Lab 2

DD2481 - Principles of Programming Languages

13 April 2022

1 Goal

The main goal of this second lab is the extension of lab 1 with type checking and additional constructs. We provide a changed and extended code template to enable parsing of the adapted syntax. Additionally, we extended the simple test infrastructure so that you can add test cases easily.

2 Syntax

The complete and updated syntax follows. Notice the additional syntax required for typing the variables when they are introduced in lambda abstraction terms. Furthermore, we add contexts and use them later as environments for type checking. These environments will get extended when entering lambda abstractions during type checking.

t	::=		*terms* :
	|	true	*constant true*
	|	false	*constant false*
	|	if t then t else t	*conditional*
	|	c	*integer constant*
	|	$t + t$	*integer addition*
	|	$-t$	*integer negation*
	|	$t < t$	*integer comparison*
	|	x	*variable*
	|	$\lambda x : T.\ t$	*abstraction*
	|	$t\ t$	*application*
	|	let $x = t$ in t	*variable binding*
	|	fix t	*fix point application*

$$
\begin{array}{rcll}
v & ::= & & \textit{values}: \\
 & | & \mathsf{true} & \textit{true value} \\
 & | & \mathsf{false} & \textit{false value} \\
 & | & c & \textit{integer value} \\
 & | & \lambda x : T.\ t & \textit{abstraction value}
\end{array}
$$

$$
\begin{array}{rcl}
(c & \in & \{\ldots, -2, -1, 0, 1, 2, \ldots\}) \\
(x & \in & \{\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{y}_1, \mathtt{y}_2, \ldots, \mathtt{z}_1, \mathtt{z}_2, \ldots, \ldots\})
\end{array}
$$

$$
\begin{array}{rcll}
T & ::= & & \textit{types}: \\
 & | & \mathsf{bool} & \textit{boolean} \\
 & | & \mathsf{int} & \textit{integer} \\
 & | & T \to T & \textit{arrow type}
\end{array}
$$

$$
\begin{array}{rcll}
\Gamma & ::= & & \textit{contexts}: \\
 & | & \emptyset & \textit{empty context} \\
 & | & \Gamma, x : T & \textit{term variable binding}
\end{array}
$$

3 Semantics rules

The complete and updated small-step evaluation rules follow.

E-IfTrue
$\mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3 \to t_2$

E-IfFalse
$\mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3 \to t_3$

E-If
$$\frac{t_1 \to t_1'}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \to \mathsf{if}\ t_1'\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3}$$

E-Add
$c_1 + c_2 \to c_1\ \mathcal{I}(+)\ c_2$

E-AddRight
$$\frac{t_2 \to t_2'}{c_1 + t_2 \to c_1 + t_2'}$$

E-AddLeft
$$\frac{t_1 \to t_1'}{t_1 + t_2 \to t_1' + t_2}$$

E-MinusVal
$-c \to \mathcal{I}(-)\ c$

E-Minus
$$\frac{t \to t'}{-t \to -t'}$$

E-LessThan
$c_1 < c_2 \to c_1\ \mathcal{I}(<)\ c_2$

E-LessThanRight
$$\frac{t_2 \to t_2'}{c_1 < t_2 \to c_1 < t_2'}$$

E-LessThanLeft
$$\frac{t_1 \to t_1'}{t_1 < t_2 \to t_1' < t_2}$$

E-App1
$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}$$

E-App2
$$\frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'}$$

E-AppAbs
$(\lambda x.\ t_{12})\ v_2 \to [x \mapsto v_2]t_{12}$

E-LetV
$\mathsf{let}\ x = v_1\ \mathsf{in}\ t_2 \to [x \mapsto v_1]t_2$

E-Let
$$\frac{t_1 \to t_1'}{\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \to \mathsf{let}\ x = t_1'\ \mathsf{in}\ t_2}$$

E-FixBeta
$\mathsf{fix}\ (\lambda x : T_1.\ t_2) \to [x \mapsto (\mathsf{fix}\ (\lambda x : T_1.\ t_2))]t_2$

E-Fix
$$\frac{t_1 \to t_1'}{\mathsf{fix}\ t_1 \to \mathsf{fix}\ t_1'}$$

4 Typing rules

The corresponding typing rules follow. Notice that the typing context Γ, or sometimes environment, accounts for bound variables in the respective scopes.

