# Exercise 05

## Task 0: Russel's Paradox

*(This task is not relevant for the exam)*

Last week we learned about the lambda calculus. It has 3 expressions: variables, function abstraction, and function application.

$$e ::= x \mid \lambda x.e \mid e\ e$$

The values of the lambda calculus are function abstraction. Lambda calculus terms are reduced by using beta reduction $(\lambda x.e_0)\ e_1 = e_0[e_1/x]$.

Functions can be used to model other datatypes. Last week, we saw how booleans and natural numbers can be encoded. Today we will see another data type: *Sets*. Sets can be modelled as an unary function that returns true if it contains its argument and false otherwise. That means for lambda terms $e_0$ and $e_1$, we define:

$$e \in s = s\ e$$

1. Define boolean `not`, `and` and `or` in the lambda calculus using the definitions of Church booleans from last week. Based on that, define `union` and `intersection` on sets. You can use the provided template to implement these operations in our language `FE`.

**Answer**:

```
def union(s1 : FE, s2 : FE) : FE =
    Fun('e, or(App(s1, 'e), App(s2, 'e)))
def intersect(s1 : FE, s2 : FE) : FE =
    Fun('e, and(App(s1, 'e), App(s2, 'e)))
```

In 1901, Bertrand Russel discovered his famous Russel's Paradox. The paradox is as follows:

- Let R be the set of all sets that do not contain themself. Does R contain itself?
  - If R does not contain itself, then by definition it contains itself;
  - and if R does contain itself, then by definition it does not contain itself.

You can read more about Russel's paradox on Wikipedia: https://en.wikipedia.org/wiki/Russell's_paradox. The formal definition of Russel's set, i.e. the set that contains all sets that do not contain themselves, is

$$R = \{s \mid s \notin s\}$$

Russel's paradox is the question whether $R \in R$? We just learned how to define sets in the lambda calculus. This gives us the possiblity to use our interpreter and *compute* whether R is in R!

2. Define R as a function in the lambda calculus. Use the interpreter to check whether R is an element of R, i.e. interpret $R\ R$. What is the result?

**Answer**:

Russel's set in lambda calculus:

$$R = \lambda s.not(s\ s)$$

This can be directly transaferred to Scala:

```
def R : FE = Fun('s, NOT(App('s, 's)))
```

It is now possible to compute whether R is in R by interpreting

```
interp(App(R, R))
```

Running this program results in a StackOverflowError.

3. Check the outcome of the interpreter by manually applying beta reduction.

**Answer**:

$$R\ R = (\lambda s.not(s\ s))R =_\beta not(R\ R) = not(not(R\ R)) = ...$$

We see that there can be non-terminating terms in the lambda calculus.

## Task 1: Fixpoint Combinators

In the lambda calculus (and FLAE), we have expressions that do not terminate. For example, consider the following expression, called the Omega combinator:

$$\Omega = (\lambda x.x\ x)(\lambda x.x\ x)$$

1. What happens if we evaluate $\Omega$?

**Answer**: $\Omega$ is a non-terminating expression. It evaluates to itself.

$$\Omega = (\lambda x.xx)(\lambda x.xx) =_\beta (\lambda x.xx)[(\lambda x.xx)/x] = (\lambda x.xx)(\lambda x.xx) = \Omega$$

$\Omega$ is not very useful by itself, but we can modify it to implement recursive functions! First, consider this well-known defintion of the factorial function:

```
def fact(n : Int) = if (n == 0) 1 else n * fact(n - 1)
```

2. Write this function in lambda calculus or FLAE. What is the problem? Note: We saw how to encode natural numbers and booleans in the lambda calculus. Thus, you can just use them normally. You can use the provided template that defines all the needed operations in our language.

**Answer**: Let's try to just implement the function as is:

$$\lambda n.\textbf{if } (n == 0)\ 1 \textbf{ else } n * fact(n-1)$$

Problem: If we try to simply translate the function to lambda calculus, we do not know what `fact` is. We could solve this problem by binding `fact` as well.

$$\lambda fact.\lambda n.\textbf{if } (n == 0)\ 1 \textbf{ else } n * fact(n-1)$$

But now we have the problem that we have to bind *fact* recursively. How can we do that?

