

1 Overview

The aim of this practical work is to familiarize you with the Prolog language while learning stack-based languages.

As with the previous labs, the steps are as follows:

1. Improve your knowledge of Prolog.
2. Read and understand this data.
3. Read, find, and understand the code provided.
4. Complete the code provided.
5. Write a report. It should describe your experience during the points previous: problems encountered, surprises, choices you had to make, options that you have knowingly rejected, etc. The report should not exceed 5 pages.

2 The Cat language (just a made up name given to the project not a real language)

You are going to write a Prolog program which implements the static semantics

(i.e. type inference) and dynamic semantics (i.e. an interpreter) of a little flip language. Push-on languages are languages which have the particularity

to use a value stack instead of local variables. E.g. Postscript is a push-on language.

The basic principle is very simple: the code consists of a sequence of operations that modify the stack. E.g. **dup** takes the item at the top of the stack and adds a copy on top. **add** takes both elements at the top of the stack, adds them up and replaces them with the result. The operation **push V** adds the value V to the top of the stack; many stack languages drop the push keyword to leave only V to add V to the top of the battery. Thus a code such as [12 75 dup add add] amounts to add 162 to the top of the stack.

The Cat language that you need to implement knows the following Val values:

<i>Num</i>	A number.(integer)
<i>true</i>	The true boolean.
<i>false</i>	The false boolean.
<i>nil</i>	The empty list.
<i>X[^]Xs</i>	List made up of element X and list Xs.
<i>[Ops...]</i>	Anonymous function made up of Ops operations.

The Ops operations are as follows (I use ToS (Top of Stack) to refer at the top of the stack):

<i>dup</i>	Duplicates the ToS.
<i>swap</i>	Change both to ToS elements.
<i>pop</i>	Remove the element from ToS.
<i>get(N)</i>	Gets the Nth item on the stack and add a copy to the ToS. <i>get(0) == dup</i>
<i>set(N)</i>	Take the item from the ToS and then replace the N-th item with this value. <i>[set(0)] == [swap pop]</i>
<i>Val</i>	Adds the value <i>Val</i> to ToS.
<i>add</i>	Replace the two elements in ToS by their sum.
<i>if</i>	Replaces the three elements at ToS by the second or the third depending on whether the first is true or false.
<i>cons</i>	Replace the two elements X and Xs in ToS with the list X [^] X s.
<i>empty?</i>	Replaces the list in ToS by the boolean which indicates if it is empty.
<i>car</i>	Replaces the ToS list with its head.
<i>cdr</i>	Replaces the ToS list with its tail.
<i>apply</i>	Calls the function that is in ToS.
<i>papply</i>	Partial application: takes the function F and the value V to the ToS and replaces them with a new function (a closure) which, when called, executes F after adding V to the ToS.

Op : Type Explicit type annotation. Behaves like *Op*.

So `[13 [add] papply]` constructs a function which increments the ToS by 13.

The function `[]` which does not contain any operation corresponds to the function identity.

So `[7 [] papply]` constructs a function which adds the number 7 to the top of stack and `[6 7 [] papply papply]` constructs a function that adds 6 and 7 to the stack.

$\vdash Val : T$ Value Val a type T

$$\begin{array}{c}
\overline{\vdash Num : \text{int}} \quad \overline{\vdash \text{true} : \text{bool}} \quad \overline{\vdash \text{false} : \text{bool}} \quad \frac{T \text{ is a type}}{\vdash \text{nil} : \text{list}(T)} \\
\\
\frac{\vdash X : T \quad \vdash Xs : \text{list}(T)}{\vdash X \wedge Xs : \text{list}(T)} \quad \overline{\vdash [] : ST \rightarrow ST} \\
\\
\frac{\text{For all } 1 \leq i \leq n \quad \vdash Op_i : ST_{i-1} \Rightarrow ST_i}{\vdash [Op_1 \dots Op_n] : ST_0 \rightarrow ST_n}
\end{array}$$

Fig 1: Rules for typing values.

$\vdash Op : ST_1 \Rightarrow ST_2$ Op changes a stack of type ST_1 into one of type ST_2

