

Computer Science 3MI3 – 2020 assignment 2

Typing a λ -calculus

Mark Armstrong

October 30th, 2020

Contents

Introduction

This assignment asks you to construct representation of a simply-typed λ -calculus, construct a typechecker for that λ -calculus, and finally to implement *type-erasure* and a simple translator to simplify terms to untyped λ -calculus terms.

Updates and file history

November 10th

- The testing section has been completed.
- The subclasses of the `ULTerm` class in Ruby have had a `==` method added to them. This method is necessary for the testing of Part 3, and you will need to copy its definitions or re-download the [a2_ulterm.rb](#) file.
- Notes about the means of implementing type checking have been added.
- The names of the types for natural numbers and booleans in the *ST* language have been shortened, to `nat` and `bool`.
 - Classes to represent these types in both Scala and Ruby have also been provided.

November 6th

- The typing rules for the *ST* language were added in part 0.1.

November 5th

- A typo in a variable name in the Ruby implementation of `ULTerm` was corrected.
- The provided Scala code for the `ULTerm` type was modified slightly to include better `toString` methods.
- Example code showing how to construct `ULTerm` terms and perform substitutions with them was added to part 0.2.

November 1st

- Part 4 was made bonus
 - and the task of translating from `ULTerm`'s to `STTerm`'s was made part of the question.

October 30th

- Initial version posted.
 - Testing not posted yet.

Boilerplate

Documentation

In addition to the code for the assignments, you are required to submit (relatively light) documentation, along the lines of that found in [the literate programs](#) from lectures and tutorials.

- Those occasionally include a lot of writing when introducing concepts; you do not have to introduce concepts, so your documentation should be similar to the *end* of those documents, where only the purpose and implementation details of types, functions, etc., are discussed.

This documentation is not assigned its own marks; rather, 20% of the marks of each part of the assignment will be for the documentation.

This documentation **must be** in the literate style, with (nicely typeset) English paragraphs alongside code snippets; comments in your source code do not count. The basic requirement is

- the English paragraphs must use non-fixed width font, whereas
- the code snippets must use fixed width font.
- For example, see these lecture notes on Prolog:
 - <https://courses.cs.washington.edu/courses/cse341/98sp/logic/prolog.html>

But you are encouraged to strive for nicer than just “the basic requirement”. (the ability to write decent looking documentation is an asset!

You are free to present your documentation in any of these formats:

- an HTML file,
 - (named `README.html`)
- a PDF (for instance, by writing it in \LaTeX using the `listings` or `minted` package for your code blocks),
 - (named `README.pdf`), or
- rendering on GitLab (for instance, by writing it in markdown or Org)
 - (named `README.md` or `README.org`.)

If you wish to use another format, contact Mark to discuss it.

Not all of your code needs to be shown; only portions which are of interest are needed. Feel free to omit some “repetitive” portions. (For instance, if there are several cases in a definition which look almost identical, only one or two need to be shown.)

Submission procedures

The same guidelines as for homework (which can be seen in any of the homework files) apply to assignments, except for the differences below.

Assignment naming requirements

Place all files for the assignment inside a folder titled **an**, where **n** is the number of the assignment. So, for assignment 1, use the folder **a1**, for assignment 2 the folder **a2**, etc. Ensure you do not capitalise the **a**.

Each part of the assignments will direct you on where to save your code for that part. Follow those instructions!

If the language supports multiple different file extensions, you must still follow the extension conventions noted in the assignment.

Incorrect naming of files may result in up to a 5% deduction in your grade.

This is slightly decreased from the 10% for homeworks.

Proper conduct for coursework

Refer to the homework code of conduct available in any of the homework files. The same guidelines apply to assignments.

Part 0.1 – Description of the λ -calculus, *ST* [0 marks]

The λ -calculus you are to work with during this assignment we call *ST*, standing for *simply typed*. It adds to the pure untyped λ -calculus *UL* terms *zero*, *suc*, *iszero*, *tt*, *ff*, and *test*, with the following syntax.

```
 $\langle \text{typedterm} \rangle ::= \text{var}$   
                  |  $\langle \text{typedterm} \rangle \langle \text{typedterm} \rangle$   
                  |  $\lambda \text{ var} : \langle \text{type} \rangle \rightarrow \langle \text{typedterm} \rangle$   
                  | zero  
                  | suc  $\langle \text{typedterm} \rangle$   
                  | iszero  $\langle \text{typedterm} \rangle$   
                  | true  
                  | false  
                  | test  $\langle \text{typedterm} \rangle \langle \text{typedterm} \rangle \langle \text{typedterm} \rangle$ 
```

```
 $\langle \text{type} \rangle ::= \langle \text{type} \rangle \rightarrow \langle \text{type} \rangle$   
          | nat  
          | bool
```

We also introduce the following *typing rules* for these typed λ -terms. The rules make use of a *typing context* or *type environment* Γ .

