Coursework

Due: 8pm, 6 May 2022

Instructions Complete the given assignments in the file `Coursework.hs`, and submit this on Moodle by Friday 6 May 8pm. Make sure your file does not have **syntax errors** or **type errors**; where necessary, comment out partial solutions. Use the provided function names. You may use auxiliary functions, and you are encouraged to add your own examples and tests.

Assessment and feedback Your work will be judged primarily on the correctness of your solutions. Incorrect or partial solutions may be given partial marks if they operate correctly on certain inputs. Marking is part-automated, part-manual. Feedback will be given through an overall document, published on Moodle, and by making solutions available.

Plagiarism warning The assessed part of this coursework is an **individual assignment**. Collaboration is not permitted: you should not discuss your work with others, and the work you submit should be your own. Disclosing your solutions to others, and using other people's solutions in your work, both constitute plagiarism: see http://www.bath.ac.uk/quality/documents/QA53.pdf.

Strong Normalization (100 marks)

An important theorem fo ngly normalizing: every reduction path will e work we will consider a proof method by R. O. Gandy. The idea is as follo upper bound to the length of all possible reductions of a typed term bound always reduces when a reduction step is applied. Here, w part, calculating the upper bound. We won't do a formal proof that the bound reduces, but we will observe that it does in examples.

Simply-typed terms

First, we will extend the λ-calculus of the tutorials with simple types. Types are given by the following grammar, where o is a **base type** and $\sigma \to \tau$ an **arrow type**.

$$\rho, \sigma, \tau \ ::= \ o \ | \ \sigma \to \tau$$

The terms of the **simply-typed λ-calculus** are given by the following grammar.

$$M, N \ ::= \ x \ | \ \lambda x^{\tau}.M \ | \ M\,N$$

Assignment 1: (20 marks)

In your file you are given the implementation of the λ-calculus from the tutorials. We will first extend this to the simply-typed calculus.

a) Complete the datatype `Type` to represent simple types following the grammar above. For the base type, use the constuctor `Base`, and for the arrow type, use the (infix) constructor `:->`. Un-comment the function `nice`, the `Show` instance, and the examples `type1` and `type2` to see if everything type-checks.

b) Make the datatype `Type` a member of the type class `Eq` so that `(==)` gives equality of types.

c) Adapt the datatype `Term` for λ-terms from the tutorials to simply-typed terms, following the grammar above. Un-comment the function `pretty` and the `Show` instance to see if everything type-checks.

d) Un-comment the function `numeral` from the tutorials and adapt it to work with simply-typed terms, following the definition here:

$$N_n = \lambda f^{o \to o}.\lambda x^o.L_n \qquad L_0 = x \qquad L_{n+1} = f\, L_n$$

Un-comment also the other functions and examples for numerals, from `sucterm` to `example7`.

e) Un-comment the functions `used`, `rename`, `substitute`, and `beta` from the tutorials and adapt th

f) Un-comment

ch term. We will adapt this to display the upper bound to reductions la or now.


```
*Main> type2
(o -> o) -> o -> o
*Main> type2 == type1 :-> type1
True
*Main> type2 == type1
False
*Main> numeral 4
\f: o -> o . \x: o . f (f (f (f x)))
*Main> example1
(\m: (o -> o) -> o -> o . \f: o -> o . \x: o . m f (f x))
                                        (\f: o -> o . \x: o . x)
*Main> normalize it
  0 -- (\m: (o -> o) -> o -> o . \f: o -> o . \x: o . m f (f x))
                                        (\f: o -> o . \x: o . x)
  0 -- \a: o -> o . \b: o . (\f: o -> o . \x: o . x) a (a b)
  0 -- \a: o -> o . \b: o . (\b: o . b) (a b)
  0 -- \a: o -> o . \b: o . a b
```

Type checking

The types we have added so far are only an annotation, but really we want those terms that are **well-typed**: where the types of functions and arguments match in the expected way. To check if a given term is well-typed is a simple inductive algorithm that we will implement here. (Note that this is different from **type inference**, which is the more involved algorithm that decides whether an **untyped** term can be given a type.) The definitions are as follows.