$$\text{T-True}\quad \Gamma \vdash \mathsf{true} : \mathsf{bool} \qquad \text{T-False}\quad \Gamma \vdash \mathsf{false} : \mathsf{bool} \qquad \text{T-Int}\quad \Gamma \vdash c : \mathsf{int}$$

$$\text{T-If}\quad \frac{\Gamma \vdash t_1 : \mathsf{bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T}$$

$$\text{T-Add}\quad \frac{\Gamma \vdash t_1 : \mathsf{int} \quad \Gamma \vdash t_2 : \mathsf{int}}{\Gamma \vdash t_1 + t_2 : \mathsf{int}} \qquad \text{T-Minus}\quad \frac{\Gamma \vdash t : \mathsf{int}}{\Gamma \vdash -t : \mathsf{int}} \qquad \text{T-LessThan}\quad \frac{\Gamma \vdash t_1 : \mathsf{int} \quad \Gamma \vdash t_2 : \mathsf{int}}{\Gamma \vdash t_1 < t_2 : \mathsf{bool}}$$

$$\text{T-Var}\quad \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{T-Abs}\quad \frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.\ t_2 : T_1 \to T_2} \qquad \text{T-App}\quad \frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$$

$$\text{T-Let}\quad \frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : T_2} \qquad \text{T-Fix}\quad \frac{\Gamma \vdash t_1 : T_1 \to T_1}{\Gamma \vdash \mathsf{fix}\ t_1 : T_1}$$

5 Task

As in the last lab, the new interpreter should be based directly on the small-step evaluation rules above. Additionally, the typechecking rules above have to be implemented. In order to convince yourself that you implemented the rules correctly you will develop a small test suite containing test programs and expected outputs. Make sure that you also test the typechecker, with correct and wrong inputs to see it succeeding and failing when it should.

Start by creating a branch of your final version of the interpreter from lab 1 in your git repository. This is required because we changed the language we used so far. Because of this, you may need to translate your existing test cases and develop new ones as well. Check the changes we made to the test cases we provided to see how this can be done.

After implementing the base language, you will implement the extensions tuples, records, and lists. The corresponding syntax, semantics and

typechecking rules for this are presented in Section 8. Make sure that your implementation is directly based on the presented rules.

Follow these steps to structure your work for this lab.

1. Create a branch for lab 1 and one for lab 2,
2. merge your code with the provided template in your repository,
3. adjust your existing rules to match the ones above,
4. test the adjustments,
5. incrementally
   - implement the new rules above and
   - test,
6. continue in the same manner with the extensions in Section 8.

6 Parsing

In the later part of the lab, you are going to extend your interpreter and typechecker with tuples, records, and lists. Notice that the parser and AST of the code template already includes all constructs for your convenience. You should extend your implementation step by step as you advance throughout the lab. See also the corresponding test cases for examples on how to write the respective constructs in the `sint` language.

The parser will automatically handle the following derived from so that we can use `letrec` to define recursive functions in the input. This is the so called "syntactic sugar" and represents a fix point expression internally. In this way, we do not have to add extra evaluation and typing rules.

$$\mathsf{letrec}\ x : T_1 = t_1\ \mathsf{in}\ t_2 \stackrel{\text{def}}{=} \mathsf{let}\ x = fix(\lambda x : T_1.\ t_1)\ \mathsf{in}\ t_2$$

Let us think a bit about the `let` construct in our language. The same trick could have been applied here as well. But instead we chose to apply a bit type inference as in the book's Section 11.5 to make usage more convenient. For a "syntactic sugaring", we would have needed to add a type annotation to our `let`. Then we could have a derived form like the following, effectively removing the rules for `let`.

$$\mathsf{let}\ x : T_1 = t_1\ \mathsf{in}\ t_2 \stackrel{\text{def}}{=} (\lambda x : T_1.\ t_2)\ t_1$$

Do you see that they have the same evaluation result under the rules above?

Further, notice that the arrow type operator is right associative. The function application is left associative though. Can you see why?