The first rule says that variables have the type they are given by the environment Γ (assuming they are given a type at all.)

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \text{ T-Var}$$

The second rule says that if by adding “ x has type A ” to the environment, we can conclude that t_2 has type B , then the term $\lambda x : A \rightarrow t_2$ has type $A \rightarrow B$. (Notice that this rule is the only time we add to the environment.)

$$\frac{\Gamma, (x : A) \vdash t_2 : B}{\Gamma \vdash \lambda x : A \rightarrow t_2 : A \rightarrow B} \text{ T-Abs}$$

The third rule says that if t_1 has the function type $A \rightarrow B$, and t_2 has the type A , then t_2 applied to t_1 has type B .

$$\frac{\Gamma \vdash t_1 : A \rightarrow B \quad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B} \text{ T-App}$$

The remaining rules give the typings for the constants and function terms added to this language.

$$\frac{}{\Gamma \vdash \text{zero} : \text{nat}} \text{ T-zero} \qquad \frac{\Gamma \vdash t : \text{nat}}{\Gamma \vdash \text{suc } t : \text{nat}} \text{ T-suc}$$

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \text{ T-true} \qquad \frac{}{\Gamma \vdash \text{false} : \text{bool}} \text{ T-false}$$

$$\frac{\Gamma \vdash t : \text{nat}}{\Gamma \vdash \text{iszero } t : \text{bool}} \text{ T-iszero}$$

$$\Gamma \vdash b : \text{bool} \quad \Gamma \vdash t_1 : A \quad \Gamma \vdash t_2 : A$$

$$\Gamma \vdash \text{test } b \ t_1 \ t_2 : A$$

Part 0.2 – A representation of the untyped λ -calculus, *UL* [0 marks]

Nameless representation of terms

We use *de Bruijn indices* in place of named variables. The index “points” to a binder, or to a free variable.

- 0 points to the first enclosing variable binder, or the first free variable if there are no enclosing binders.
- 1 points to the second enclosing variable binder, or the 2-*n*’th free variable if there are only *n* enclosing binders, $n \leq 1$.
- 2 points to the third enclosing variable binder, or the 3-*n*’th free variable if there are only *n* enclosing binders, $n \leq 2$.
- ...
- *i* points to the *i*’th enclosing variable binder, or the (*i*+1)-*n*’th free variable if there are only *n* enclosing binders, $n \leq i$.
- ...

This representation avoids any need for renaming variables during substitution.

It does make terms less human readable; we can correct for this by writing a *pretty printer* for λ -terms (which will be the focus of a homework.)

Scala implementation

Pure untyped λ -terms can only be variables, abstractions or applications. (Updated November 5th) We include as parts of the case classes overrides of the `toString` method, which improve the appearance of these terms when they are converted to strings.

```
sealed trait ULTerm
case class ULVar(index: Int) extends ULTerm {
  override def toString() = index.toString()
}
```

```

case class ULAbs(t: ULTerm) extends ULTerm {
  override def toString() = "lambda . " + t.toString()
}
case class ULApp(t1: ULTerm, t2: ULTerm) extends ULTerm {
  override def toString() = "(" + t1.toString() + ") (" +
    ↪ t2.toString() + ")"
}

```

The use of de Bruijn indices necessitates a method to “shift” the indices of free variables up or down; for instance, when applying a term to an abstraction, we must shift them up to avoid capturing what should be free variables in a variable binder.

