Computer Science 3MI3 – 2020 Assignment 2: Typing a -calculus

Cynthia Liu

November 26, 2020

Contents

1 Introduction

This is the documentation for Assignment 2 for COMPSCI3MI3 2020fall.

It is about constructing representation of a simply-typed -calculus, and a typechecker for that -calculus, as well as a type-erasure and a simple translator to simplify terms to untyped -calculus terms.

This Assignment is written in both Scala and Ruby.

2 Part one: The representation

2.1 Representation in Scala

The terms of the -calculus ST are expressed in the new type `STTerm`.

- The constructor `STVar` take a parameter of type `Int`.
- The constructor `STZero`, `STTrue`, `STFalse` take no arguments, so they are implemented as case object
- The constructor `STSuc`, `STIsZero` take one parameters of type `STTerm`.
- The constructor `STApp` takes two parameters of type `STTerm`.
- The constructor `STAbs` takes two parameters, one is of type `STTerm`, the other is of type `STType`.
- The constructor `STTest` takes three parameters of type `STTerm`.

```
sealed trait STTerm
case class STVar(index: Int) extends STTerm
case class STApp(t1: STTerm,t2: STTerm) extends STTerm
case class STAbs(t: STType,term: STTerm) extends STTerm

case object STZero extends STTerm
case class STSuc(t: STTerm) extends STTerm
case class STIsZero(t: STTerm) extends STTerm

case object STTrue extends STTerm
case object STFalse extends STTerm
case class STTest(t1: STTerm,t2: STTerm,t3: STTerm) extends STTerm
```

2.2 Representation in Ruby

In Ruby, all constructors are subtypes of type `STTerm`, that is, they all inherites from `STTerm`

```
class STTerm end
```

The operator `<` means inheritance, for each class, we have an initialize method to initialize the object, and we have a == method to do comparsion.

- For `STVar`, it is initialized with an integer.

```
class STVar < STTerm
  attr_reader :index
  # We require our variables are only indexed by integers.
  def initialize(index)
    unless index.is_a?(Integer)
      throw "Constructing a STVar out of non-integer terms"
    end
    @index = index
  end
  def ==(type); type.is_a?(STVat) && type.index==@index end
end
```

- For `STZero`, `STTrue`, `STFalse`, they are initialized directly.

```
class STZero < STTerm
  def ==(type); type.is_a?(STZero) end
  def to_s; "zero" end
  def typeOf(arr); STNat.new end
end
```

- For `STSuc`, `STIsZero`, they are initialized with a `STTerm`.

```
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 ==(type); type.is_a?(STSuc) && type.t==@t end
end
```

- For `STApp`, it is initialized with two `STTerm`.

```
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 ==(type); type.is_a?(STApp) && type.t1==@t1 && type.t2==@t2 end
  end
```

- For STAbs, it is initialized with a STTerm and a STType.

```
class STAbs < STTerm
    attr_reader :t1
    attr_reader :t2
    def initialize(t1,t2)
      unless t1.is_a?(STType) && t2.is_a?(STTerm)
throw "Constructing a lambda term out of non-lambda terms"
      end
      @t1 = t1; @t2 = t2
    end

    def ==(type); type.is_a?(STAbs) && type.t1==@t1 && type.t2==@t2 end
end
```

- For STTest, it is initialized with three STTerm.

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

   # We require our variables are only indexed by integers.
   def initialize(t1,t2,t3)
     unless t1.is_a?(STTerm) && t2.is_a?(STTerm) && t2.is_a?(STTerm)
       throw "Constructing a lambda term out of non-lambda terms"
     end
     @t1 = t1; @t2 = t2; @t3=t3
   end

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

3 Part two: Typechecking

This method takes a STTerm, and returns true if the represented term obeys the type rules of ST; otherwise, it returns false.

3.1 Scala implementation

3.1.1 typeOf

This method determine a type for the input STTerm. It has two arguments, one is the STTerm, the other one is an typing context, here an empty list is used. It return a Option[STType].

- Acccording to the typing rule, for STVar, the type is given by the environment

```
case STVar(index) => None
```

- For STTrue, STFalse, the type is STBool

```
case STTrue | STFalse => Some(STBool)
```

- For STIsZero, if the type of its parameter t is STNat, then the type of STBool

```
case STIsZero(t) => if (typeOf(t,List[STType]())==Some(STNat)) Some(STBool) else None
```

- For STZero, the type is STNat
- For STSuc, if the type of its parameter t is STNat, then the type is STNat

```
case STZero => Some(STNat)
case STSuc(t) => if (typeOf(t,List[STType]())==Some(STNat)) Some(STNat) else None
```

- For STAbs, we add the type to the list(environment) first, if the index(index parameter of the STVar) at the list is of type t, return A -> A.