The binding strengths of the syntactical elements are the following in ascending order:

1. Ifthenelse, lambda abstraction, let, letrec, isnil, head, tail
2. Less than
3. Plus
4. Function application
5. List construction
6. Unary minus
7. Projection for tuples and records

7 Implementation

In order to keep the development manageable, it is advisable to start by disabling the typechecker. Integrated development and testing iterations of your semantics implementation would not be possible otherwise since it runs before evaluation. For this purpose add the following comment to the typechecker application in "Main.scala".

```
...
  def apply(filename: String):
    Either[InterpreterError, AST] = {
      for {
        input <- FileLoader(filename).right
        tokens <- Lexer(input).right
        ast <- Parser(tokens).right
        _ <- ASTPrinter(ast).right
        // add a comment here: _ <- Typechecker(ast).right
        res <- Interpreter(ast).right
      } yield res
  }
...
```

Do not forget to develop test cases in the provided test framework. Make sure that you understand the test cases we already provided as part of the code template. You can easily deactivate the ones testing not yet implemented parts of your interpreter by commenting the respective calls to the scala testing functions. In this way you can develop the rules together with the tests. When you run the `sbt test` command from time to time you will notice when something breaks that previously worked.

8 Extensions

After implementing the base language, we are adding the following three extensions to round out our interpreter. Add many test cases to convince yourself that your implementation is correct as usual. By testing individual evaluation steps like some of the provided test cases, you make sure that step-wise evaluation works as expected.

For each of the extensions, you need to develop a little test program of your choice as well. Create a core function for which you also describe in a comment what it is computing. Tuples can be useful for aggregated data like coordinates or similar. So why not creating a function for calculating a vector addition? Records can be used to represent objects where data and functions are mixed. So why not creating a record value which carries both data and a function taking as first input parameter a record of its own type? Lists bring a certain kind of dynamic into play. Now you could for example combine tuples or records with lists or simply create a sorting function over integers. Recursively defined algorithms like merge sort and quicksort can serve as a nice and rewarding exercise here.

Good luck!

8.1 Tuples

The syntax extension for tuples follows.

$$
\begin{array}{rcll}
t & ::= & \ldots & \textit{terms} : \\
 & | & (t_1, t_i\ ^{i \in 2..n}) & \textit{tuple} \\
 & | & t.i & \textit{projection}
\end{array}
$$

$$
\begin{array}{rcll}
v & ::= & \ldots & \textit{values} : \\
 & | & (v_1, v_i\ ^{i \in 2..n}) & \textit{tuple value}
\end{array}
$$

$$
\begin{array}{rcll}
T & ::= & \ldots & \textit{types} : \\
 & | & (T_1, T_i\ ^{i \in 2..n}) & \textit{tuple type}
\end{array}
$$

The semantics extension for tuples follows.

$$\text{E-ProjTuple}\quad (v_i\ ^{i\in 1..n}).j \to v_j \qquad \text{E-Proj}\quad \frac{t_1 \to t_1'}{t_1.i \to t_1'.i}$$

$$\text{E-Tuple}\quad \frac{t_j \to t_j'}{(v_i\ ^{i\in 1..j-1}, t_j, t_k\ ^{k\in j+1..n}) \to (v_i\ ^{i\in 1..j-1}, t_j', t_k\ ^{k\in j+1..n})}$$

The typing extension for tuples follows.

$$\text{T-Tuple}\quad \frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash (t_i\ ^{i\in 1..n}) : (T_i\ ^{i\in 1..n})} \qquad \text{T-Proj}\quad \frac{\Gamma \vdash t_1 : (T_i\ ^{i\in 1..n})}{\Gamma \vdash t_1.j : T_j}$$

8.2 Records

The syntax extension for records follows.

$$\begin{array}{rcll} t & ::= & \ldots & \textit{terms}: \\ & | & \{l_i = t_i\ ^{i\in 1..n}\} & \textit{record} \\ & | & t.l & \textit{projection} \end{array}$$

$$\begin{array}{rcll} v & ::= & \ldots & \textit{values}: \\ & | & \{l_i = v_i\ ^{i\in 1..n}\} & \textit{record value} \end{array}$$

$$\begin{array}{rcll} T & ::= & \ldots & \textit{types}: \\ & | & \{l_i : T_i\ ^{i\in 1..n}\} & \textit{record type} \end{array}$$