Shifting is done by walking through the term, incrementing the variable indices by the shift amount if their index is greater than the number of enclosing binders.

```

// Shift the numbering of unbound variables
def shift(shiftAmount: Int, t: ULTerm): ULTerm = {
  // Walk through the term and shift all variables with index
  // greater than or equal to c, which is maintained to be
  // the number of variable binders (abstractions) outside the
  ↪ current subterm.
  def walk(currentBinders: Int, t: ULTerm): ULTerm = t match {
    // Check if x is a free variable; that is,
    // if the number x is greater than or equal to
    // the number of variable binders encountered outside this
    ↪ subterm.
    case ULVar(x) if (x >= currentBinders) =>
      ↪ ULVar(x+shiftAmount)
    case ULVar(x) => ULVar(x)

    case ULAbs(t) =>
      // We now have one more variable binder outside the
      ↪ subterm.
      // Increment currentBinders and walk into the subterm.
      ULAbs(walk(currentBinders+1, t))

    case ULApp(t1,t2) =>
      // No new variable binders. Just walk into the subterms.
      ULApp(walk(currentBinders,t1),walk(currentBinders,t2))
  }
}

```

```

    // Walk the term and perform the shift of free variables.
    // We begin with 0 variable binders outside.
    walk(0, t)
}

```

Substitution is similarly defined by “walking” through the term, but here, when we find variables, we choose whether to “replace them” by the term being subbed in or not. We have to adjust the variable being substituted and the free variables in the term being subbed in according to the number of variable binders we enter.

```

// In our usual syntax, we would write substitution as `t[x :=
↪ r]`.
// Here we write `substitute(t,x,r)`.
def substitute(t: ULTerm, x: Int, r: ULTerm): ULTerm = {
    // We want to substitute for the free variable with number
    ↪ x.
    // Inside a variable binder (abstraction),
    // the index of all free variables is shifted up by 1.
    // So we must keep track of the number of binders outside
    ↪ the current subterm.
    def walk(currentBinders: Int, t: ULTerm): ULTerm = t match {
        case ULVar(y) if y == x + currentBinders =>
            // y is the xth free variable. Substitute for it,
            // making sure to shift the free variables in r
            // to account for the number of variable binders outside
            ↪ this subterm.
            shift(currentBinders,r)

        case ULVar(y) =>
            // Otherwise, y is not the xth free variable;
            // leave it as is.
            ULVar(y)

        case ULAbs(t) =>
            // We now have one more variable binder outside the
            ↪ subterm.
            // Increment currentBinders and walk into the subterm.
            ULAbs(walk(currentBinders+1,t))
    }
}

```



```

    case ULApp(t1,t2) =>
        // No new variable binders. Just walk into the subterms.
        ULApp(walk(currentBinders,t1),walk(currentBinders,t2))
}

// Walk the term, performing the substitution.
// We begin with 0 variable binders outside.
walk(0,t)
}

```

We need to check if terms are values for call-by-value semantics.

```

// We need to know if a term is a value during reduction
// when using call-by-value semantics.
def isValue(t: ULTerm): Boolean = t match {
    case ULAbs(_) => true
    case _ => false
}

```

Those semantics are given by a reduction function, which reduces terms by one step, and then an evaluation function, which keeps reducing until we get stuck (if we get stuck; we might have an infinite reduction sequence.)

```

// Call-by-value reduction function.
// Performs one step of evaluation, if possible according to
// ↪ the call-by-value rules.
// If no reduction is possible, returns None.
def reduce(t: ULTerm): Option[ULTerm] = t match {

    // Case: the left term is an abstraction, and the right is a
    // ↪ value.
    // Then apply the value to the abstraction.
    case ULApp(ULAbs(t),v) if isValue(v) =>
        // When we apply the value to the abstraction,
        // we must shift the value's free variables up by 1 to
        // ↪ account
        // for the abstraction's variable binder.
        val r = substitute(t,0,shift(1,v))
        // Then, we need to shift the result back.
        // Since the abstraction's variable is now "used up".
        Some(shift(-1,r))
}

```

```

// Case: the left term is a value, then try to reduce the
    ↪ right term.
case ULApp(v,t) if isValue(v) =>
  reduce(t) match {
    case Some(r) => Some(ULApp(v,r))
    case None => None
  }

// Case: the left term is not a value (not an abstraction.)
// Try to reduce it.
case ULApp(t1,t2) =>
  reduce(t1) match {
    case Some(r1) => Some(ULApp(r1,t2))
    case None => None
  }

case _ => None
}

// Evaluation just repeatedly applies reduce,
// until we reach None (signifying reduction failed.)
def evaluate(t: ULTerm): ULTerm = reduce(t) match {
  case None => t
  case Some(r) => evaluate(r)
}

```

Ruby implementation

In Ruby, we use implement *UL* terms using a (super) class `ULTerm` with subclasses for each kind of *UL* term.