  If it is a free variable, return None. If it is not a STVar, use recursion to find the type of the term

```
case STAbs(t,term) =>
  term match {
    case STVar(index) if ((l:+(t)).length < index) => print((t::l).length< index); None
    case STVar(index) if (((l:+(t)).lift(index).get)==t) => Some(STFun(t,STNat))
    case STApp(t1, t2) => ((l:+(t)).lift(0).get) match {case STFun(dom,codom) if (dom==
    case _ => Some(STFun(t,typeOf(term,l:+(t)).get))
  }
```

- For STApp, we add the type of t1 to the list and get the first element of the list.

  If it is of type A -> B, check if type1 is the same as type of t2, if so, the general type is type2, if not, return None.

```
case STApp(t1, t2) => (typeOf(t1,List[STType]())::l)(0) match {
    case Some(STFun(type1,type2)) if Some(type1)==typeOf(t2,List[STType]()) => print((
    case _ => None
  }
```

3.1.2 typecheck

This method takes an STTerm, and returns true if the represented term obeys the type rules of ST; otherwise, it returns false.

The typeOf method is called in this method, if it return some type, the result is true, else it is false. Also, if exception occurs, it is false

```
def typecheck(input:STTerm):Boolean= try {
  if (typeOf(input, List[STType]())==None) {
    return false
  }
  else {
    return true
  }
}
  catch {
      case _: Throwable => false
  }
```

3.2 Ruby implementation

3.2.1 typeOf

This method determine a type for the input STTerm. As it is implemented in each class, an empty environment is passed as argument. Here, an empty array is used.

- For STVar, the type is given by the environment

```
def typeOf(arr); nil end
```

- For STTrue, STFalse, the type is STBool

```
def typeOf(arr); STBool.new end
```

- For STIsZero, if the type of its parameter t is STNat, then the type of STBool

```
def typeOf(arr)
      if t.typeOf(Array.new)==STNat.new
return STBool.new
      else
return nil
      end
    end
```

- For STZero, the type is STNat

```
def typeOf(arr); STNat.new end
```

- For STAbs, if t2 is of type STVar, we add the type to the list(environment) first, and the index(index parameter of the STVar) at the list is of type t1, return A -> A.

  If it is a free variable, return None. If it is not a STVar, use recursion to find the type of the term

```
def typeOf(arr)
  case t2
  when STVar
    if (arr<<t1)[t2.index]==t1
      return STFun.new(t1,STNat.new)
  else
   return nil
  end
 when STApp
  if (arr<<t1)[0].dom == t1
   return STNat.new
  end
 else
  return STFun.new(t1,t2.typeOf(arr<<t1))
 end
end
```

- For `STApp`, we check the type of t1, if it is `STFun`, we then check if its dom is the same as the type of t2.

  If so, return its condom, else, return nil

```
def typeOf(arr)
   case t1.typeOf(Array.new)
   when STFun
if t1.typeOf(Array.new).dom == t2.typeOf(Array.new)
 return t1.typeOf(Array.new).codom
else
 return nil
end
   else
return nil
   end
  end
```

3.2.2 typecheck

This method takes an `STTerm`, and returns `true` if the represented term obeys the type rules of ST; otherwise, it returns false.

The `typeOf` method is called in this method, if it return some type, the result is true, else it is false. Also, as this method applies to all class, it is defined in the super class STTerm

```
class STTerm
 def typecheck
  if typeOf(Array.new)==nil
   return false
```

```
    else
      return true
    end
  end
end
```

4 Part three: Translation to the untyped -calculus

This method translates a STTerm into elements of ULTerm.

4.1 Scala implementation

- For `STVar`, `STTrue`, `STFalse`, `STZero` they are translated directly

```
case STVar(index) => ULVar(index)
case STTrue => ULAbs(ULAbs(ULVar(1)))
case STFalse => ULAbs(ULAbs(ULVar(0)))
case STZero => ULAbs(ULAbs(ULVar(0)))
```

- For `STSuc(t)`, the `Suc` part is default, and `ULApp` is used to combine it with t in untyped lambda form.

```
case STSuc(t) => ULApp(ULAbs(ULAbs(ULAbs(ULApp(ULVar(1),ULApp(ULApp(ULVar(2),ULVar(1))
```

- For `STApp` and `STAbs`, they are translated using recursion

```
case STApp(t1, t2) => ULApp(eraseTypes(t1),eraseTypes(t2))
case STAbs(t, term) => ULAbs(eraseTypes(term))
```

4.2 Ruby implementation

- For `STVar`, `STTrue`, `STFalse`, `STZero` they are translated directly

```
def eraseTypes; ULVar.new(index) end
def eraseTypes; ULAbs.new(ULAbs.new(ULVar.new(1))) end
def eraseTypes; ULAbs.new(ULAbs.new(ULVar.new(0))) end
def eraseTypes; ULAbs.new(ULAbs.new(ULVar.new(0))) end
```

- For `STSuc(t)`, the `Suc` part is default, and `ULApp` is used to combine it with t in untyped lambda form.

```
def eraseTypes
    return 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.e
end
```

- For `STApp` and `STAbs`, they are translated using recursion

```
def eraseTypes; ULApp.new(t1.eraseTypes,t2.eraseTypes) end
def eraseTypes; ULAbs.new(t2.eraseTypes) end
```