[１] Prove that any pure term has at most one normal form. That is, prove the following: if $e \rightarrow^*_{full} e_1$,$e \rightarrow^*_{full} e_2$, $\neg\exists e_1'.e_1 \rightarrow_{full} e_1'$, and $\neg\exists e_2'.e_2 \rightarrow_{full} e_2'$, then $e_1 = e_2$. (Hint: Use the confluence property of $\rightarrow_{full}$).

The confluence theorem says: If $e\rightarrow^*_{full}$ e_1 and $e\rightarrow^*_{full}$ e_2 then there is e' such that $e_1\rightarrow^*_{full}$ e' and $e_2\rightarrow^*_{full}$ e'.
I am going to base my answer in this theorem. Let e_β and e_β' be normal forms. I'll differentiate two cases:
<u>First case</u>
e is one step from the normal form. This would mean that:
$e\rightarrow_{full}e_\beta$ and/or $e\rightarrow_{full}e_\beta$'. Since one of the two (or both) is a normal form this would mean that there is no e' such that $e_\beta\rightarrow^*_{full}$ e' and $e_\beta'\rightarrow^*_{full}$ e' (since at least one of them cannot be reduced) which is the opposite of the confluence theorem.

<u>Second case</u>
e is more than one step from the normal form. This would mean that:
There is such an e that $e\rightarrow^*_{full}$ e_β and $e\rightarrow^*_{full}$ e_β'. Let this e be the closest one from the normal form. This e would be the point of divergence and since it is the closest one it would mean that there is no other point of confluence. $e\rightarrow^*_{full}$ e_1 and $e\rightarrow^*_{full}$ e_2 then there would be no such an e' that $e_1\rightarrow^*_{full}$ e' and $e_2\rightarrow^*_{full}$ e' which is the exact opposite of the confluence theorem.

［2］ Consider the following terms.
two λs.λz.s (s z)
four λs.λz.s (s (s (s z)))
times λm.λn.λs.λz.2n (λz′.m s z′) z
Show the steps of the evaluation times two two $\rightarrow^*_{full}$ four. (You do not need to show the derivation tree of each step.)

(λm.λn.λs.λz. n (λz′.m s z′) z)(λs.λz.s (s z))(λs.λz.s (s z)) =
(λm.λn.λs.λz. n (λz′.m s z′) z)(2)(2) =
λs.λz.2 (λz′.2 s z′) z =
λs.λz.(λs.λz.s (s z)) (λz′.2 s z′) z =
λs.λz.(λz′.2 s z′) (((λz′.2 s z′)) z) =
λs.λz.2 s ((λz′.2 s z′) z) =
λs.λz.2 s (2 s z) =
λs.λz.2 s ((λs.λz.s (s z)) s z) =
λs.λz.2 s (s (s z)) =
λs.λz.(λs.λz.s (s z)) s (s (s z)) =
λs.λz.s (s (s (s z))) =

［3］Prove that call-by-value evaluation of pure calculus is deterministic. That is, prove the following:
if $e \to_{val} e_1$ and $e \to_{val} e_2$, then $e_1 = e_2$. (Hint: Prove by induction on the structure of e).

As we have seen $\to_{full}$ is non-deterministic. However, $\to_{full}$ satisfy a certain "determinism-like" property called confluence.
This would mean that if $e \to_{full}^{*} e_1$ and $e \to_{full}^{*} e_2$ then there is e' such that $e_1 \to_{full}^{*} e'$ and $e_2 \to_{full}^{*} e'$.
From this property we can understand that the non-deterministic property of $\to_{full}$ comes from the order in which we reduce the terms however it will not affect the final result.
However, call-by-value can be understood as a restriction of full β evaluation. There is no reduction under λ, and it reduces the function side of application first and restricts application to happen only after the argument is evaluated to values.
This would mean that call-by-value restricts the order of the evaluation. Knowing that any case in $\to_{full}$ that could cause an non-deterministic behaviour comes from the evaluation order, we conclude that any non-deterministic cases in $\to_{full}$ will be deterministic in $\to_{value}$.