The super class defines so default methods to keep track of what kind of term we have. These could be implemented as fields (instance variables), but the use of methods implies that these values are constant for all objects and across all time.

```

# Our top-level ULTerm class defines some default
# methods to track what kind of term we have
# (which must be overridden in non-default cases)
# as well as the shift, substitute and eval methods

```

```

# which are defined in terms of other methods
# defined by the subclasses.
class ULTerm

  # By default, we assume terms are irreducible,
  # not abstractions, and not values.
  # Subclasses which should have these properties
  # must override these methods.
  # (In our basic calculus with call-by-value semantics,
  # only applications are reducible and only abstractions
  # are values. This can be changed for different
  ↪ calculi/semantics.)
  def reduce; nil end
  def absBody; nil end
  def isValue?; false end

```

We would not usually have enough information in this super `ULTerm` class to be able to define the `shift` and `substitution` methods, without resorting to (what I feel is) an ugly approach of using `is_a?` to check whether the term is a variable, abstraction or application.

Previously, we used a local `walk` method inside of each of the methods, which actually carried out the work on the terms. We could do the same here, repeating the definition of `walk` inside of the `shift` and `substitute` methods for each type of term (at least, I believe we can do so.) However, this repetition of code is very undesirable.

Instead, we take advantage of the fact that the `walk` method was in fact *almost identical* for both shifting and substitution, only acting differently on *variables*, to reimagine `walk` as an iterator. This iterator will take as a block argument (a lambda) the action to carry out on variables.

So, assuming that the `walk` method will be defined for each of the subclasses, we can define `shift` and `substitute` here in the superclass by writing the action to take on variables as a block, and calling `walk` on the term with that action.

```

# Shifting is just walking, where in the base case,
# we either increment the variable by shiftAmount or
# leave it alone.
def shift(shiftAmount)
  # walk is an iterator.
  # The block tells us what to do with variables.
  walk(0) { |x,currentBinders|

```

```

    if x >= currentBinders
      ULVar.new(x+shiftAmount)
    else
      ULVar.new(x)
    end }
  end

  # Substitution is just walking, where we either
  # replace the variable, or leave it alone.
  def substitute(x,r)
    walk(0) { |y,currentBinders|
      if y == x + currentBinders
        r
      else
        ULVar.new(y)
      end }
  end
end

```

Similarly to how we assume above that the `walk` method will be defined for all subclasses, we also assume that the `reduce` method will be defined for all subclasses, since we lack a nice means to define it here. However, we can easily define `eval` in terms of those `reduce` methods.

```

def eval
  r = nil
  r_next = self
  # Keep reducing until it fails (reduce returns nil.)
  # This is the recommended "do...while" form in Ruby.
  loop do
    r = r_next
    r_next = r.reduce
    break unless r_next
  end

  return r
end
end

```

As mentioned above, in each of the subclasses of `ULTerm`, we need to define the `walk` and `reduce` methods. But for variables, reduction is undefined, so we do not define that method here. (We do define here and below `to_s` methods, to allow these terms to be printed somewhat nicely.)

```

class UVar < UTerm
  attr_reader :index

  # We require our variables are only indexed by integers.
  def initialize(i)
    unless i.is_a?(Integer)
      throw "Constructing a lambda term out of non-lambda
        ↪ terms"
    end
    @index = i
  end

  def walk(currentBinders, &block)
    # This is a variable. Run the code in &block.
    # (yield does this; it "yields" control to the block.)
    yield(@index, currentBinders)
  end

  def to_s
    @index.to_s
  end

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

```

Again, we cannot reduce an abstraction, so we do not define that method here. But we do set override the `absBody` and `isValue?` methods since this is an abstraction, and abstractions are values.

```

class ULAbs < UTerm
  attr_reader :t

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

  def walk(currentBinders, &block)

```

```

    # Increment the local variable counter within the variable
    ↪ binder.
    t = @t.walk(currentBinders+1,&block)
    ULAbs.new(t)
end

# Abstractions are an abstraction (of course),
# with body @t,
# and are also considered values.
def absBody; @t end
def isValue?; true end

def to_s
  "lambda . " + @t.to_s
end

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

```

The application subclass is actually the only one where we define the `reduce` method. The logic of it is the same as in the Scala version, though unfortunately the lack of pattern matching makes it appear much worse. (The source code was edited November 6th to remove some diagnostic printing statements which were unfortunately included previously.)

```

class ULApp < ULTerm
  attr_reader :t1
  attr_reader :t2

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

  def walk(currentBinders,&block)
    t1 = @t1.walk(currentBinders,&block)
    t2 = @t2.walk(currentBinders,&block)
    ULApp.new(t1,t2)
  end
end

```

```

end

# Applications can be reduced.
def reduce
  if @t1.absBody && @t2.isValue?
    body = @t1.absBody
    (body.substitute(0,@t2.shift(1))).shift(-1)
  elsif @t1.isValue?
    r = @t2.reduce
    if r
      ULApp.new(@t1,r)
    else
      nil
    end
  else
    r = @t1.reduce
    if r
      ULApp.new(r,@t2)
    else
      nil
    end
  end
end

def to_s
  "(" + @t1.to_s + ") (" + @t2.to_s + ")"
end

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

```

One important fact bears mentioning about our implementation of `ULTerm` and its subclasses: note that all of the fields of each class are read-only, and that their values are only ever set in the constructors. These are *value classes*; a `ULTerm` object is intended to be (and will be, barring any misuse) *immutable* (unchanging over time.)

This design leads to better predictability of code; there should never be an instance where a `ULTerm` changes unexpectedly because of some method call, because `ULTerm`'s never change after their creation.

Examples of interacting with these representations

The following code snippets show how you might use these implementations to perform some simple computations.

In Scala:

```
// The term "lambda x . lambda y . lambda z . u (x (y z))"
// Note the first variable (The one initialised with UVar(3))
  ↪ is free,
// because it's index is greater than the number of
  ↪ abstractions
// surrounding it.
val x = ULAbs(
    ULAbs(
        ULAbs(ULApp(ULVar(3),
                    ULApp(ULVar(2),
                        ULApp(ULVar(1),
                            ULVar(0))))))
println("An unnamed representation of lambda x . lambda y .
  ↪ lambda z -> u x y z:")
print("\t")
println(x)

// Now substitute that term itself in for the free variable.
println("The result of substituting that term into itself for
  ↪ the variable u:")
print("\t")
println(substitute(x,0,x))
```

And in Ruby:

```
# The term "lambda x . lambda y . lambda z . u (x (y z))"
# Note the first variable (The one initialised with
  ↪ UVar.new(3)) is free,
# because it's index is greater than the number of
  ↪ abstractions
# surrounding it.
x = ULAbs.new(
    ULAbs.new(
        ULAbs.new(ULApp.new(ULVar.new(3),
                            ULApp.new(ULVar.new(2),
```



```

                                ULApp.new(ULVar.new(1),
                                                ↪ ULVar.new(0))))))
puts "An unnamed representation of lambda x . lambda y .
    ↪ lambda z -> u x y z:"
print "\t"
puts x

# Now substitute that term itself in for the free variable.
puts "The result of substituting that term into itself for the
    ↪ variable u:"
print "\t"
puts x.substitute(0,x)

# Note that the term itself remains unchanged;
# we've made sure this type is immutable
# by always creating new terms, or reusing them if that's not
    ↪ necessary,
# in the class methods. The fields are only ever changed in
    ↪ the constructors.
puts x

```

Part 1 – The representation [10 marks]

Place your code for this part in the files `a2.sc` and `a2.rb`.

Implement, in both Scala and Ruby, a type `STTerm` to represent terms of the λ -calculus ST defined above.

The constructors of the type should be named

- `STVar`,
- `STApp`,
- `STAbs`,
- `STZero`,
- `STSuc`,
- `STIsZero`,

- STTrue,
- STFalse, and
- STTest.

(The remaining text of this section was added November 10th.)

The STZero, STTrue and STFalse types should be declared as singleton classes in Scala (i.e., as `case object`'s instead of `case class`'s.)

Note that the STAbs constructor must take an argument specifying the *type* of the variable being abstracted. This argument should be of the following types. (This code should be included in your a2 file.) In [Scala](#):

```
sealed trait STType
case object STNat extends STType {
  override def toString() = "nat"
}
case object STBool extends STType {
  override def toString() = "bool"
}
// Functions have a domain type and a codomain type.
case class STFun(dom: STType, codom: STType) extends STType {
  override def toString() = "(" + dom.toString + ") -> (" +
    ↪ codom.toString + ")"
}

// Example use: the type "nat -> bool" is written
↪ STFun(STNat,STBool)
```

And in [Ruby](#):

```
class STType end

class STNat < STType
  # Comparison and printing methods
  def ==(type); type.is_a?(STNat) end
  def to_s; "nat" end
end

class STBool < STType
  # Comparison and printing methods
  def ==(type); type.is_a?(STBool) end
```

```

    def to_s; "bool" end
end

# Functions have a domain type and a codomain type.
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

  # Comparison and printing methods
  def ==(type); type.is_a?(STFun) && type.dom == @dom &&
    ↪ type.codom == @codom end
  def to_s; "(" + dom.to_s + ") -> (" + codom.to_s + ")" end
end

# Example use: the type "nat -> bool" is written
↪ STFun.new(STNat.new, STBool.new)

```

Part 2 – Type checking [40 marks]

Place your code for this part in the files `a2.sc` and `a2.rb`.