A **context** Γ is a finite function from variables to types, written as a comma-separated list.

$$\Gamma \ ::= \ x_1 : \tau_1, \ldots, x_n : \tau_n$$

A term M has type τ in a context Γ, written $\Gamma \vdash M : \tau$, if that statement can be derived using the following type checking rules.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x^{\sigma}.M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash M : \sigma}{\Gamma \vdash M\,N : \tau}$$

The type checking rules then give an inductive algorithm. To find a type for M, the inputs are Γ and M, and the algorithm either outputs a type τ if M is well-typed, or fails if M is not well-typed. The conclusion of a rule is what is computed, and premises of a rule (the parts above the line) give the recursive calls.

Assignment 2: **(20 marks)**

a) Complete the type `Context` as a list of pairs of va ng the `()` with the correct definition.

b) Implement the type-checking algorithm described abo `typeof` . If the term is well-typed, return its `Type` ; if it is not, you may throw an exception (any will do; it doesn't need to match the examples). Un-comment `example8` for testing.

```
*Main> typeof [] (numeral 3)
(o -> o) -> o -> o
*Main> typeof [] example7
(o -> o) -> o -> o
*Main> example8
\x: o . f x x
*Main> typeof [("f",Base :-> (Base :-> Base))] example8
o -> o
*Main> typeof [] example8
o -> *** Exception: Variable f not found
*Main> typeof [("f",Base)] example8
o -> *** Exception: Expecting ARROW type, but given BASE type
*Main> typeof [("f",(Base :-> Base) :-> Base)] example8
o -> *** Exception: Expecting type o -> o, but given type o
```

Higher-order numeric functions

Now, we will start on the constructions we need for counting reduction steps. This document will explain the ideas behind them, but note that it is not necessary to fully understand everything: you can build each function by following the specification closely.

We want to build a function from simply-typed terms to natural numbers, such that when a term reduces, the number gets smaller. It then follows that all reduction paths must eventually end, and that the term is strongly normalizing.

The problematic case is an application: suppose that for a term $M\,N$ we know that M reduces in at most m steps, and N reduces in at most n steps. But that does not help us to give the number of steps for $M\,N$. For example, if M and N are normal (at most zero reduction steps), then $M\,N$ might not be normal. The solution is to give for M not just a number, but a **function** that, given a bound for N, produces a bound for $M\,N$.

The types will make sure that everything works out. We will build a function that interprets terms as **functionals**, higher-order functions over numbers, as follows:

A term ...	becomes a ...
$M : o$	
$M : o \to o$	
$M : o \to o \to o$	
$M : (o \to o) \to o$	function $f : (\mathbb{N} \to \mathbb{N}) \to$
etc.	etc.

This gives us a higher-order function over numbers, but not yet a number. To get that, we evaluate the functional with **dummy** arguments: zero for $\mathbb{N}$, the function $g(x) = 0$ for $\mathbb{N} \to \mathbb{N}$, etc:

A term ...	becomes a ...	and gives a number ...
$M : o$	number $n \in \mathbb{N}$	n
$M : o \to o$	function $f : \mathbb{N} \to \mathbb{N}$	$f\,0$
$M : o \to o \to o$	function $f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$	$f\,0\,0$
$M : (o \to o) \to o$	function $f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$	$f\,g$ where $g(x) = 0$
etc.	etc.	etc.

We will now start making these ideas precise, using the following definitions.

- The sets of **functionals** we need are given by the following grammar.

$$\mathbb{F} \ ::= \ \mathbb{N} \mid \mathbb{F} \to \mathbb{F}$$

The function $|\tau|$ takes every type τ to a set of functionals, defined by:

$$
\begin{aligned}
|o| &= \mathbb{N} \\
|\sigma \to \tau| &= |\sigma| \to |\tau|
\end{aligned}
$$

Note that since a type is of the form $\tau = \tau_1 \to \ldots \to \tau_n \to o$, a set of functionals that type is of the following form.

$$|\tau_1| \to \cdots \to |\tau_n| \to |o| = \mathbb{F}_1 \to \cdots \to \mathbb{F}_n \to \mathbb{N}$$

- For every type τ, the **dummy** element $\perp_\tau \in |\tau|$ is defined by:

$$
\begin{aligned}
\perp_o &= 0 \in \mathbb{N} \\
\perp_{\sigma\to\tau} &= f \in |\sigma \to \tau| \quad \text{where } f(x) = \perp_\tau
\end{aligned}
$$

