Programming Assignment 4 Write-Up

Mark Pfluger and Eric Shih

1 Introduction and Typing Rules

The type system that has been implemented in this assignment has six basic types and makes use of recursion to evaluate expressions. The basic types are String, Int, StringList, IntList, Variable, and FunctionLambda. These types were chosen because they were simple and generic enough to cover the entirety of the L language, but still specific enough to type all the necessary programs correctly. Recursion was chosen because it was deemed the most easily implementable system for our specific case. The recursion algorithm will be explained in detail in Section 3. The typing rules for L are listed below, along with the proofs for preservation and progress,in Section 2, to ensure that the type system is sound and will not "get stuck".

Typing Rules

$$\frac{Int\ i}{\Gamma \vdash\ i : Int} \qquad \frac{String\ s}{\Gamma \vdash\ s : String} \qquad \frac{Variable\ i}{\Gamma \vdash\ i : Variable}$$

$$\frac{\Gamma \vdash\ S1 :\ Int \quad \Gamma \vdash\ S2 :\ Int}{\Gamma \vdash\ S1 + S2 : Int} \qquad \frac{\Gamma \vdash\ S1 :\ Int \quad \Gamma \vdash\ S2 :\ Int}{\Gamma \vdash\ S1 \& S2 : Int}$$

$$\frac{\Gamma \vdash\ S1 :\ Int \quad \Gamma \vdash\ S2 :\ Int}{\Gamma \vdash\ S1 | S2 : Int} \qquad \frac{\Gamma \vdash\ S1 :\ Int \quad \Gamma \vdash\ S2 :\ Int}{\Gamma \vdash\ S1 - S2 : Int}$$

$$\frac{\Gamma \vdash\ S1 :\ Int \quad \Gamma \vdash\ S2 :\ Int}{\Gamma \vdash\ S1 * S2 : Int} \qquad \frac{\Gamma \vdash\ S1 :\ Int \quad \Gamma \vdash\ S2 :\ Int}{\Gamma \vdash\ S1 / S2 : Int}$$

$$\frac{\Gamma \vdash\ S1 :\ Int \quad \Gamma \vdash\ S2 :\ Int}{\Gamma \vdash\ S1 = S2 : Int} \qquad \frac{\Gamma \vdash\ S1 :\ Int \quad \Gamma \vdash\ S2 :\ Int}{\Gamma \vdash\ S1 <> S2 : Int}$$

$$\frac{\Gamma \vdash\ S1 :\ Int \quad \Gamma \vdash\ S2 :\ Int}{\Gamma \vdash\ S1 < S2 : Int} \qquad \frac{\Gamma \vdash\ S1 :\ Int \quad \Gamma \vdash\ S2 :\ Int}{\Gamma \vdash\ S1 <= S2 : Int}$$

$$\frac{\Gamma \vdash\ S1 :\ Int \quad \Gamma \vdash\ S2 :\ Int}{\Gamma \vdash\ S1 > S2 : Int} \qquad \frac{\Gamma \vdash\ S1 :\ Int \quad \Gamma \vdash\ S2 :\ Int}{\Gamma \vdash\ S1 >= S2 : Int}$$

$$\frac{\Gamma \vdash\ S1 :\ \tau_1 \quad \tau = \tau_1 \quad \Gamma[id \leftarrow \tau] \vdash\ S2 : \tau_3}{\Gamma \vdash\ letid = S1 in S2 :\ \tau_3} \qquad \frac{\Gamma \vdash\ S1 :\ list(\tau)}{\Gamma \vdash\ !S1 : \tau}$$

$$\frac{\Gamma \vdash\ S1 :\ \tau(\tau\ not\ list)}{\Gamma \vdash\ !S1 : \tau} \qquad \frac{\Gamma \vdash\ S1 :\ list(\tau)}{\Gamma \vdash\ \#S1 : \tau}$$

$$\frac{\Gamma \vdash\ S1 :\ \tau(\tau\ not\ list)}{\Gamma \vdash\ \#S1 : \tau}$$

$$\frac{\Gamma \vdash\ S1 :\ \tau_1 \quad \Gamma \vdash\ S2 :\ \tau_2 \quad \tau_1 = tau_2 = Int}{\Gamma \vdash\ S1 @ S2 : Int} \qquad \frac{\Gamma \vdash\ S1 :\ \tau_1 \quad \Gamma \vdash\ S2 :\ \tau_2 \quad \tau_1 = tau_2 = String}{\Gamma \vdash\ S1 @ S2 : String}$$

$$\frac{\Gamma[x \leftarrow \tau_1] \vdash\ S1 :\ \tau_2}{\Gamma \vdash lambda\ x : \tau_1 . S1 : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash\ S1 :\ \tau_1}{\Gamma \vdash\ IsNil\ S1 : \tau_1}$$

$$\frac{\Gamma \vdash\ S1 :\ Int}{\Gamma \vdash\ Nil\ S1 : Int} \qquad \frac{\Gamma \vdash S1 :\ Int \quad \Gamma \vdash S1 :\ \tau \quad \Gamma \vdash S1 :\ \tau}{\Gamma \vdash if\ S1\ then\ S2\ else\ S3 :\ \tau}$$

*Function is not needed because it is parsed as a lambda

2 Prove Preservation and Progress

Proof preservation of the type system with respect to the operational semantics.

