A Well-Typed Interpreter in Agda

1 Introduction and Methodology

The goal of this mini-project is to create a well-typed interpreter for the simply-typed lambda calculus (STLC) in the Agda theorem proving language. Literate Agda[1] has been used to write this report, and thus it includes both the complete source code inline and my reasoning/commentary on the code.

A key focus of this project was to try and make the types for expressions reflect closely their corresponding type rules, and as such Agda's support for mixfix and unicode has been heavily used. This is partially to reflect the logic-oriented parts of the program, and partially to permit a more fluent abstract syntax (almost like concrete syntax) which significantly improves the readability of the test examples. Hopefully, the reader agrees that this has been the right decision and does not experience difficulties in reading the program.

2 Set-up

I have chosen to rely on some parts of the Agda standard library (I needed some additional things than were provided in the lecture-provided Prelude). The first thing that is necessary is to import the relevant parts.

```
open import Data.Char
open import Data.Bool
open import Data.Nat
open import Data.Product
open import Data.Sum
open import Relation.Binary.PropositionalEquality as PropEq
open import Relation.Nullary.Core
open import Data.Nat.Show
import Data.String as Str
import Data.Unit as U
import Data.List as List
```

I will also do some fixity declarations for some operators I am going to need later.

```
infix 3 _:::_,_
infix 2 _∈_
infix 1 _⊢_
```

3 Types for Expressions

First and foremost it is necessary to define what the valid types are in the STLC. The types of the expression language are encoded as a set of *codes*, which are represented as constructors of single a data-type, *universe*, such that each constructor is analogous to an actual type in Agda.

In a similar fashion to Agda, I have chosen to call the type of types for `‘Set` (notice the backtick ‘ character which is used to distinguish from the actual type of types `Set` in agda).

```
data ‘Set : Set where
```

The first type that is supported for expressions in this implementation is the type of Boolean values.

```
  ‘Bool : ‘Set
```

Furthermore, natural numbers are supported.

```
  ‘Nat  : ‘Set
```

The trivial type of unit, is also supported.

```
  ‘Unit : ‘Set
```

A more interesting type that is supported, is the type of functions; which depends on two other codes for types. The resulting type is the type of functions from the first input type to the second input type. Notice that this constructor utilizes the mixfix functionality of Agda, and therefore a function from a type `t` to a type `s` can be encoded as `‘ t ⟹ s`.

```
  ‘_⟹_  : ‘Set → ‘Set → ‘Set
```

Similar to the encoding of the type of functions is the product type, which represents the type of paired values.

```
  ‘_×_  : ‘Set → ‘Set → ‘Set
```

Finally, the sum type is supported, which represents that either a value could be either one of the input types given. In other words, given a type `‘ t + s` then the type represents a value of type `t` or a value of type `s`.

```
  ‘_+_  : ‘Set → ‘Set → ‘Set
```

Since at some point the interpreter needs to interpret STLC expressions to Agda values[2], a *decoding* function is necessary to translate the codes for types of the expression language to actual Agda types. The decoding function is implemented in a straight-forward fashion where for each data type it translates to the analogous Agda type, and calls itself recursively when necessary.

An interesting feature of the decoding function is that it produces a type in Agda that is dependent on the code given. As such, while the codes for data types could have been written in an ordinary functional language; it is only possible to write the decoding function and use it in a dependently typed language. This is therefore one of the key features of dependently typed languages that allows both efficient **and** well-typed interpretation of the STLC language.

```
#_ : ‘Set → Set
# ‘Nat = ℕ
# ‘Bool = Bool
# (‘ t ⟹ s) = # t → # s
# ‘Unit = U.Unit
# (‘ t × s) = # t × # s
# (‘ t + s) = # t ⊎ # s
```

4 The Typing Environment

In an STLC setting, it is relatively easy to check the type of constant expressions. However, how is it possible to ensure that expressions containing variables are well-typed? The common answer is to use a *typing evironment*. The typing environment is the set of all variables introduced in the context of an expression using binders such as lambda functions or let-expressions.

A common way to define the typing environment, Γ, is that either it is the empty typing environment, $\cdot$, or it is an extension of an existing typing environment with some variable `x` of type `T` *i.e.* `x : T , Γ`. This way of representing the typing environment is advantageous because it preserves the order of bindings and it allows *overshadowing*, which is binding of a new variable using the same name (making the previous name-sharing variable inaccessible in the local context).

```
data Γ : Set where
  ·         : Γ
  _:::_,_   : Char → ‘Set → Γ → Γ
```

Now that there is a way to represent bound variables in the context, there is a need for a way of checking the type of a particular variable in an expression. The typical way of representing the type rule for checking variable types is the following (where means `e` is of type `T` under the typing environment Δ):

$$\textsc{Lookup}\ \frac{x : T \in \Delta}{\Delta \vdash x : T}$$

The typing rule says that `x` is of type `T` under the typing environment Δ ($\Delta \vdash$ `e : T`), if we can lookup `x` in Δ and get `T`.

Unfortunately, it seems hard to implement this rule directly in a straightforward way which Agda accepts. Instead, it is necessary to manually tell Agda where a particular variable exists in the environment, and this can be done using two rules: `Here` and `There` [3].

$$\textsc{Here}\ \frac{}{x : T \in x : T, \Delta} \qquad \textsc{There}\ \frac{x : T \in \Delta \qquad y \neq x}{x : T \in y : S, \Delta}$$

The `Here` rule says that we can conclude that the variable `x` has type `T` when it is the top variable in the typing environment. The `There` rule says that if we have found the variable in the typing environment, we can add a different variable on the top and it will still be a part of the environment; this allows us to do a search for the related variable when proving, skipping unrelated variables. These two rules have been encoded in Agda as follows:

```
data _∈_ :  Char → Γ → Set where
  H  : ∀ {x x′ Δ t } → {{eq : (x == x′) ≡ true}} → x ∈ x′ ::: t , Δ
  TH : ∀ {x y Δ s } → {{eq : (x == y) ≡ false}} → x ∈ Δ → x ∈ y ::: s , Δ
```

Note that the rules are further complicated since `Char` is used for variable names, and it does not have a native definitional equality in Agda. The double braces around the equality however should enable writing the proofs in a simple way without thinking about the `Char` equality and Agda should will figure out the right instance automatically.

To retrieve the actual type of a variable from the proven location, a simple lookup function is created. The lookup function uses the input location similar to indexing an array (essentially `Here-There` is just a fancy way of writing a natural number, with some additional properties at compile time). Notice that Agda automatically finds out that it is impossible to look up something in the empty typing environment, so that case is written using the absurd pattern.

```
!Γ_[_] : ∀ {x} → (Δ : Γ) → x ∈ Δ → ‘Set
!Γ_[_] · ()
!Γ _ ::: t , Δ [ H ]     = t
!Γ _ ::: _ , Δ [ TH i ]  = !Γ Δ [ i ]
```

5 Expressions

Finally, it is now time to talk about the various kinds of expressions in this version of STLC. The first oddity the user might have noticed is that I have used an infix $\vdash$ operator to represent expressions, making them seem like type judgements. While this completely irrational at first it will soon be apparent that this has a nice side effect, namely that if we write `e : Δ ⊢ T` in Agda it looks very similar to the type rule notation $\Delta \vdash$ `e : T`[1]. This also makes the

[1]The idea of encoding the environment in the expression instead of in the interpreter as done by the Augstussun paper[2], is probably inspired by a course I had taken earlier on Idris[4]. However, it will become apparent that much of how the expressions are represented is modelled in a completely different way (that I believe better models the type corresponding type rules).

constructor rules reflect the corresponding type rules and in many ways easier to understand.

```
data _⊢_ : Γ → ‘Set → Set where
```

