Project 3: Simply Typed Lambda Calculus

Deadline: October 29, 23:59 **Submission:** FoS submission email **Code:** 3-typed.zip

1 Assignment

The goal of this exercise is to familiarize yourself with the simply typed λ-calculus; your work consists of implementing a type checker and a reducer for simply typed λ-terms. To make it more interesting, we extend λ-calculus with boolean and integer values, as well as let and pair constructs. This time we'll fix a call-by-value strategy, so there will be only one reducer you need to write (actually, you can reuse some of the reducer from the previous assignment.

We first introduce the syntax for typed λ-calculus without *let* and *pairs*, which we'll add later. It might be a good idea to start implementing this language first and later add the rest. The syntax is presented in Figure 1.

$t ::=$ "$true$"	true value
| "$false$"	false value
| "if" t_1 "$then$" t_2 "$else$" t_3	if
| $numericLit$	integer
| "$pred$" t	predecessor
| "$succ$" t	successor
| "$iszero$" t	iszero
| x	variable
| "\" x ":" T "." t	abstraction
| $t\ t$	application (left associative)
| "(" t ")"	
$v ::=$	application (values)
| "$true$"	
| "$false$"	
| nv	numeric value
| "\" x ":" T "." t	abstraction value
$nv ::=$	numeric values
| "0"	
| "$succ$" nv	

Fig. 1. λ-calculus syntax

Note: As in the first assignment, we add syntactic sugar for numeric literals. They are desugared to the corresponding sequence of `succ succ ... 0`, as described before.

The only new thing in the above rules is the type annotation for lambda abstractions. We see that the variable name is followed by colon and a type. It roughly says "this function expects an argument of type T". In the above rules, T stands for types, and here's the syntax for types:

$T ::= \text{"}Bool\text{"}$	boolean type
$\mid \text{"}Nat\text{"}$	numeric type
$\mid T \text{"}->\text{"} T$	function types (right associative)
$\mid \text{"(" } T \text{ ")"}$	

There are three kinds of types in this language: booleans, natural numbers and function types. The type constructor " $\rightarrow$ " is right-associative – that is, the expression $T_1 \rightarrow T_2 \rightarrow T_3$ stands for $T_1 \rightarrow (T_2 \rightarrow T_3)$ [TAPL, p.100].

Reduction rules for this language are presented in Figure 2. You may note that they already fix the evaluation strategy to call-by-value, and that the type of an abstraction is ignored during evaluation. We can say that evaluation of simply typed lambda terms proceeds exactly the same as for untyped lambda terms. The operation of stripping off type annotations is called *erasure* and it is what enabled the addition of Java generics without modifying the virtual machine.

The typing rules if Figure 3 define a typing relation, similar to the evaluation relation. However, while evaluation is a relation between terms, typing is a relation between terms, types and contexts. A typing rule like the first one can be read "under context gamma, term *true* has type *Bool*". The role of the context (or environment) is to keep around a mapping between variable names and their types. It will be used to type free variables, when they are encountered. This is illustrated by the variable rule which can be read "under context gamma, variable x has type T provided context gamma has a binding for variable x to type T". The purpose of this type system is to prevent "bad things" to happen. So far, the only bad thing we know is stuck terms, and this type system prevents stuck terms. In other words, a term that can be assigned a type (it type checks) is guaranteed not to get stuck. The result of its evaluation will be a value.

2 Adding *let* and *pairs*

Let us now proceed to the addition of let:

$$\begin{aligned} t ::= & \ldots \\ & \mid \text{"}let\text{" } x \text{ " : " } T \text{ " = " } t \text{ "}in\text{" } t \end{aligned}$$

We can define let in terms of the existing concepts, and this has the advantage that once the translation is done in the parser, no addition is necessary to the type checker or to the evaluator. Such an addition is called a derived

Computation

$$if\ true\ then\ t_1\ else\ t_2 \rightarrow t_1 \qquad pred\ 0 \rightarrow 0$$

$$if\ false\ then\ t_1\ else\ t_2 \rightarrow t_2 \qquad pred\ succ\ NV \rightarrow NV$$

$$iszero\ 0 \rightarrow true$$

$$iszero\ succ\ NV \rightarrow false \qquad (\lambda x:\ T.t_1)\ v_2 \rightarrow [x \rightarrow v_2]\ t_1$$

Congruence

$$\frac{t_1 \rightarrow t_1'}{if\ t_1\ then\ t_2\ else\ t_3 \rightarrow if\ t_1'\ then\ t_2\ else\ t_3} \qquad \frac{t \rightarrow t'}{succ\ t \rightarrow succ\ t'}$$

$$\frac{t \rightarrow t'}{iszero\ t \rightarrow iszero\ t'} \qquad \frac{t_1 \rightarrow t_1'}{t_1\ t_2 \rightarrow t_1'\ t_2}$$

$$\frac{t \rightarrow t'}{pred\ t \rightarrow pred\ t'} \qquad \frac{t_2 \rightarrow t_2'}{v_1\ t_2 \rightarrow v_1\ t_2'}$$