Implement, in both Scala and Ruby, a *type checker* method named `typecheck` for elements of `STTerm`.

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

(The remaining text of this section was added November 10th.)

The definition of `typecheck` will require a method to determine the type of a given expression. The name `typeOf` is recommended for this method. This method should have as an argument the *environment*; i.e., a map, list or similar type which relates variables to types. (Since variables are represented as integers, this means relating integers to `STType` values.)

I (Mark) *suggest* that `typeOf` method return a `Option[STType]` in the

Scala implementation, but this is not a requirement. You will not be penalised for using any reasonable implementation, including

- use of exceptions when a term has no type,
- use of the `Try`, `Success` and `Failure` types, or
- use of the `Either` type.

If you have another implementation in mind, you are welcome to check with Mark.

Part 3 – Translation to the untyped λ -calculus; type erasure [40 marks]

Place your code for this part in the files `a2.sc` and `a2.rb`.

Implement, in both Scala and Ruby, a method `eraseTypes` for elements of `STTerm`, which *translates* them into elements of `ULTerm` (definition given above.)

This translation also needs to translate the natural and boolean constants into the pure λ -calculus encodings that represent them. Recall these translations from the notes (the translation for `iszero` has been added.)

```
true   =  $\lambda t \rightarrow \lambda f \rightarrow t$ 
false  =  $\lambda t \rightarrow \lambda f \rightarrow f$ 
test   =  $\lambda l \rightarrow \lambda m \rightarrow \lambda n \rightarrow l\ m\ n$ 
zero   =  $\lambda s \rightarrow \lambda z \rightarrow z$ 
suc     =  $\lambda n \rightarrow \lambda s \rightarrow \lambda z \rightarrow s\ (n\ s\ z)$ 
iszero =  $\lambda m \rightarrow m\ (\lambda x \rightarrow \text{false})\ \text{true}$ 
```

Bear in mind that application is left-associative, so, e.g., `l m n` is equivalent to `(l m) n`.

(You should import the `a2_ulterm` files into your file in a manner compatible with the Docker testing environments, or copy the definitions into your file and create blank `a2_ulterm` files to satisfy the import statements in the testing.)

Part 4 – Bonus: Interpreting *SL* programs [10 marks]

Place any code for this part in files `a2p4.sc` and `a2p4.rb`.

Implement an evaluation method for your `STTerm` type.

Make use of the evaluation method for `ULTerm`'s in your definition. You will also need a method to convert results back to an `STTerm` representation.

Part 5 – Bonus: pairs [10 bonus marks]

Place any code for this part in files `a2p5.sc` and `a2p5.rb`.

Implement another λ -calculus, called *ST2*, which includes the type of *pairs* as well as naturals and booleans, along with a type checker, type eraser and evaluation method.

Submission checklist

For your convenience, this checklist is provided to track the files you need to submit. Use it if you wish.

- [] Documentation; one of
 - [] README.html
 - [] README.pdf
 - [] README.md
 - [] README.org
- [] Code files
 - [] a2.sc
 - [] a2.rb
- [] Part 2 tests
 - [] a2p2_test.sc tests have passed! (No submission
 ↪ needed.)
 - [] a2p2_test.rb tests have passed! (No submission
 ↪ needed.)
- [] Part 3 tests
 - [] a2p3_test.sc tests have passed! (No submission
 ↪ needed.)
 - [] a2p3_test.rb tests have passed! (No submission
 ↪ needed.)
- [] Part 4 (Bonus)
 - [] a2p4.sc
 - [] a2p4.rb

Testing

Unit tests for the requested types, methods and predicates are available here.

- [a2_test.sc](#)
- [a2_test.rb](#)

The contents of the unit test files are also repeated below.

The tests can be run by placing the test files in the same directory as your code files.

To run the tests for the Scala portions, use the command

```
amm a2_test.sc
```

To run the tests for the Ruby portions, use the commands

```
ruby a2_test.rb
```

You are strongly encouraged to add your own additional test cases to those provided for you.

The provided test cases check a very minimal amount!