The first types of expressions in this implementation of the STLC are rules for introducing primitive values, namely Boolean, Unit and natural numbers. To simplify the definition of natural numbers (and allow the use of number literals), it is simply allowed to use a natural number from Agda.

```
‘false          : ∀ {Δ} → Δ ⊢ ‘Bool
‘true           : ∀ {Δ} → Δ ⊢ ‘Bool
‘tt             : ∀ {Δ} → Δ ⊢ ‘Unit
‘n_             : ∀ {Δ} → ℕ → Δ ⊢ ‘Nat
```

The next type of expression is more interesting, as it concerns access to variables. To ensure that only bound variables are accessed and the expressions are well-typed, the programmer must in addition to the variable name provide a proof that the variable exists in the typing environment (the proof is in `[ ]`).

While this might seem tedious, a pattern I usually use to access a variable, `x`, is `" ‘x’ [ ? ]` where `?` represents a hole to proof later. When I am done writing the expression I need, I can proof the holes easily using the interactive editing of Agda. I believe that this should be possible to infer automatically at compile time using a method, but I am currently unaware of how to instruct Agda to do that.

One thing to notice is that the membership proof when seen as a natural number, actually reflects a De Bruijn index. While it then could be argued that De Bruijn indices were all that is necessary, I find them tedious to use in practice and the combination of variable naming and interactive Agda editing is much more compelling. Furthermore the expressions are less fragile when some reordering of bindings happen.

The resulting type of a variable is simply looked up in the typing environment.

```
‘‘_[_]  : ∀ {Δ} → (x : Char) → (i : x ∈ Δ) → Δ ⊢ !Γ Δ [ i ]
```

The rules for function application and lambda-expressions are obviously also included in this version of STLC. As mentioned at the start of this section, constructors of the expressions in this language reflect type rules very well. Every parameter for a constructor can be seen as a premise of the corresponding type rule, and the result can be seen as the conclusion. For example, the type rules for application and lambda-expressions are:

$$\textsc{App}\ \frac{\Delta \vdash e_f : T \to S \qquad \Delta \vdash e_x : T}{\Delta \vdash e_f\, e_x : S} \qquad \textsc{Lambda}\ \frac{x : T_x, \Delta \vdash e : T_r}{\Delta \vdash \lambda\,(x : T_x) \to e : T_r}$$

and the corresponding implementations are:

```
‘_⁻_              : ∀ {Δ t s} → Δ ⊢ ‘ t ⟹ s → Δ ⊢ t → Δ ⊢ s
‘λ_‘:_⟹_          : ∀ {Δ tr} → (x : Char) → (tx : ‘Set)
                    → x ::: tx , Δ ⊢ tr → Δ ⊢ ‘ tx ⟹ tr
```

Notice how even the extension of the environment seems to match in a close fashion.

I have also included some various interesting binary and unary operators in this languages such as addition, multiplication, logical and, logical or, less than and not. The corresponding constructors can be seen below.

```
‘_+_              : ∀ {Δ} → Δ ⊢ ‘Nat → Δ ⊢ ‘Nat → Δ ⊢ ‘Nat
‘_*_              : ∀ {Δ} → Δ ⊢ ‘Nat →  Δ ⊢ ‘Nat → Δ ⊢ ‘Nat
‘_∧_              : ∀ {Δ} → Δ ⊢ ‘Bool → Δ ⊢ ‘Bool → Δ ⊢ ‘Bool
‘_∨_              : ∀ {Δ} → Δ ⊢ ‘Bool →  Δ ⊢ ‘Bool → Δ ⊢ ‘Bool
‘_≤_              : ∀ {Δ} → Δ ⊢ ‘Nat → Δ ⊢ ‘Nat →  Δ ⊢ ‘Bool
‘¬_               : ∀ {Δ} → Δ ⊢ ‘Bool →  Δ ⊢ ‘Bool
```