That is, the functional f above takes one argument $x \in |\sigma|$, throws it away, and returns $\perp_\tau$. Informally, if $\tau = \tau_1 \to \ldots \to \tau_n \to o$ then the dummy element

$$\perp_\tau \in |\tau_1| \to \cdots \to |\tau_n| \to \mathbb{N}$$

takes n argum

- For a functional $\mathbb{N}$ returns a number by providing the necessary dummy arguments, define

$$
\begin{aligned}
\lfloor n \rfloor_o &= \\
\lfloor f \rfloor_{\sigma\to\tau} &= \lfloor f(\perp_\sigma) \rfloor_\tau
\end{aligned}
$$

Informally, if $\tau = \tau_1 \to \ldots \to \tau_n \to o$ then for $f \in |\tau|$ the counting operation gives the following (verify for yourself that this is indeed a number).

$$\lfloor f \rfloor_\tau = f\,(\perp_{\tau_1}) \ldots (\perp_{\tau_n})$$

- The **increment** function $f +_\tau n$ increments a functional $f \in |\tau|$ by a number n, defined by:

$$
\begin{aligned}
m +_o n &= m + n \\
f +_{\sigma\to\tau} n &= g \qquad \text{where } g(x) = f(x) +_\tau n
\end{aligned}
$$

Informally, for $\tau = \tau_1 \to \ldots \to \tau_n \to o$ and any functional in the set

$$|\tau_1| \to \cdots \to |\tau_n| \to \mathbb{N}$$

the increment function $+_\tau$ adds a number “to the last $\mathbb{N}$”.

Assignment 3: **(30 marks)**

We will now implement these definitions. You are given a data type `Functional`, with a constructor `Num` for $\mathbb{N}$ and a constructor `Fun` for $\mathbb{F} \to \mathbb{F}$. The data type comes with a `Show` instance, but since we cannot show functions, it will only properly display a functional if it is a number. Further, to apply a functional $\mathbb{F} \to \mathbb{F}$ as a function, the constructor `Fun` gets in the way. The function `fun` is included for this purpose: it takes `Fun f` and extracts `f`, which is a function of type `Functional -> Functional`.

a) Complete the following example functionals: `plussix` of type $\mathbb{N} \to \mathbb{N}$, which adds six to a given input; `plus` of type $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ that implements addition; and `twice` of type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$ which takes a functional f and an input n and applies f to n twice.

b) Complete the function `dummy` that returns the dummy element $\bot_\tau$ for each type τ.

c) Implement $\lfloor f \rfloor_\tau$ as the function `count`, which takes as input the type τ and a functional $f \in |\tau|$. (You do not need to check if f belongs to $|\tau|$, and you may throw an error if it doesn't.)

d) Implement $+_\tau$ … e τ as an argument; inste… tional.


```
*Main> fun plussix (Num 3)
Num 9
*Main> fun (fun plus (Num 3)) (Num 4)
Num 7
*Main> fun (fun twice plussix) (Num 1)
Num 13
*Main> dummy Base
Num 0
*Main> dummy type1
Fun ??
*Main> fun (dummy type1) (Num 4)
Num 0
*Main> count type1 plussix
6
*Main> count type2 twice
0
*Main> count type1 (fun twice plussix)
12
*Main> increment (dummy Base) 5
```

```
Num 5
*Main> fun (increment plussix 3) (Num 1)
Num 10
*Main> fun (fun (increment twice 4) plussix) (Num 1)
Num 17
*Main> count type1 (fun (increment twice 4) plussix)
16
```

Counting reduction steps

To give an upper bound to the number of reduction steps, we will define a function $\|M\|$ that takes a term $M : \tau$ to a function $f \in |\tau|$. This will be a straightforward induction on M. As with the type checking function, where we needed a context Γ to know the types of the free variables of M, here we need a **valuation** which assigns to each variable $x : \tau$ a functional $f \in |\tau|$. We write v for a valuation, and $v[x \mapsto f]$ for the valuation that maps x to f and any other variable y to $v(y)$.

The interpretation of M is then constructed as follows.