Automated testing via Docker

The Docker setup and usage scripts are available at the following links. Their contents are also repeated below.

- [Dockerfile](#)
- [docker-compose.yml](#)
- [setup.sh](#)
- [run.sh](#)

Place them into your `a2` directory where your code files and the test files (linked to above) exist, then run `setup.sh` and `run.sh`.

Note that the use of the `setup.sh` and `run.sh` scripts assumes that you are in a `bash` like shell; if you are on Windows, and not using WSL or WSL2, you may have to run the commands contained in those scripts manually.

The tests

Scala

a2_test.sc

```
import $file.a2, a2._
import $file.a2_ulterm, a2_ulterm._

/* Given an expected result and a computed result,
   check if they are equal in value.
   If so, return 0. Otherwise, inform the user, and return 1,
   so the number of failures can be counted. */
def test[A](given: A, expected: A, the_test: String) =
  if (!(given equals expected)) {

    ↪ println("+-----")
    println("| " + the_test + " failed.")
    println("| Expected " + expected + ", got " + given + ".")

    ↪ println("+-----")
    1
  } else {
    0
  }

val providedTests : List[Tuple3[Unit => Any, Any, String]] =
  ↪ List(

    // Positive typechecking test cases
    (_ => typecheck(STZero), true,
     ↪ "Typecheck zero"),
    (_ => typecheck(STSuc(STZero)), true,
     ↪ "Typecheck one"),
    (_ => typecheck(STIsZero(STZero)), true,
     ↪ "Typecheck iszero zero"),
    (_ => typecheck(STTrue), true,
     ↪ "Typecheck true"),
    (_ => typecheck(STFalse), true,
     ↪ "Typecheck false"),
```

```

( _ => typecheck(STTest(STTrue,STZero,STZero)), true,
  ↪ "Typecheck test true zero zero"),
( _ => typecheck(STAbs(STNat,STVar(0))), true,
  ↪ "Typecheck lambda x : nat . x"),
( _ => typecheck(STApp(STAbs(STNat,STVar(0)),STZero)), true,
  ↪ "Typecheck (lambda x : nat . x) (zero)"),

// Negative typechecking test cases
( _ => typecheck(STSuc(STFalse)), false,
  "Successor of false should not typecheck."),
( _ => typecheck(STVar(0)), false,
  "Free variables should not typecheck"),
( _ => typecheck(STApp(STAbs(STNat,STVar(0)),STFalse)),
  ↪ false,
  "Applying a boolean to a natural function should not
  ↪ typecheck"),

// Type erasure tests
( _ => eraseTypes(STTrue), ULAbs(ULAbs(ULVar(1))), "Erase
  ↪ types of true"),
( _ => eraseTypes(STSuc(STZero)),
  ULApp(
    // Encoding of suc
    ULAbs( // lambda n .
      ULAbs( // lambda s .
        ULAbs( // lambda z .
          ↪ ULApp(ULVar(1),ULApp(ULApp(ULVar(2),ULVar(1)),ULVar(0))))),
          ↪ // s (n s z)
        // Encoding of zero
        ULAbs( // lambda s .
          ULAbs( // lambda z .
            ULVar(0))), // z
        "Erase types of one"),
    ↪ eraseTypes(STApp(STAbs(STNat,STVar(0)),STZero)),
    ULApp(ULAbs(ULVar(0)), ULAbs(ULAbs(ULVar(0)))),
    "Erase types of (lambda x : nat . x) (zero)"),
  )

```



```

def runTests[A](tests: List[Tuple3[Unit => A,A,String]]): Unit
  ↪ = {
    // Apply test to each element of tests, and sum the return
    ↪ values.
    // This is essentially a for loop.
    val failed = tests.foldLeft(0) {
      (failures, next) => next match {
        // Deconstruct the tuple to get its parts
        case (given, expected, the_test) => {
          val failure = try {
            test(given(), expected, the_test)
          } catch {
            case e: Exception => {
              print("Test " + the_test + "encountered exception:
                ↪ ")
              println(e)
              1
            }
          }
          failures + failure
        }
      }
    }

    ↪ println("+-----")
    println("| " + failed + " tests failed")

    ↪ println("+-----")
  }

```

```
runTests(providedTests)
```

Ruby

```
a2_test.rb
```

```

require_relative "a2"
require_relative "a2_ulterm"
require "test/unit"