$$\begin{array}{c}
\overline{\vdash \text{dup} : T \wedge ST \Rightarrow T \wedge T \wedge ST} \quad \overline{\vdash \text{swap} : T_1 \wedge T_2 \wedge ST \Rightarrow T_2 \wedge T_1 \wedge ST} \\
\\
\overline{\vdash \text{pop} : T \wedge ST \Rightarrow ST} \quad \overline{\vdash \text{get}(N) : T_0 \wedge \dots \wedge T_N \wedge ST \Rightarrow T_N \wedge T_0 \wedge \dots \wedge T_N \wedge ST} \\
\\
\overline{\vdash \text{set}(N) : T \wedge T_0 \wedge \dots \wedge T_N \wedge ST \Rightarrow T_0 \wedge \dots \wedge T \wedge ST} \quad \frac{\vdash val : T}{\vdash Val : ST \Rightarrow T \wedge ST} \\
\\
\overline{\vdash \text{add} : \text{int} \wedge \text{int} \wedge ST \Rightarrow \text{int} \wedge ST} \quad \overline{\vdash \text{if} : \text{bool} \wedge T \wedge T \wedge ST \Rightarrow T \wedge ST} \\
\\
\overline{\vdash \text{cons} : T \wedge \text{list}(T) \wedge ST \Rightarrow \text{list}(T) \wedge ST} \\
\\
\overline{\vdash \text{empty} : \text{list}(T) \wedge ST \Rightarrow \text{bool} \wedge ST} \quad \overline{\vdash \text{car} : \text{list}(T) \wedge ST \Rightarrow T \wedge ST} \\
\\
\overline{\vdash \text{cdr} : \text{list}(T) \wedge ST \Rightarrow \text{list}(T) \wedge ST} \quad \overline{\vdash \text{apply} : (ST_1 \rightarrow ST_2) \wedge ST_1 \Rightarrow ST_2} \\
\\
\overline{\vdash \text{papply} : ((T \wedge ST_1) \rightarrow ST_2) \wedge T \wedge ST_3 \Rightarrow (ST_1 \rightarrow ST_2) \wedge ST_3} \\
\\
\frac{\vdash Op : ST \Rightarrow T \wedge ST}{\vdash Op : T : ST \Rightarrow T \wedge ST}
\end{array}$$

Fig 2: Rules for typing operation.

3 The type system

The Cat language is typed similarly to Haskell, that is, types are generally inferred, and it provides parametric polymorphism. The stack state is described by an ST type which can have the following forms:

nil Empty stack type
 $T \wedge ST$ Stack top has type T and the rest has type ST.

The possible types of values are:

int Type of numbers.
 $bool$ Type of booleans.
 $list(T)$ Element list of type T
 $ST_1 \rightarrow ST_2$ Functions from an ST1 type stack to an ST2 type stack.
 $\forall t.T$ Polymorphic type.

The typing rules are given in Figs. 1 and 2. These rules do not hold polymorphism account. When mixing polymorphism and inference of types, we easily obtain a non-decidable system, that is to say a system where it may happen that the type inference does not end. We can also easily get an inconsistent system where the inferred type is incorrect. To avoid these 2 problems, we will limit the polymorphism as follows:

Only functions can be polymorphic. I.e. only types that are like $\forall t_0 \dots \forall t_n. ST_1 \rightarrow ST_2$ are allowed. Not $\forall t. list(T)$ or $\forall t.t$ nor of $(\forall t. ST_1) \rightarrow ST_2$.

The two typing rules that govern polymorphism are the elimination rule:

$$\frac{\vdash Val : \forall t.T}{\vdash Val : T[T'/t]}$$

which says that if a value Val is polymorphic then we can eliminate the polymorphism by specializing the type parameter t to a particular type T' . The notation $T[T'/t]$ signifies the substitution of the variable t by the term T' in the term T.

The rule for introducing polymorphism is:

$$\frac{\text{for all } T' \quad \vdash Fun : T[T'/t]}{\vdash Fun : \forall t.T}$$

Which says that if Fun has type T $[T'/t]$ whatever the term T' then it also has the type polymorphic $\forall t.T$. Of course, we cannot try all types to verify if this is true, but if T' is **not instantiated** Prolog variable then the result is the same, since a term with such a variable represents the set of terms that can be obtained by instantiating this variable.

4 The code provided

The code provided shows the syntax of the terms you need to manipulate. This syntax is described as declarations of Prolog predicates. These predicates may not be useful to you in your code and therefore may be left unused, but feel free to use them if it suits you. Their purpose is to tell you which representation to use to encode the terms, values and types of the Cat language in Prolog.

The two most important predicates (the entry points of the code) are ***eval*** and ***typeof***.

5 What you should do

You have a few predicates to fill in including `typeof val`, which infers / checks the type of a value, `typeof op`, which infers / checks the type of an operation, and `eval` which evaluates a Cat program. I also recommend that you start by inferring types regardless of polymorphism and adding polymorphism only after the rest of the type inference is working.

Unlike typing rules which are complete and formal, the description of the behavior of each primitive (in section 2) is informal. If you have doubts about certain details, the typing rules are usually good for eliminating ambiguities, but if you still have doubts, choose the behavior that seems to you most useful and / or natural and explain the problem and your choice in the report.

6 Notes

You must submit two files: `cat.prolog` and `rapport.word`

- You can, of course, define as many new function as you want, but you should not change any function other than the ones mentioned. If, however, you find it necessary to change them, write down and justify the change in the report.
- Each line of code must be less than 80 characters. Any excess will be considered an error.