To constructor a pair type, the `,` has been used as an operator *e.g.* `a , b` is the pair consisting of `a` and `b`. To retrieve the individual components `fst` and `snd` are available as operators.

```
‘_,_              : ∀ {Δ t s} → Δ ⊢ t →  Δ ⊢ s →  Δ ⊢ ‘ t × s
‘fst              : ∀ {Δ t s} → Δ ⊢ ‘ t × s → Δ ⊢ t
‘snd              : ∀ {Δ t s} → Δ ⊢ ‘ t × s → Δ ⊢ s
```

Similarly, `left` and `right` represent the two injections of the sum type. To actually do computation using the sum type, it is necessary to do a case analysis depending on which of the injections is received. In this version of the STLC, a `case` operator (similar to the one Haskell) is available for doing case analysis on values of the sum type.

```
‘left             : ∀ {Δ t s} → Δ ⊢ t → Δ ⊢ ‘ t + s
‘right            : ∀ {Δ t s} → Δ ⊢ s → Δ ⊢ ‘ t + s
‘case_‘of_||_     : ∀ {Δ t s u} → Δ ⊢ ‘ t + s
                      → Δ ⊢ ‘ t ⟹ u → Δ ⊢ ‘ s ⟹ u → Δ ⊢ u
```

Finally, let-bindings and if-then-else are included to make it easier to do some more advanced scripts. A key thing to notice is that the type is not explicitly given in the let-binding, as it only depends on the type of the expression (similar to type inference).

```
‘let_‘=_‘in_      : ∀ {Δ th tb} → (x : Char)
                    → Δ ⊢ th → x ::: th , Δ ⊢ tb → Δ ⊢ tb
‘if_‘then_‘else_ : ∀ {Δ t} → Δ ⊢ ‘Bool → Δ ⊢ t → Δ ⊢ t → Δ ⊢ t
```

6 The Runtime Environment

While the typing environment provides a way to check if a variable is well-typed, it does not provide a value for that variable at run-time. In order to store the values that the various variables are bound to at runtime, it is necessary to create a new *runtime environment*. The runtime environment is based on the typing environment, which ensures that both the length of the environment is correct (*i.e.* the number of variables matches exactly those that are bound), and ensures that the correct types of variables are stored.

```
data ⟨_⟩ : Γ → Set₁ where
  []   : ⟨ · ⟩
  _::_  : ∀ {x t Δ} → # t → ⟨ Δ ⟩ → ⟨ x ::: t , Δ ⟩
```

Similarly to the typing environment, a lookup function for retrieving values of variables is needed.

```
!_[_] : ∀ {x Δ} → ⟨ Δ ⟩ → (i : x ∈ Δ) → # !Γ Δ [ i ]
!_[_] [] ()
! val :: env [ H ]     = val
! val :: env [ TH i ]  = ! env [ i ]
```

7 The Interpreter

The last and probably most important part of the program, is the actual interpreter itself. Because the expressions are constructed in a well-typed fashion, the interpreter is seemingly simple (almost like in a dynamic language!). The first part of the interpreter accepts only an expression that is closed under its own environment, since the expression wouldn't be well-typed if it refers to free variables. The top-level interpretation function then just delegates to a recursive implementation of the interpreter and initializes it with the empty runtime environment.

```
interpret : ∀ {t} → · ⊢ t → # t
interpret = interpret' []
```

The recursive interpretation function uses the runtime environment to store the values bound when recursively interpreting an expression. Additionally, the typing environment may vary depending on how variables are bound in the recursive call chain.

```
  where interpret' : ∀ {Δ t} → ⟨ Δ ⟩ → Δ ⊢ t → # t
```

The interpretation of constant values is simple, and maps directly to the Agda equivalents.

```
        interpret' env `true = true
        interpret' env `false = false
        interpret' env `tt = U.unit
        interpret' env (`n n) = n
```

Similarly, when interpreting a variable its value is simply looked up in the runtime environment.

```
interpret' env `` x [ idx ] = ! env [ idx ]
```

Function application is done by recursively interpreting both the function and argument, and then simply performing the application in Agda.

```
interpret' env (` f _ x) = (interpret' env f) (interpret' env x)
```