Let's consider some mathematic basics to implement recursion with the lambda calculus. From a mathematic point of view, in order to achieve the necessary "cyclicity", we need to compute a *fixpoint* of a function.

A fixpoint `x0` of a function `f` is defined as follows:

```
f(x0) = x0
```

A *fixpoint combinator* is a function `fix` that computes one fixpoint of a function value (there may be many fixpoints):

```
fix(f) = x0  where  f(x0) = x0
```

As a first step, let us assume that we have a fixpoint combinator "fix" given, ignoring how it would look like. The fixpoint combinator achieves recursion by re-applying `f` any number of times

```
fix(f) = x0
<=> fix(f) = f(x0)
<=> fix(f) = f(fix(f))
<=> fix(f) = f(f(fix(f)))
<=> ...
```

We can use `fix` to define recursive functions. To do this, we define a function that expects its own fixpoint as first argument. For example:

```
def myfun = (myfix => (n => myfix(n+1)))
```

If we apply our fixpoint combinator `fix` to `myfun`, we get a fixpoint `f` such that:

```
fix(myfun) = f  where  myfun(f) = f
```

Accordingly, by unfolding `myfun(f)` (beta reduction) we get the following equations:

```
 f == myfun(f) == (n => f(n+1))
```

Now we can inline `f` into `(n => f(n+1))` an arbitrary number of times:

```
fix(myfun) == f
== (n => f(n+1))
== (n => (n => f(n+1))(n+1)) == (n => f((n+1)+1)) == (n => f(n+2))
== (n => (n => f(n+1))(n+2)) == (n => f((n+1)+2)) == (n => f(n+3))
== (n => (n => f(n+1))(n+3)) == (n => f((n+1)+3)) == (n => f(n+4))
```

This shows that `f` is indeed a recursive function representing the body of `myfun`.

You can find more details on fixpoint combinators online, for example on Wikipedia: http://en.wikipedia.org/wiki/Fixed-point_combinator.

To understand better how this works, let us assume the following Scala definition of `fix` for now:

```
def fix[A,B](f: (A => B) => (A => B)): A => B =
      (x:A) => f(fix(f))(x)
```

3. Implement a function `factorial` that computes the factorial of a given number n using `fix`. As a first version, you can come up with an untyped term which is not necessarily accepted by the compiler. Then, try to come up with a version that the compiler accepts.

**Answer**:

```
val factorial : Int => Int = fix (
    (fact: (Int => Int))  =>
        ((n:Int) => if (n <= 1) 1 else n * fact(n-1))
)

val fact5 = factorial(5)
```

Note that the Scala `fix`-definition that we used in the previous subtask relied on Scala recursion - so we did not yet specify our "own" recursion. We will now study the mathematical definition of the fixpoint combinator in an *untyped* language (untyped lambda calculus):

$$Y = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

The *Y-combinator* is one implementation of a fixed point combinator.

4. Study how Y achieves recursion:

a) On paper, apply $Y$ to `myfun` (see above) from above and show that $Y$ indeed computes a fixpoint of `myfun`.
b) On paper, apply $Y$ to an untyped version of your factorial function from above. Compute `factorial(2)` using $Y$.
c) More generally (and on paper), show that $Y$ computes the fixpoint of any function $g$. Hint: Repeatedly apply beta reduction to $Y\ g$ for a function symbol $g$.

**Answer**: a). The Y-Combinator:

$Y = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$

myfun (from above) in lambda calculus:

$myfun = \lambda m.\lambda n.m(n+1)$

Apply $Y$ to myfun:

$Y\ myfun =_\beta (\lambda x.myfun\ (x\ x))(\lambda x.myfun\ (x\ x)) =: f$

$=_\beta myfun((\lambda x.myfun\ (x\ x))(\lambda x.myfun\ (x\ x))) = myfun\ f$

$=_\beta \underline{\lambda n.f(n+1)} = \lambda n.(myfun\ f)(n+1)$

$=_\beta \lambda n.(\lambda n.f(n+1))(n+1)$

$=_\beta \lambda n.f((n+1)+1) = \underline{\lambda n.f(n+2)} = ...$

myfun has no recursion anchor. The computation will never stop.

b).