For the base case:

$$\frac{Int\ i}{\Gamma \vdash i :\ Int} \qquad \frac{String\ i}{\Gamma \vdash i :\ String} \qquad \frac{Variable\ i}{\Gamma \vdash i :\ Variable}$$

$\alpha(i)$ follows from the given hypothesis for each respective base case.

For the case:

$$\frac{E \vdash S1 : \ e_1 \quad E \vdash S2 : \ e_2}{E \vdash S1 + S2 : e_1 + e_2} \qquad \frac{\Gamma \vdash S1 : \ Int \quad \Gamma \vdash S2 : \ Int}{\Gamma \vdash S1 + S2 : Int}$$

By the inductive hypothesis we have that S1 and S2 are both of type Int; since the product of two Int is also an Int, $\alpha(S1 + S2) = Int$. We argue similarly for the other cases, and thus prove that the type system is preserved.

Proof progress of the type system with respect to the operational semantics.
For the base case:

$$\frac{Int\ i}{E \vdash i : \ i} \qquad \frac{Int\ i}{\Gamma \vdash i : \ Int} \qquad \frac{String\ i}{E \vdash i : \ i} \qquad \frac{String\ i}{\Gamma \vdash i : \ String} \qquad \frac{Variable\ i}{E \vdash i : \ i} \qquad \frac{Variable\ i}{\Gamma \vdash i : \ Variable}$$

If i types as the respective type in the base case, then the semantic rule will apply for each one.
For the case:

$$\frac{E \vdash S1 : \ c_1 \quad E \vdash S2 : \ c_2}{E \vdash S1 + S2 : c_1 + c_2} \qquad \frac{\Gamma \vdash S1 : \ Int \quad \Gamma \vdash S2 : \ Int}{\Gamma \vdash S1 + S2 : Int}$$

By inductive hypothesis we know that S1, S2 will not get stuck. From our preservation proof, we know that the Int typed S1, S2, will evaluate to type Int. Therefore the operation semantic rule will apply, since this is the requirement of its hypotheses.
We then argue similarly for the other cases, and thus prove progress for the type system.

3 Algorithm and Inference Rules

The recursion was the algorithm of choice for this type system because it was deemed the best fit for implementation. The algorithm would read through an entire expression and check whether or not the expression was able to be typed. If not, the program would call itself with a more specific expression. This pattern would then continue until the expression was finally typed. Once typed, the type would be returned and the recursion would continue checking types as they were returned until the final expression was checked. If all checks were passed, then the program was successfully typed. If at any point there was a type mismatch, the program would automatically exit with a typing error statement.

Type Inference Rules

$$\frac{Int\ i}{\Gamma \vdash \ i : \alpha_1} \qquad \frac{\Gamma \vdash \ S1 : \ \alpha_1 \quad \Gamma \vdash \ S2 : \ \alpha_2 \quad \alpha_1 = \alpha_2 = Int}{\Gamma \vdash \ S1 \& S2 : \alpha_1} \qquad \frac{\Gamma \vdash \ S1 : \ \alpha_1 \quad \Gamma \vdash \ S2 : \ \alpha_2 \quad \alpha_1 = \alpha_2 = Int}{\Gamma \vdash \ S1 - S2 : \alpha_1} \qquad \frac{\Gamma \vdash \ S1 : \ \alpha_1 \quad \Gamma \vdash \ S2 : \ \alpha_2 \quad \alpha_1 = \alpha_2 = Int}{\Gamma \vdash \ S1/S2 : \alpha_1}$$

$$\frac{String\ s}{\Gamma \vdash \ s : \alpha_1}$$

$$\frac{Variable\ i}{\Gamma \vdash \ i : \alpha_1}$$

$$\frac{\Gamma \vdash \ S1 : \ \alpha_1 \quad \Gamma \vdash \ S2 : \ \alpha_2 \quad \alpha_1 = \alpha_2 = Int}{\Gamma \vdash \ S1 + S2 : \alpha_1} \qquad \frac{\Gamma \vdash \ S1 : \ \alpha_1 \quad \Gamma \vdash \ S2 : \ \alpha_2 \quad \alpha_1 = \alpha_2 = Int}{\Gamma \vdash \ S1 | S2 : \alpha_1} \qquad \frac{\Gamma \vdash \ S1 : \ \alpha_1 \quad \Gamma \vdash \ S2 : \ \alpha_2 \quad \alpha_1 = \alpha_2 = Int}{\Gamma \vdash \ S1 * S2 : \alpha_1} \qquad \frac{\Gamma \vdash \ S1 : \ \alpha_1 \quad \Gamma \vdash \ S2 : \ \alpha_2 \quad \alpha_1 = \alpha_2 = Int}{\Gamma \vdash \ S1 = S2 : \alpha_1}$$