```

```

class ProvidedTests < Test::Unit::TestCase

  # Positive type checking tests
  def test_simple0
    assert_equal(true, STZero.new.typecheck, "Typecheck zero")
  end
  def test_simple1
    assert_equal(true, STSuc.new(STZero.new).typecheck,
      ↪ "Typecheck one")
  end
  def test_simple2
    assert_equal(true, STIsZero.new(STZero.new).typecheck,
      ↪ "Typecheck iszero zero")
  end
  def test_simple3
    assert_equal(true, STTrue.new.typecheck, "Typecheck true")
  end
  def test_simple4
    assert_equal(true, STFalse.new.typecheck, "Typecheck
      ↪ false")
  end
  def test_simple5
    assert_equal(true,
      ↪ STTest.new(STTrue.new,STZero.new,STZero.new).typecheck,
      "Typecheck test true zero zero")
  end
  def test_simple6
    assert_equal(true,
      STAbs.new(STNat.new,STVar.new(0)).typecheck,
      "Typecheck lambda x : nat . x")
  end
  def test_simple7
    assert_equal(true,
      ↪ STApp.new(STAbs.new(STNat.new,STVar.new(0)),STZero.new).typecheck,
      "Typecheck (lambda x : nat . x) (zero)")
  end

  # Negative typechecking tests

```

```

def test_simple8
  assert_equal(false,
    STSuc.new(STFalse.new).typecheck,
    "Successor of false should not typecheck.")
end
def test_simple9
  assert_equal(false,
    STVar.new(0).typecheck,
    "Free variables should not typecheck")
end
def test_simple10
  assert_equal(false,
    ↪ STApp.new(STAbs.new(STNat.new, STVar.new(0)), STFalse.new).typeche
    "Applying a boolean to a natural function
    ↪ should not typecheck")
end

def test_simple11
  assert_equal(ULAbs.new(ULAbs.new(ULVar.new(1))),
    ↪ STTrue.new.eraseTypes, "Erase types of true")
end
def test_simple12
  assert_equal(ULApp.new(
    # Encoding of suc
    ULAbs.new( # lambda n .
      ULAbs.new( # lambda s .
        ULAbs.new( # lambda z .

          ↪ ULApp.new(ULVar.new(1), ULApp.new(ULApp.new(ULVar.new(2),
          ↪ # s (n s z)
      # Encoding of zero
      ULAbs.new( # lambda s .
        ULAbs.new( # lambda z .
          ULVar.new(0))))), # z
    STSuc.new(STZero.new).eraseTypes,
    "Erase types of one")
  end
def test_simple13

```

```

assert_equal(ULApp.new(ULAbs.new(
  ↪ ULVar.new(0)),ULAbs.new(ULAbs.new(ULVar.new(0)))),
  ↪ STApp.new(STAbs.new(STNat.new,STVar.new(0)),STZero.new).eraseType
  "Erase types of (lambda x : nat . x) (zero)")
end
end

```

The Docker setup

Dockerfile

```

# Define the argument for openjdk version
ARG OPENJDK_TAG=8u232

FROM ruby:2.7.2-buster

# Setup to install Scala
RUN apt-get update && \
  apt-get install scala -y && \
  apt-get install -y curl && \
  sh -c '(echo "#!/usr/bin/env sh" && \
  curl -L
  ↪ https://github.com/lihaoyi/Ammonite/releases/download/2.1.1/2.12-
  ↪ 2.1.1) > /usr/local/bin/amm &&
  ↪ \
  chmod +x /usr/local/bin/amm'
RUN (rm -rf /root/.cache)

# Set the name of the maintainers
MAINTAINER Habib Ghaffari Hadigheh, Mark Armstrong
  ↪ <ghaffh1@mcmaster.ca, armstmp@mcmaster.ca>

# Set the working directory
WORKDIR /opt/a2

  docker-compose.yml

version: '2'
services:
  service:

```

```

build: .
image: 3mi3_a2_docker_image
volumes:
  - ./opt/a2
container_name: 3mi3_a2_container
command: bash -c
  "echo 'Scala testing' &&
  echo
  ↪ '-----'
  ↪ &&
  amm a2_test.sc &&
  printf '\\n\\n\\n' &&
  echo 'Ruby testing' &&
  echo
  ↪ '-----'
  ↪ &&
  ruby a2_test.rb &&
  echo
  ↪ '-----'"

```

[setup.sh](#)

```
docker-compose build --force-rm
```

[run.sh](#)

```

# Run the container
docker-compose up --force-recreate
# Stop the container after finishing the test run
docker-compose stop -t 1

```