A more interesting part of the interpreter, is the interpretation of lambda expressions. To do that, a lambda expression in Agda is made where the argument has the decoded type of the lambda argument in STLC. Additionally, the value environment must be extended with the value of the lambda argument such that when the argument gets value it is available for variables to access.

```
interpret' env (`λ _ `: tx ⟹ body)
   = λ (x : # tx) → interpret' (x :: env) body
```

For binary and unary operators, the arguments are just interpreted recursively and the corresponding operator in Agda is applied. An exception here is the less-than operator, because Agda does not have a corresponding boolean version; instead, the decidable version is used and translated to the corresponding boolean value.

```
interpret' env (` l + r) =
   interpret' env l + interpret' env r
interpret' env (` l * r) =
   interpret' env l * interpret' env r
interpret' env (` l ∧ r) =
   interpret' env l ∧ interpret' env r
interpret' env (` l ∨ r) =
   interpret' env l ∨ interpret' env r
interpret' env (` l ≤ r) with interpret' env l ≤? interpret' env r
interpret' env (` l ≤ r) | yes p = true
interpret' env (` l ≤ r) | no ¬p = false
interpret' env (`¬ x) = not (interpret' env x)
```

The interpretation for product types is similarly simple, and the with-pattern is only used to retrieve the relevant components (first component for `fst`, and second component for `snd`).

```
interpret' env (` f , s) = interpret' env f ,′ interpret' env s
interpret' env (`fst p) with interpret' env p
interpret' env (`fst p) | f , s = f
interpret' env (`snd p) with interpret' env p
interpret' env (`snd p) | f , s = s
```

The injections of a sum type are mapped directly to the Agda equivalents (with the inner expressions, interpreted recursively). Interestingly, the `case`-pattern starts by interpreting the value and chooses what branch to interpret dependening on what the result of that value is (*i.e.* the branches are lazily interpreted and don't use unnecessary time if left unused).

```
interpret' env (‘left v) = inj₁ (interpret' env v)
interpret' env (‘right v) = inj₂ (interpret' env v)
interpret' env (‘case s ‘of le || re) with interpret' env s
interpret' env (‘case s ‘of le || re) | inj₁ l = (interpret' env le) l
interpret' env (‘case s ‘of le || re) | inj₂ r = (interpret' env re) r
```

The `let`-bindings are also interpreted in a straight-forward fashion, by interpreting the variable value and binding it to variable in Agda. Then the bound result is added to the runtime environment of the `let`-body.

```
interpret' env (‘let _ ‘= h ‘in b) = let hval = interpret' env h
                                     in interpret' (hval :: env) b
```

Finally the `if`-expressions are also interpreted lazily (similar to `case`) depending on the value of the condition.

```
interpret' env (‘if b ‘then et ‘else ef) with interpret' env b
interpret' env (‘if b ‘then et ‘else ef) | true = interpret' env et
interpret' env (‘if b ‘then et ‘else ef) | false = interpret' env ef
```

8 Final Remarks

I have presented a well-typed interpreter for the simply-typed lambda calculus which seem to reflect the corresponding type rules very well. Even the expression language's abstract syntax reflects a possible concrete syntax very closely. After encoding the type of expressions correctly, the interpreter was easier to create (and more correct) than in a dynamicly typed language; particularly because of the good tooling support in Agda.

The code was written using a literate form for Agda, which while is convenient, provides less flexibility in the order the code can be presented. Finally, it seems that the tool support for the literate part of Agda is rather underwhelming.

