Project 4: Simply Typed Lambda Calculus Extensions

Deadline: November 5, 23:59 (1 week only) **Submission:** Moodle

1 Assignment

In this exercise, we reuse the combinator parsing library introduced in exercise 3. The project skeleton is the same one you started with for the last assignment. You should continue working on your project, and the final submission should contain *all extensions*, both from this assignment and from the last assignment. You will only be graded on the extensions.

In this assignment you will extend your simply typed lambda calculus project from exercise 3 with *sum types* and *recursive* functions.

1.1 Sum Types

Sum types are handy when one needs to handle heterogeneous collections of values. Imagine you have a data type like a tree, where each element can be a leaf (containing a value) or an inner node (containing two references to subtrees). A sum type is a type which draws its values from exactly two other types. In other words, values of a sum type T_1 + T_2 can be either of type T_1 or of type T_2.

We introduce now syntactic rules for sum types, together with evaluation and typing rules.

$t ::= \;..$	terms
$\quad \mid\; "inl"\; t\; "as"\; T$	inject left
$\quad \mid\; "inr"\; t\; "as"\; T$	inject right
$\quad \mid\; "case"\; t\; "of"\; "inl"\; x\; "=>"\; t\; "\mid"\; "inr"\; x\; "=>"\; t$	case
$v ::= \;..$	values
$\quad \mid "inl"\; v\; "as"\; T$	
$\quad \mid "inr"\; v\; "as"\; T$	
$T ::= \;..$	types
$\quad \mid T\; "+"\; T$	sum type (right associative)

Fig. 1. λ-calculus with sum types

We create values of a sum type by *injecting* a value in the left or right "part". To deconstruct such values, we use a `case` expression, similar to Scala's `match` construct.

Sum types are right associative and `+` has the same precedence as $*$.

$$\mathit{case}\ (\mathit{inl}\ v_0)\ \mathit{of}\ \mathit{inl}\ x_1\ \texttt{=>}\ t_1 \mid \mathit{inr}\ x_2\ \texttt{=>}\ t_2 \to [x_1 \to v_0]t_1$$

$$\mathit{case}\ (\mathit{inr}\ v_0)\ \mathit{of}\ \mathit{inl}\ x_1\ \texttt{=>}\ t_1 \mid \mathit{inr}\ x_2\ \texttt{=>}\ t_2 \to [x_2 \to v_0]t_2$$

$$\frac{t \to t'}{\mathit{case}\ t\ \mathit{of}\ \mathit{inl}\ x_1\ \texttt{=>}\ t_1 \mid \mathit{inr}\ x_2\ \texttt{=>}\ t_2 \to \mathit{case}\ t'\ \mathit{of}\ \mathit{inl}\ x_1\ \texttt{=>}\ t_1 \mid \mathit{inr}\ x_2\ \texttt{=>}\ t_2}$$

$$\frac{t \to t'}{\mathit{inl}\ t \to \mathit{inl}\ t'}$$

$$\frac{t \to t'}{\mathit{inr}\ t \to \mathit{inr}\ t'}$$

Fig. 2. Language reduction rules for λ-calculus with sum types

And last, the additional typing rules are:

$$\frac{\begin{array}{c}\Gamma \vdash t_0 : T_1\ +\ T_2 \\ \Gamma, x_1 :\ T_1 \vdash t_1 : T \\ \Gamma, x_2 :\ T_2 \vdash t_2 : T\end{array}}{\Gamma \vdash \mathit{case}\ t_0\ \mathit{of}\ \mathit{inl}\ x_1\ \texttt{=>}\ t_1 \mid \mathit{inr}\ x_2\ \texttt{=>}\ t_2\ :\ T}$$

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \mathit{inl}\ t_1\ \mathit{as}\ T_1\ +\ T_2\ :\ T_1\ +\ T_2}$$

$$\frac{\Gamma \vdash t_1 : T_2}{\Gamma \vdash \mathit{inr}\ t_1\ \mathit{as}\ T_1\ +\ T_2\ :\ T_1\ +\ T_2}$$

Fig. 3. Typing rules for sum types

Evaluation and typing rules are pretty straight forward, the only thing a bit out of ordinary is the type adornment for `inl` and `inr`. This is necessary because the two constructors can't possibly know what is the type of the other component of the sum type. By giving it explicitly, the type checker is able to proceed. Alternatives to this decision will be explored in other exercises, when we know about type inference.

1.2 The fix operator

To bring back the fun into the game, we need some way to write non terminating programs (or at least, looping programs). Since types arrived, we were unable to type the fixpoint operator, and indeed there's a theorem which states that all well typed terms in simply typed lambda calculus evaluate to a normal form. The only alternative is to add it into the language.

$$\begin{array}{rll} t ::= & .. & \text{terms} \\ & \mid\ "fix"\ t & \text{fixed point of t} \end{array}$$

Fig. 4. λ-calculus with fix

$$fix\ (\lambda x:\ T_1.t_2)\ \rightarrow [x \rightarrow fix\ (\lambda x:\ T_1.t_2)]t_2$$

$$\frac{t \rightarrow t'}{fix\ t \rightarrow fix\ t'}$$

Fig. 5. Reduction rules for fix

$$\Gamma \vdash t_1\ :\ T_1 -> T_1$$

$$\Gamma \vdash fix\, t_1 :\ T_1$$

Fig. 6. Typing rules for fix

In order to make the fixpoint operator easier to use, add the following derived form:

`letrec x:` T_1 `=` t_1 `in` t_2 $\Rightarrow$ `let x = fix (`λ`x:`T_1`.`t_1`) in` t_2

Don't forget that we implement let as a derived form as well, so the actual desugaring is a bit more complex.

2 Implementation

Here is a summary of what you need to implement for this assignment:

- Extend your parser with `sum types` and `fix`
- A type checker which given a term finds its type (or it prints an error message).
- Extend the call-by-value reducer (small step) to account for the new constructs.

3 Input/Output

Your program should read a string from standard input until end-of-file is encountered, which represents the input program. If the program is syntactically correct, it should print its type and then print each step of the small-step reduction, starting with the input term, until it reaches a normal value. You should not worry about non-terminating programs. If there is a type error, it should print the error message *and the position where it occured* and skip the reduction. *The provided framework already implements this behavior. You should use it as-is.*

4 Hints

- Don't forget to override the method `toString()` in the subclasses of class `Term` in order to get a clean output!
- You should maintain positions in your abstract syntax trees. This is done by using `positioned` around parsers (the skeleton project already has them in place). This method takes care of updating the position on your trees, during parsing. Type errors should mention tree positions (have a look at class `TypeError` to see how it's done).