Fernando Vallecillos Ruiz 1W18CG03

[4] Show the derivation of the typing judgement $g : int \to int \vdash (\lambda \mathrm{f}. \lambda \mathrm{y}. \mathrm{f}\,(\mathrm{y} + \mathrm{y}))\; g : int \to int$

$$\dfrac{\dfrac{\dfrac{\dfrac{g: int \to int, f: int \to int, y: int \vdash \mathrm{f}: \mathrm{int} \to \mathrm{int} \qquad \dfrac{g: int \to int, f: int \to int, y: int \vdash \mathrm{y}: \mathrm{int} \qquad g: int \to int, f: int \to int, y: int \vdash \mathrm{y}: \mathrm{int}}{g: int \to int, f: int \to int, y: int \vdash \mathrm{y} + \mathrm{y}: \mathrm{int}}}{g: int \to int, f: int \to int, y: int \vdash \mathrm{f}(\mathrm{y} + \mathrm{y}): \mathrm{int}}}{g: int \to int, f: int \to int \vdash \lambda \mathrm{y}. \mathrm{f}(\mathrm{y} + \mathrm{y}): \mathrm{int} \to \mathrm{int}}}{g: int \to int \vdash \lambda \mathrm{f}. \lambda \mathrm{y}. \mathrm{f}(\mathrm{y} + \mathrm{y}): (\mathrm{int} \to \mathrm{int}) \to int \to int} \qquad g: int \to int \vdash \mathrm{int} \to \mathrm{int}}{g : int \to int \vdash (\lambda \mathrm{f}. \lambda \mathrm{y}. \mathrm{f}\,(\mathrm{y} + \mathrm{y}))\; g: int \to int}$$

［5］While type soundness gurantees that the evaluation a closed typable term does not get stuck, it turns out that the converse does not always hold. That is, it is not the case that all closed terms whose evaluation do not get stuck are typable (in fact, there is no recursive set of rules that can type exactly such terms). Give an example of a closed extended calculus term e whose evaluation does not get stuck but e is not typable.

First let's define some terms:
We call a λ term e closed if fv(e) = ∅.
We say that a closed term e is typable if there exists τ such as ⊢ e: τ.
Non-reducible non-value terms are called stuck.

A quick example of a not typable term that does not get stuck is:

Y = (λzx. x(z z x))λzx. x(z z x)

This expression is involved in infinite derivation, but it is not typable.

［6］Look up the terminologies “static type system" and “dynamic type system" on your favorite Internet search engine. Describe what the terminologies mean, and also discuss the pros and cons of the two approaches.

Type system: set of rules that assigns a property called type to constructs of a computer program. These types formalize and enforce implicit categories the programmer uses for algebraic data, data structures or other components. Often specifies as part of programming languages.

Static type system: process of verifying the type safety of a program is based on analysis of a program's text (source code).
Pros:

- Allows many type errors to be caught beforehand.
- May be more efficient (faster/reduced memory usage) by omitting runtime type checks and enabling other optimizations.
- Help in the design of a clear and well-structured programs.
- Easier to fix type errors.
- Less ambiguity.
- Easier to work with other systems which rely on static types.
- May be too rigid.

Cons:

- Programmer must specify what type each variable is.
- Verbose type declarations.
- Complex error messages.
- Once you set a variable to a type, you cannot change it.
- May reject some programs that are well-behaved at run-time.
- Overly restrictive.

Dynamic type system: the process of verifying the type safety of a program is at runtime.
Pros:

- Programmers can write quicker.
- Less verbose.
- Makes the debug cycle much shorter.
- More tolerant to changes.
- Code is polymorphic without programmer decoration.
- More flexible.
- Allow you to provide information.

Cons:

- Interpreter can misinterpret the type of a variable.
- Accepts and attempts to execute some programs which may be ruled as invalid by a static type checker.
- Less documentation in the form of type signatures.
- Less efficient.