Fig. 2. Language reduction rules

$$\Gamma \vdash true:\ Bool \qquad \frac{\Gamma \vdash t_1:\ Bool \quad \Gamma \vdash t_2:\ T \quad \Gamma \vdash t_3:\ T}{\Gamma \vdash if\ t_1\ then\ t_2\ else\ t_3:\ T}$$

$$\Gamma \vdash false:\ Bool$$

$$\Gamma \vdash 0:\ Nat \qquad \frac{x:\ T \in \Gamma}{\Gamma \vdash x:\ T}$$

$$\frac{\Gamma \vdash t:\ Nat}{\Gamma \vdash pred\ t:\ Nat} \qquad \frac{\Gamma,\ x:\ T_1 \vdash t:\ T_2}{\Gamma \vdash \lambda x:\ T_1.t_2 : T_1 \rightarrow T_2}$$

$$\frac{\Gamma \vdash t:\ Nat}{\Gamma \vdash succ\ t:\ Nat} \qquad \frac{\Gamma \vdash t_1:\ T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2:\ T_{11}}{\Gamma \vdash t_1\ t_2:\ T_{12}}$$

$$\frac{\Gamma \vdash t:\ Nat}{\Gamma \vdash iszero\ t:\ Bool}$$

Fig. 3. Language typing rules

form. The language that is accepted by our parser is called external language and the language understood by the evaluator (and type checker) is called internal language. And here is the translation of let in terms of abstraction and application:

$$\text{"}let\text{"}\ x\ \text{" : "}\ T\ \text{" = "}\ t_1\ \text{"}in\text{"}\ t_2\ \Rightarrow\ (\backslash x :\ T.\, t_2)\, t_1$$

To add pairs, we can't do the same trick so we'll need to extends the existing syntax, evaluation and typing rules:

$$\begin{array}{l} t ::= ... \\ \quad \mid \text{"\{"}\ t\ \text{","}\ t\ \text{"\}"} \\ \quad \mid \text{"}fst\text{"}\ t \\ \quad \mid \text{"}snd\text{"}\ t \\ \\ v ::= ... \\ \quad \mid \text{"\{"}\ v\ \text{","}\ v\ \text{"\}"} \\ \\ T ::= ... \\ \quad \mid T\ \text{" * "}\ T \qquad\qquad \text{(right associative)} \end{array}$$

The first form creates a new pair, and the other two are called projections, and extract the first and the second element of a pair. We add a new kind of values, pair values, and a new kind of type, for the corresponding pair type. We decide that the pair type constructor (denoted by $*$) takes precedence over the arrow constructor, so $Nat * Nat \to Bool$ is parsed as $(Nat * Nat) \to Bool$, and that function application takes precedence over projections so $snd\ x\ y$ is parsed as $snd\ (x\ y)$. The new evaluation rules are presented in Figure 4 and typing rules in Figure 5.

$$fst\ \{v_1,\ v_2\} \to v_1$$

$$snd\ \{v_1,\ v_2\} \to v_2$$

$$\frac{t \to t'}{fst\ t \to fst\ t'}$$

$$\frac{t \to t'}{snd\ t \to snd\ t'}$$

$$\frac{t_1 \to t_1'}{\{t_1, t_2\} \to \{t_1', t_2\}}$$

$$\frac{t_2 \to t_2'}{\{v_1, t_2\} \to \{v_1, t_2'\}}$$

Fig. 4. Reduction rules for *pairs*

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{t_1, t_2\} : T_1 * T_2} \qquad \frac{\Gamma \vdash t : T_1 * T_2}{\Gamma \vdash snd\, t : T_2}$$

$$\frac{\Gamma \vdash t : T_1 * T_2}{\Gamma \vdash fst\, t : T_1}$$

Fig. 5. Typing rules for *pairs*

3 Implementation

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

- A parser for the given grammar (including let and pairs).
- A type checker which given a term finds its type (or it prints an error message).
- A call-by-value reducer (small step) with the corresponding path method which gives back the stream of intermediate terms.

3.1 Input/Output

Your program should read a string from standard input until end-of-file (EOF) 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 (remember that simply typed lambda calculus is strongly normalizing). If there is a type error, it should print the error message and the position where it occurred and skip the reduction. The provided framework already implements this behavior. You should use it as-is. Below are example inputs for simple programs:

```
For input:
(\x:Nat->Bool. (\y:Nat.(x y))) (\x:Nat.(iszero x)) 0
Result:
typed: Bool
(\x:Nat->Bool.(\y:Nat.x y)) (\x:Nat.iszero x) 0
(\y:Nat.(\x:Nat.iszero x) y) 0
(\x:Nat.iszero x) 0
iszero 0
true

For input:
(\x:Nat.x) true
Result:
parameter type mismatch: expected Nat, found Bool
```

```
(\x:Nat.x) true

For input:
(\x:Nat.snd x) 1
Result:
pair type expected but Nat found
(\x:Nat.snd x) 1
```

Note: please try to match the output of the above examples - it makes the grading process easier.

3.2 Hints

As in the previous exercise, the project is supplied with a SBT project file and a starting point for your project. 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).