- If M is a variabl [illegible] turn $v(x) \in |\tau|$.
- For an abstraction $\lambda x\ .M : \sigma \to \tau$, we con [illegible] $f \in |\sigma| \to |\tau|$ as follows: for any $g \in |\sigma|$, we define $f(g)$ [illegible] $M : \tau$ for the valuation $v[x \mapsto g]$. Intuitively, if we consider [illegible] t an arbitrary term, then g represents the bound on reduction steps for x, and f uses g to compute the bound for M.

- For an application $M\,N : \tau$ where $N : \sigma$, if the interpretation of M is $f \in |\sigma| \to |\tau|$ and that of N is $g \in |\sigma|$, then the basis of our bound for $M\,N$ is $f(g)$. This measures the steps in M, given that N is an argument. We then need to adjust this in two ways:
  - Since $M\,N$ could be a redex (or could become one after reduction or substitution in M), we increment $f(g)$ by one, to $f(g) +_\tau 1$.
  - In the case that M discards N, for instance if $M = \lambda x.y$, also f will discard g. But we still need to count reduction steps in N, which we do separately: we increment our answer with the number $\lfloor g \rfloor_\sigma$, which gives the bound for N, to $f(g) +_\tau (1 + \lfloor g \rfloor_\sigma)$.

Note that in the case $M\,N$, we need to know the type of N to compute $\lfloor g \rfloor_\sigma$, and for that we need a context Γ with the type of its free variables. The **interpretation** of M is then defined with a context Γ and a valuation v, as

$$\|M\|_v^\Gamma$$

and is defined inductively as follows.

$$\begin{aligned}
\|x\|_v^\Gamma &= v(x) \\
\|\lambda x^{\tau}.M\|_v^\Gamma &= f && \text{where} \quad f(g) = \|M\|_{v[x\mapsto g]}^{\Gamma,\,x:\sigma} \\
\|M\,N\|_v^\Gamma &= f(g) +_{\tau} (1 + \lfloor g \rfloor_{\sigma}) && \text{where} \quad f = \|M\|_v^\Gamma\,, \qquad \Gamma \vdash M\,N : \tau \\
& && \qquad\quad\;\; g = \|N\|_v^\Gamma\,, \qquad \Gamma \vdash N : \sigma
\end{aligned}$$

Then the **bound** $\|M\|$ of a closed term $M : \tau$ (one without free variables) is given by $\lfloor f \rfloor_\tau$, where f is the interpretation of M with the empty context and empty valuation.

Assignment 4: **(30 marks)**

We will implement the **interpretation** and **bound** functions, and adapt `normalize` to show the bound for each term, so we can see that it indeed goes down during reduction.

a) Complete the type tionals, given as a list of pairs.

b) Complete the function `interpret` to give th $\|\,\|_v^\Gamma$ of a well-typed term M as a functional.

c) Complete the function `bound` that takes a closed, well-typed term $M : \tau$, computes its interpretation f, and returns $\lfloor f \rfloor_\tau$.

d) Adapt `normalize` to show the bound of the term at each step.

```
*Main> bound example1
5
*Main> bound example2
68
*Main> bound example3
24
*Main> bound example4
2060
*Main> bound example5
1880
*Main> bound example6
18557
```

```
*Main> bound example7
3867842
*Main> normalize example1
  5 -- (\m: (o -> o) -> o -> o . \f: o -> o . \x: o . m f (f x))
                                               (\f: o -> o . \x: o . x)
  4 -- \a: o -> o . \b: o . (\f: o -> o . \x: o . x) a (a b)
  3 -- \a: o -> o . \b: o . (\b: o . b) (a b)
  1 -- \a: o -> o . \b: o . a b
```

That concludes our implementation. Note that a normal form does not need to have a bound of zero, since we are counting applications, not redexes. But that is fine: as long as the bound always goes down for a reduction step, terms will be strongly normalizing. You can use the example terms to convince yourself that this is the case, and you can change the evaluation strategy used by `normalize` to see that this works for any reduction step, or use `randomIO` to choose arbitrary redexes.

To make this construction into a strong normalization proof, we would have to **prove** that the bound always goes down. This is a nice challenge—contact me if you would like to try it.