The semantics extension for records follows.

$$\text{E-ProjRcd}\quad \{l_i = v_i\ ^{i\in 1..n}\}.l_j \to v_j \qquad \text{E-Proj}\quad \frac{t_1 \to t_1'}{t_1.l \to t_1'.l}$$

$$\text{E-Rcd}\quad \frac{t_j \to t_j'}{\begin{array}{l}\{l_i = v_i\ ^{i\in 1..j-1}, l_j = t_j, l_k = t_k\ ^{k\in j+1..n}\} \\ \to \{l_i = v_i\ ^{i\in 1..j-1}, l_j = t_j', l_k = t_k\ ^{k\in j+1..n}\}\end{array}}$$

The typing extension for records follows.

$$\text{T-Rcd}\quad \frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i\ ^{i\in 1..n}\} : \{l_i : T_i\ ^{i\in 1..n}\}} \qquad \text{T-Proj}\quad \frac{\Gamma \vdash t_1 : \{l_i : T_i\ ^{i\in 1..n}\}}{\Gamma \vdash t_1.l_j : T_j}$$

8.3 Lists

The syntax extension for lists follows.

$$
\begin{array}{rcll}
t & ::= & \ldots & \textit{terms}: \\
 & | & [T] & \textit{empty list} \\
 & | & t :: t & \textit{list constructor} \\
 & | & \mathsf{isnil}\ t & \textit{test for empty list} \\
 & | & \mathsf{head}\ t & \textit{head of a list} \\
 & | & \mathsf{tail}\ t & \textit{tail of a list}
\end{array}
$$

$$
\begin{array}{rcll}
v & ::= & \ldots & \textit{values}: \\
 & | & [T] & \textit{empty list} \\
 & | & v :: v & \textit{list constructor}
\end{array}
$$

$$
\begin{array}{rcll}
T & ::= & \ldots & \textit{types}: \\
 & | & [T] & \textit{type of lists}
\end{array}
$$

The semantics extension for lists follows.

$$\textsc{E-Cons1}\ \frac{t_1 \to t_1'}{t_1 :: t_2 \to t_1' :: t_2} \qquad \textsc{E-Cons2}\ \frac{t_2 \to t_2'}{t_1 :: t_2 \to t_1 :: t_2'}$$

$$\textsc{E-IsnilNil}\ \mathsf{isnil}\ [T] \to \mathsf{true} \qquad \textsc{E-IsnilCons}\ \mathsf{isnil}\ v_1 :: v_2 \to \mathsf{false} \qquad \textsc{E-Isnil}\ \frac{t_1 \to t_1'}{\mathsf{isnil}\ t_1 \to \mathsf{isnil}\ t_1'}$$

$$\textsc{E-HeadCons}\ \mathsf{head}\ v_1 :: v_2 \to v_1 \qquad \textsc{E-Head}\ \frac{t_1 \to t_1'}{\mathsf{head}\ t_1 \to \mathsf{head}\ t_1'}$$

$$\textsc{E-TailCons}\ \mathsf{tail}\ v_1 :: v_2 \to v_2 \qquad \textsc{E-Tail}\ \frac{t_1 \to t_1'}{\mathsf{tail}\ t_1 \to \mathsf{tail}\ t_1'}$$

The typing extension for lists follows.

$$\textsc{T-Nil}\ \Gamma \vdash [T_1] : [T_1] \qquad \textsc{T-Cons}\ \frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : [T_1]}{\Gamma \vdash t_1 :: t_2 : [T_1]}$$

$$\textsc{T-Isnil}\ \frac{\Gamma \vdash t_1 : [T_{11}]}{\Gamma \vdash \mathsf{isnil}\ t_1 : \mathsf{bool}} \qquad \textsc{T-Head}\ \frac{\Gamma \vdash t_1 : [T_{11}]}{\Gamma \vdash \mathsf{head}\ t_1 : T_{11}} \qquad \textsc{T-Tail}\ \frac{\Gamma \vdash t_1 : [T_{11}]}{\Gamma \vdash \mathsf{tail}\ t_1 : [T_{11}]}$$