$factorial = \lambda fact.\lambda n.\textbf{if}\ (n == 0)\ 1\ \textbf{else}\ n * fact(n-1)$

At first, let's evaluate $Y\ factorial$

$Y\ factorial =_\beta (\lambda x.factorial\ (x\ x))(\lambda x.factorial\ (x\ x)) =: \underline{f}$

$=_\beta factorial\ ((\lambda x.factorial\ (x\ x))\ (\lambda x.factorial\ (x\ x))) = factorial\ f$

$=_\beta \underline{\lambda n.\textbf{if}\ (n == 0)\ 1\ \textbf{else}\ n * f(n-1)}$

This gives us a useful representation of $Y\ factorial = f$. We can now evaluate $f(2)$:

$\underline{f(2)}$

$(\lambda n.\textbf{if}\ (n == 0)\ 1\ \textbf{else}\ n * f(n-1))\ 2$

$=_\beta \textbf{if}\ (2 == 0)\ 1\ \textbf{else}\ 2 * f(2-1)$

$= 2 * f(2-1) = \underline{2 * f(1)}$

$\underline{f(1)}$

$(\lambda n.\textbf{if}\ (n == 0)\ 1\ \textbf{else}\ n * f(n-1))\ 1$

$=_\beta \textbf{if}\ (1 == 0)\ 1\ \textbf{else}\ 1 * f(1-1)$

$= 1 * f(1-1) = \underline{1 * f(0)}$

$\underline{f(0)}$

$(\lambda n.\textbf{if}\ (n == 0)\ 1\ \textbf{else}\ n * f(n-1))\ 0$

$=_\beta \textbf{if}\ (0 == 0)\ 1\ \textbf{else}\ 0 * f(0-1)$

$= \underline{1}$

Inserting into $2 * f(1)$ yields: $2 * 1 * 0 = 2 = \mathit{factorial}\ 2$

c).

$Y\ g = (\lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x)))g$

$=_\beta (\lambda x.g\ (x\ x))(\lambda x.g\ (x\ x)) =: f$

$=_\beta g\ ((\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))) = g\ f$

We see that $f$ is a fixpoint of $g$, because $g\ f = f$. Thus, $\mathit{fix}\ g = f$ with $\mathit{fix} = Y$. $Y$ is indeed a fixpoint combinator.

Hence, we could implement recursion without relying on recursion being available in the interpreter language (= language-integrated recursion) by using fixpoint combinators defined via first-class functions.

5. Could we express $Y$ in Scala (or the typed lambda calculus)? Why or why not? Hint: To come up with an answer, first consider the simpler divergent combinator $\Omega = (\lambda x.xx)(\lambda x.xx)$.

**Answer**: If we consider $\Omega$ first, we notice that it is difficult to assign a type to `x` using a "standard" type system: Obviously, `x` has to have a function type (since it is applied to `x`), e.g. `A => B`. But then, if we apply `x` of type `A => B` to `x`, `x` has to be a function that also accepts a function, e.g. `A => B => C`. But then, if we apply `x` of type `A => B => C` to `x`, `x` has to be a function that also accepts a function of type `A => B => C`, e.g. `A => B => C => D`. And so on.... A similar issue arises when trying to type `x` in $Y$.

6. Collect arguments for and against language-integrated recursion in contrast to using fixpoint combinators defined via first-class functions.

**Answer**: Of course, many answers are possible. Pro using language-integrated recursion: - interpreter might be easier to understand - if recursion is implemented very efficiently in the interpreter language, we can benefit from that

Pro using fixpoint combinators defined via first-class functions: - can be used even if host-language does not provide recursion

Finally, let's implement a recursive function in FLAE.

7. Implement the Y-combinator and the factorial function in FLAE with call-by-name semantics. Compute the factorial of 5. Note: You can use arithmetic expressions and `If0` to implement the factorial function.

8. Why does the Y-combinator not work in a language with call-by-value semantics?

**Answer**: The problem is evaluation this step:

$g\ ((\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x)))$

In call-by-name semantics, we substitute $(\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x)) =: f$ into $g$ (beta-reduction) without evaluating $f$ first. Thus, we can continue evaluating the body of $g$ first.

In call-by-value semantics, we would evaluate $f$ first, before evaluating the body of $g$, which would produce

$$g\ ((\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))) = g\ f = g\ g\ f = g\ g\ g\ f = ...$$

In order to use fixpoint combinators in a call-by-value language, one could use the Z-combinator instead of the Y-combinator.