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 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.

$$
\begin{array}{lll}
\langle \mathtt{typedterm} \rangle & ::= & \mathtt{var} \\
 & | & \langle \mathtt{typedterm} \rangle \ \langle \mathtt{typedterm} \rangle \\
 & | & \lambda\ \mathtt{var}\ :\ \langle \mathtt{type} \rangle \to \langle \mathtt{typedterm} \rangle \\
 & | & \mathtt{zero} \\
 & | & \mathtt{suc}\ \langle \mathtt{typedterm} \rangle \\
 & | & \mathtt{iszero}\ \langle \mathtt{typedterm} \rangle \\
 & | & \mathtt{true} \\
 & | & \mathtt{false} \\
 & | & \mathtt{test}\ \langle \mathtt{typedterm} \rangle\ \langle \mathtt{typedterm} \rangle\ \langle \mathtt{typedterm} \rangle
\end{array}
$$

$$
\begin{array}{lll}
\langle \mathtt{type} \rangle & ::= & \langle \mathtt{type} \rangle \to \langle \mathtt{type} \rangle \\
 & | & \mathtt{natural} \\
 & | & \mathtt{boolean}
\end{array}
$$

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.)

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

The second rule says that if by adding "`x` has type `A`" to the environment, we can conclude that $\mathtt{t}_2$ has type `B`, then the term $\lambda\ \mathtt{x}\ :\ \mathtt{A} \to \mathtt{t}_2$ has type $\mathtt{A} \to \mathtt{B}$. (Notice that this rule is the only time we add to the environment.)

$$\frac{\Gamma, (\mathtt{x}\ :\ \mathtt{A}) \vdash \mathtt{t}_2\ :\ \mathtt{B}}{\Gamma \vdash \lambda\ \mathtt{x}\ :\ \mathtt{A} \to \mathtt{t}_2\ :\ \mathtt{A} \to \mathtt{B}}\ \text{T-Abs}$$

The third rule says that if $\mathtt{t}_1$ has the function type $\mathtt{A} \to \mathtt{B}$, and $\mathtt{t}_2$ has the type `A`, then $\mathtt{t}_2$ applied to $\mathtt{t}_1$ has type `B`.

$$\frac{\Gamma \vdash \mathtt{t}_1\ :\ \mathtt{A} \to \mathtt{B} \qquad \Gamma \vdash \mathtt{t}_2\ :\ \mathtt{A}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2\ :\ \mathtt{B}}\ \text{T-App}$$

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

$$\frac{}{\Gamma \vdash \texttt{zero : natural}}\ \texttt{T-zero} \qquad \frac{\Gamma \vdash \texttt{t : natural}}{\Gamma \vdash \texttt{suc t : natural}}\ \hookrightarrow\ \texttt{T-suc}$$

$$\frac{}{\Gamma \vdash \texttt{true : boolean}}\ \texttt{T-true} \qquad \frac{}{\Gamma \vdash \texttt{false : boolean}}\ \texttt{T-false}$$

$$\frac{\Gamma \vdash \texttt{t : natural}}{\Gamma \vdash \texttt{iszero t : boolean}}\ \texttt{T-iszero}$$

$$\frac{\Gamma \vdash \texttt{b : boolean} \quad \Gamma \vdash \texttt{t}_1 \texttt{ : A} \quad \Gamma \vdash \texttt{t}_2 \texttt{ : A}}{\Gamma \vdash \texttt{test b t}_1 \texttt{ t}_2 \texttt{ : A}}\ \texttt{T-test}$$

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.

```
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.)
```

```scala
def evaluate(t: ULTerm): ULTerm = reduce(t) match {
  case None => t
  case Some(r) => evaluate(r)
}
```

Ruby implementation

:TODO: Document this literately, as with the Scala above.

```ruby
# Our top-level ULTerm class defines some default
# methods to track what kind of term we have
# (which must be overidden 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

  # 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

  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

class ULVar < ULTerm
  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
end

class ULAbs < ULTerm
  attr_reader :t

  def initialize(t)
    unless t.is_a?(ULTerm)
      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
end
```

```ruby
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

  # 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?
      print "Reducing an argument\n"
      r = @t2.reduce
      if r
        ULApp.new(@t1,r)
      else
        nil
      end
    else
      print "Reducing a function\n"
      r = @t1.reduce
      if r
        ULApp.new(r,@t2)
      else
        nil
      end
```

```
    end
  end

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

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 ULVar(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
↪  ULVar.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`.

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 for elements of `STTerm`.

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

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 *type eraser* method 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.

(You should the definition of `ULTerm` and its methods into your file, or import it in a way compatible with the testing environments.)

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

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 the `STTerm` representation.

Part 5 – Bonus: pairs [10 bonus marks]

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

:TODO: