2), ULVar.new(1)), ULVar.new(0)))))), @t1.eraseTypes), @t2.eraseTypes),
        @t3.eraseTypes)
    end
end
```

For the sake of clear documentation, I have shown the classes with their respective typeOf functions within.In reality, you would simply add the function into the already created classes from earlier.