9 Tests

```
testSimpleLambda : · ⊢ ‘Nat
testSimpleLambda = ‘ (‘λ ’x’ ‘: ‘Nat ⟹ ‘ ‘‘ ’x’ [ H ] + ‘‘ ’x’ [ H ]) _ ‘n 10

testNestedLambda : · ⊢ ‘Nat
testNestedLambda = ‘ ‘ (‘λ ’x’ ‘: ‘Nat ⟹
                        (‘λ ’y’ ‘: ‘Nat ⟹
```

```
                      ‘ ‘‘ ’x’ [ TH H ] * ‘‘ ’y’ [ H ])) _ ‘n 10 _ ‘n 15

-- Should not work because the inner x is not bound to a boolean,
-- and it should not be possible to refer to the outside x using Elem
--testNamingNotWorking : · ⊢ ‘Bool
--testNamingNotWorking = ‘ ‘ ‘λ ’x’ ‘: ‘Bool ⟹
--                       (‘λ ’x’ ‘: ‘Unit ⟹ ‘‘ ’x’ [ {!!} ]) _ ‘true _ ‘tt

testNamingWorking : · ⊢ ‘Unit
testNamingWorking = ‘ ‘ ‘λ ’x’ ‘: ‘Bool ⟹
                    (‘λ ’x’ ‘: ‘Unit ⟹ ‘‘ ’x’ [ H ]) _ ‘true _ ‘tt

testSum1 : · ⊢ ‘Nat
testSum1 = ‘let ’n’ ‘= ‘case ‘left (‘n 10) ‘of
                              ‘λ ’n’ ‘: ‘Nat ⟹ ‘‘ ’n’ [ H ]
                            || ‘λ ’b’ ‘: ‘Bool ⟹ ‘if ‘‘ ’b’ [ H ]
                                                  ‘then ‘n 1
                                                  ‘else ‘n 0
           ‘in ‘‘ ’n’ [ H ]

testSum2 : · ⊢ ‘Nat
testSum2 = ‘let ’n’ ‘= ‘case ‘right ‘true ‘of
                              ‘λ ’n’ ‘: ‘Nat ⟹ ‘‘ ’n’ [ H ]
                            || ‘λ ’b’ ‘: ‘Bool ⟹ ‘if ‘‘ ’b’ [ H ]
                                                  ‘then ‘n 1
                                                  ‘else ‘n 0
           ‘in ‘‘ ’n’ [ H ]

testProduct1 : · ⊢ ‘Bool
testProduct1 = ‘fst (‘ ‘true , (‘ ‘n 10 , ‘tt ))

testProduct2 : · ⊢ ‘ ‘Nat × ‘Unit
testProduct2 = ‘snd (‘ ‘true , (‘ ‘n 10 , ‘tt ))

testDeMorganFullOr : · ⊢ ‘Bool
testDeMorganFullOr = ‘let ’s’ ‘= ‘λ ’x’ ‘: ‘Bool ⟹ ‘
                                  λ ’y’ ‘: ‘Bool ⟹
                                    ‘¬ (‘ ‘‘ ’x’ [ TH H ] ∨ ‘‘ ’y’ [ H ])
                     ‘in ‘ ‘ ‘‘ ’s’ [ H ] _ ‘true _ ‘true
testDeMorganBrokenAnd : · ⊢ ‘Bool
testDeMorganBrokenAnd = ‘let ’s’ ‘= ‘λ ’x’ ‘: ‘Bool ⟹
                                    ‘λ ’y’ ‘: ‘Bool ⟹
                                    ‘ ‘¬ ‘‘ ’x’ [ TH H ] ∧ ‘¬ ‘‘ ’y’ [ H ]
                       ‘in ‘ ‘ ‘‘ ’s’ [ H ] _ ‘true _ ‘true
```

References

[1] Chalmers University of Technology: Literate agda. http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.LiterateAgda Accessed 2013-12-19.

[2] Augustsson, L., Carlsson, M.: An exercise in dependent types: A well-typed interpreter. In: In Workshop on Dependent Types in Programming, Gothenburg. (1999)

[3] Pouillard, N.: Simply-typed Lambda Calculus. http://www.itu.dk/courses/SPLG/E2013/lecture2/lecture2.html Accessed 2013-12-19.

[4] Brady, E.: Idris Course. http://www.idris-lang.org/dependently-typed-functional-programming-with-idris-course-videos-and-slides/ Accessed 2013-12-19.