Tutorial #2 - A first flavor of types

Exercice 1: Simple extensions

In this exercise, we propose to extend the language so as to contain rules for booleans and integers. For the record, these rules are the following :

Syntax			Evaluation rules
t ::=	...	*expressions*	$\dfrac{t_1 \rightarrow_\beta t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \rightarrow_\beta \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3}$
	true, false	*booleans*	
	zero, succ t	*naturals*	
	if t then t else t	*if-then-else*	$\texttt{if true then } t_2 \texttt{ else } t_3 \rightarrow_\beta t_2$
	iszero t	*zero-equality*	$\texttt{if false then } t_2 \texttt{ else } t_3 \rightarrow_\beta t_3$
v ::=	...	*values*	$\dfrac{t \rightarrow_\beta t'}{\texttt{iszero } t \rightarrow_\beta \texttt{iszero } t'}$
	true, false	*boolean value*	
	nv	*numeric value*	$\texttt{iszero zero} \rightarrow_\beta \texttt{true}$
nv ::=		*numeric values*	$\texttt{iszero } (\texttt{succ t}) \rightarrow_\beta \texttt{false}$
	zero	*zero value*	
	succ nv	*successor value*	

The first objective consists in extending the language with the boolean expressions (`true`, `false` and `if .. then .. else`). For the sake of safety, copy your entire code in a new directory.

1. Extend the OCaml grammar for `term` with boolean expressions (4 constructors).
2. Modify the parser file `parser.mly` so as to use these constructors (this mostly consists in adapting the file comments to your code).
3. Extend the printer `term_to_string` and the matcher `is_value`.
4. Extend in order : `substitute`, `rename` and `reduce_one` (the `reduce` function should continue to work as before).

The goal now is to keep this version of your code safe (possibly by making a copy into a new directory), and apply the same operations to handle the natural numbers.

Syntax			Evaluation rules
t ::=	...	*expression*	$\dfrac{\texttt{t} \rightarrow_\beta \texttt{t'}}{\texttt{iszero t} \rightarrow_\beta \texttt{iszero t'}}$
	zero, succ t	*naturals*	
	iszero t	*zero-equality*	
v ::=	...	*values*	$\texttt{iszero zero} \rightarrow_\beta \texttt{true}$
	nv	*numeric value*	$\texttt{iszero (succ t)} \rightarrow_\beta \texttt{false}$
nv ::=		*numeric values*	
	zero	*zero value*	
	succ nv	*successor value*	

5. Extend the OCaml grammar for `term` with integer expressions (2 constructors).
6. Modify the parser file `parser.mly` so as to use these constructors (this mostly consists in adapting the file comments to your code), `term_to_string` and `is_value`.
7. Extend in order : `substitute`, `rename` and `reduce_one` (the `reduce` function should continue to work as before).

Exercice 2: From typed to untyped
Consider now adding types to our λ-calculus.

The set of all possible types is defined inductively as follows :

— *Type variables* : T, U ... are an infinite set of abstract type variables.

— *Type constants* : Nat, Bool ... are a finite set of constant type names.

— *Function type* : if T_1 and T_2 are types, then $\mathsf{T}_1 \to \mathsf{T}_2$ is also a type.

A type is said to be *concrete* if it contains no type variables as a sub-expression (and therefore is constructed only with constants and arrows).

1. Propose an OCaml grammar to represent types.

At first, we restrict ourselves to the language restricted to the booleans (this is a subset of the one studied before) :

Syntax			Evaluation rules
t ::=		*expressions*	$\dfrac{\mathtt{t}_1 \to_\beta \mathtt{t}_1'}{\mathtt{if\ t_1\ then\ t_2\ else\ t_3} \to_\beta \mathtt{if\ t_1'\ then\ t_2\ else\ t_3}}$
	true, false	*booleans*	
	if t then t else t	*if-then-else*	$\mathtt{if\ true\ then\ t_2\ else\ t_3} \to_\beta \mathtt{t_2}$
v ::=	...	*values*	$\mathtt{if\ false\ then\ t_2\ else\ t_3} \to_\beta \mathtt{t_3}$
	true, false	*boolean value*	

Following the model given for propositional logic, we propose to construct a proof that an expression is correctly typed using a system of sequents. For this, we define a *typing* e : T as a pair of a λ-expression and a type. An *environment* Γ is a list of typings.

To *deduce a typing* from Γ, noted $\Gamma \vdash$ t : T, consists in proving that the expression t is related to a type T by building a derivation tree using Γ as a set of axioms and a finite set of typing rules. The properties of the typing-relation will result from the form of these typing rules.

An expression possessing a typing for which there is a derivation tree is said to be *typable*.

An *inference rule* for deducing a simple typing looks like :

$$\text{Hypothesis} \longrightarrow \frac{\Gamma \vdash \mathtt{v} : \mathsf{Nat}}{\Gamma \vdash \mathtt{iszero\ v} : \mathsf{Bool}} \longleftarrow \text{Deduction}$$

2. Propose a set of typing rules for the language restricted to boolean expressions (one per constructor).
3. Construct a derivation tree for the expression `if true then false else true`.

Now consider the previous language extended with natural values :

Syntax			Evaluation rules
t ::=	...	*expression*	
	zero, succ t	*naturals*	
	iszero t	*zero-equality*	$\dfrac{\texttt{t} \to_\beta \texttt{t'}}{\texttt{iszero t} \to_\beta \texttt{iszero t'}}$
v ::=	...	*values*	
	nv	*numeric value*	$\texttt{iszero zero} \to_\beta \texttt{true}$
			$\texttt{iszero (succ t)} \to_\beta \texttt{false}$
nv ::=		*numeric values*	
	zero	*zero value*	
	succ nv	*successor value*	

4. Propose a set of typing rules for this extension (again, one per constructor).
5. Construct a derivation tree for the expression : `if (iszero zero) then zero else (succ zero)`.
6. Write a `type_check` function in OCaml that, given an expression in this λ-calculus, returns a boolean expressing whether this expression is typable or not.

And finally for the remainder of the language :

Syntax			Evaluation rules
t ::=		*expressions*	
	x	*variable*	$\dfrac{\texttt{t}_1 \to_\beta \texttt{t}_1'}{(\texttt{t}_1\ \texttt{t}_2) \to_\beta (\texttt{t}_1'\ \texttt{t}_2)}$
	λx.t	*abstraction*	
	(t t)	*application*	$\dfrac{\texttt{t} \to_\beta \texttt{t'}}{(\texttt{v}\ \texttt{t}) \to_\beta (\texttt{v}\ \texttt{t'})}$
v ::=		*values*	
	λx.t	*abstraction value*	$(\lambda\texttt{x.t}_1\ \texttt{t}_2) \to_\beta [\texttt{x} \mapsto \texttt{t}_2]\texttt{t}_1$

7. Propose a set of typing rules for the full λ-calculus (again, one per constructor).
   Remark : these rules are more complex than the previous ones, but can be deduced without too much difficulty from the ones for propositional logic.
8. (*Difficult*) Extend the `type_check` function to handle these new constructors (and explain how you do it with some semblance of precision)

   or

   (*Somewhat less difficult*) Express why it is difficult to extend the `type_check` function (possibly with examples of difficult cases).