$$\frac{\Gamma \vdash\ S1 :\ \alpha_1 \quad \Gamma \vdash\ S2 :\ \alpha_2 \quad \alpha_1 = \alpha_2 = Int}{\Gamma \vdash\ S1 <> S2 : \alpha_1}$$

$$\frac{\Gamma \vdash\ S1 :\ \alpha_1 \quad \Gamma \vdash\ S2 :\ \alpha_2 \quad \alpha_1 = \alpha_2 = Int}{\Gamma \vdash\ S1 < S2 : \alpha_1}$$

$$\frac{\Gamma \vdash\ S1 :\ \alpha_1 \quad \Gamma \vdash\ S2 :\ \alpha_2 \quad \alpha_1 = \alpha_2 = Int}{\Gamma \vdash\ S1 <= S2 : \alpha_1}$$

$$\frac{\Gamma \vdash\ S1 :\ \alpha_1 \quad \Gamma \vdash\ S2 :\ \alpha_2 \quad \alpha_1 = \alpha_2 = Int}{\Gamma \vdash\ S1 > S2 : \alpha_1}$$

$$\frac{\Gamma \vdash\ S1 :\ \alpha_1 \quad \Gamma \vdash\ S2 :\ \alpha_2 \quad \alpha_1 = \alpha_2 = Int}{\Gamma \vdash\ S1 >= S2 : \alpha_1}$$

$$\frac{\Gamma \vdash\ S1 :\ \alpha_1 \quad \alpha = \alpha_1 \quad \Gamma[id \leftarrow \alpha] \vdash\ S2 : \alpha_2}{\Gamma \vdash\ let\ id = S1\ in\ S2 :\ \alpha_2}$$

$$\frac{\Gamma \vdash\ S1 :\ \alpha_1 \quad \alpha_1 = list(\alpha_2)}{\Gamma \vdash\ !S1 : \alpha_2}$$

$$\frac{\Gamma \vdash\ S1 :\ \alpha_1 \quad \alpha_1 \neq list(\alpha_2)}{\Gamma \vdash\ !S1 : \alpha_2}$$

$$\frac{\Gamma \vdash\ S1 :\ \alpha_1 \quad \alpha_1 = list(\alpha_2)}{\Gamma \vdash\ \#S1 : \alpha_2}$$

$$\frac{\Gamma \vdash\ S1 :\ \alpha_1 \quad \alpha_1 \neq list(\alpha_2)}{\Gamma \vdash\ \#S1 : \alpha_2}$$

$$\frac{\Gamma \vdash\ S1 :\ \alpha_1 \quad \Gamma \vdash\ S2 :\ \alpha_2 \quad \alpha_1 = \alpha_2 = Int}{\Gamma \vdash\ S1@S2 : \alpha_1}$$

$$\frac{\Gamma \vdash\ S1 :\ \alpha_1 \quad \Gamma \vdash\ S2 :\ \alpha_2 \quad \alpha_1 = \alpha_2 = String}{\Gamma \vdash\ S1@S2 : \alpha_1}$$

$$\frac{\Gamma[x \leftarrow \alpha_1] \vdash\ S1 :\ \alpha_2}{\Gamma \vdash lambda\ x : \alpha_1.S1 : \alpha_1 \rightarrow \alpha_2}$$

$$\frac{\Gamma \vdash\ S1 :\ \alpha_1}{\Gamma \vdash\ IsNil\ S1 : \alpha_1}$$

$$\frac{\Gamma \vdash\ S1 :\ \alpha_1 \quad \alpha_1 = Int}{\Gamma \vdash\ Nil\ S1 : \alpha_1}$$

$$\frac{\Gamma \vdash S1 :\ \alpha_1 \quad \Gamma \vdash S1 :\ \alpha_2 \quad \Gamma \vdash S1 :\ \alpha_3 \quad \alpha_1 = Int \quad \alpha_2 = \alpha_3}{\Gamma \vdash if\ S1\ then\ S2\ else\ S3 :\ \alpha_3}$$

4 Example Programs that Fail

1. if 1 then 1 + "duck" else 1 + 2
2. 3 + "nothing"
3. let f = lambda x, y. x+y in
   let k = (f 2) in
   (k "thing")
4. let x = lambda y,z. 1+y+z in
   let k = (x "this") in
   let u = (k that) in u

5 Trade-offs

A few trade-off were encountered when design choices were made in this implementation. Having only six types meant that the type system could not be as detailed as should be in a fully usable programming language. The types were able to fully cover the language, but to add more would have caused more trouble than good. Another trade-off that we encountered was the messiness of returning the success message once it was realized that the type inference was successful. A success statement had to be created for each AST Node, which because slightly cumbersome to keep track of. Another problem with recursion was the lack of elegance in our coding. Many if-else blocks were required to filter out different types during recursion. Finally, the most obvious trade-off with this implementation of L is the efficiency issue. Efficiency was not a top priority in the creation of this system, and as a result, the implementation is not the most efficient code available. But these issues are not noticeable at the level that this implementation is being used.