CSCI 312: Principles of Programming Languages
L Type Inference Manual

Robert Johns

April 7, 2012

1. Introduction

The type system and algorithm that follows provides type inference for programs in the L programming language. As shall be demonstrated and proven, the type system is complete, prevents all run-time errors and allows all legal L programs to be interpreted.

2. Typing Rules, Proof of Preservation And Progress

Types: $Int, String, ListInt, ListString, Nil$

Base Rules:

$$\frac{\Gamma \vdash x : Int}{\Gamma \vdash x : Int} \quad \frac{\Gamma \vdash x : String}{\Gamma \vdash x : String} \quad \frac{\Gamma \vdash x : Nil}{\Gamma \vdash x : Nil} \quad \frac{\Gamma \vdash x : ListInt}{\Gamma \vdash x : ListInt} \quad \frac{\Gamma \vdash x : ListString}{\Gamma \vdash x : ListString}$$

With all of these rules, it follows from the assumption that the input is of the given type that the type system result will over-approximate the type of the concrete value, so preservation holds.

Since these are the only primitive types in L, no run-time errors will occur with these rules, so progress holds.

Binary Operations:

$$\frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \Gamma \vdash S_2 : \alpha_2 \\ \alpha_1 = \alpha_2 = Int\end{array}}{\Gamma \vdash S_1 \texttt{+} S_2 : \alpha_1} \quad \frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \Gamma \vdash S_2 : \alpha_2 \\ \alpha_1 = \alpha_2 = Int\end{array}}{\Gamma \vdash S_1 \texttt{-} S_2 : \alpha_1} \quad \frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \Gamma \vdash S_2 : \alpha_2 \\ \alpha_1 = \alpha_2 = Int\end{array}}{\Gamma \vdash S_1 \texttt{*} S_2 : \alpha_1} \quad \frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \Gamma \vdash S_2 : \alpha_2 \\ \alpha_1 = \alpha_2 = Int\end{array}}{\Gamma \vdash S_1 \texttt{/} S_2 : \alpha_1} \quad \frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \Gamma \vdash S_2 : \alpha_2 \\ \alpha_1 = \alpha_2 = String\end{array}}{\Gamma \vdash S_1 \texttt{+} S_2 : \alpha_1}$$

$$\frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \Gamma \vdash S_2 : \alpha_2 \\ \alpha_1 = \alpha_2 = Int\end{array}}{\Gamma \vdash S_1 \texttt{ \& } S_2 : \alpha_1} \quad \frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \Gamma \vdash S_2 : \alpha_2 \\ \alpha_1 = \alpha_2 = Int\end{array}}{\Gamma \vdash S_1 \texttt{ | } S_2 : \alpha_1} \quad \frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \Gamma \vdash S_2 : \alpha_2 \\ \alpha_1 = \alpha_2 = Int\end{array}}{\Gamma \vdash S_1 \texttt{ < } S_2 : \alpha_1} \quad \frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \Gamma \vdash S_2 : \alpha_2 \\ \alpha_1 = \alpha_2 = Int\end{array}}{\Gamma \vdash S_1 \texttt{ > } S_2 : \alpha_1} \quad \frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \Gamma \vdash S_2 : \alpha_2 \\ \alpha_1 = \alpha_2 = Int\end{array}}{\Gamma \vdash S_1 \leq S_2 : \alpha_1} \quad \frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \Gamma \vdash S_2 : \alpha_2 \\ \alpha_1 = \alpha_2 = Int\end{array}}{\Gamma \vdash S_1 \geq S_2 : \alpha_1}$$

$$\frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \Gamma \vdash S_2 : \alpha_2 \\ \alpha_1 = \alpha_2 \\ \alpha_3 = Int\end{array}}{\Gamma \vdash S_1 \texttt{ = } S_2 : \alpha_3} \quad \frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \Gamma \vdash S_2 : \alpha_2 \\ \alpha_1 = \alpha_2 \\ \alpha_3 = Int\end{array}}{\Gamma \vdash S_1 \texttt{ <> } S_2 : \alpha_3} \quad \frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \Gamma \vdash S_2 : \alpha_2 \\ \alpha_1 = \alpha_2 = Int \\ \alpha_3 = ListInt\end{array}}{\Gamma \vdash S_1 \texttt{ @ } S_2 : \alpha_3} \quad \frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \Gamma \vdash S_2 : \alpha_2 \\ \alpha_1 = \alpha_2 = String \\ \alpha_3 = ListString\end{array}}{\Gamma \vdash S_1 \texttt{ @ } S_2 : \alpha_3}$$

For each integer binary operation, by the inductive hypothesis we know that the abstraction of both integers are integers, and so the abstraction of the sum, product, etc. must also be an integer. The same logic holds for string concatenation and the `=` or `<>` operators on strings. For list creation, it follows from the inductive hypothesis (that each object operated on is either an int or a string) that the list created by the `@` operator on those objects will be of type ListInt or ListString respectively. So preservation holds.

The constraints on each rule ensure that each binary operation can only be performed on the proper type, so, inductively assuming validity of the base cases, no run-time errors will occur and progress will hold for all binary operations.

Unary Operations:

$$\frac{\Gamma \vdash S_1 : \alpha_1 \quad \alpha_1 = ListInt}{\Gamma \vdash \texttt{!}S_1 : Int} \qquad \frac{\Gamma \vdash S_1 : \alpha_1 \quad \alpha_1 = ListString}{\Gamma \vdash \texttt{!}S_1 : String} \qquad \frac{\Gamma \vdash S_1 : \alpha_1 \quad \alpha_1 \neq ListInt \neq ListString}{\Gamma \vdash \texttt{!}S_1 : \alpha_1}$$

$$\frac{\Gamma \vdash S_1 : \alpha_1 \quad \alpha_1 = ListInt}{\Gamma \vdash \texttt{\#}S_1 : Int} \qquad \frac{\Gamma \vdash S_1 : \alpha_1 \quad \alpha_1 = ListString}{\Gamma \vdash \texttt{\#}S_1 : String} \qquad \frac{\Gamma \vdash S_1 : \alpha_1 \quad \alpha_1 \neq ListInt \neq ListString}{\Gamma \vdash \texttt{\#}S_1 : \alpha_1}$$

$$\frac{\Gamma \vdash S_1 : \alpha_1 \quad \alpha_2 = Int}{\Gamma \vdash \texttt{isNil}\ S_1 : \alpha_2} \qquad \frac{\alpha_1 = Int}{\Gamma \vdash \texttt{print}\ S_1 : \alpha_1} \qquad \frac{\alpha_1 = Int}{\Gamma \vdash \texttt{readInt}\ S_1 : \alpha_1} \qquad \frac{\alpha_1 = String}{\Gamma \vdash \texttt{readString}\ S_1 : \alpha_1}$$

If follows from the inductive hypothesis for each unary operator that each rule over-approximates the type of the concrete value, so preservation holds.

It follows from the constraints on each rule that each unary operation can be performed only on the specified type, so inductively assuming the validity of the base cases, no run-time errors can occur and progress will hold for all unary operations.

Conditionals:

$$\frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \Gamma \vdash e_1 : \alpha_2 \\ \Gamma \vdash e_2 : \alpha_3 \\ \alpha_1 = Int \\ \alpha_2 = \alpha_3\end{array}}{\Gamma \vdash \texttt{if}\ S_1\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \alpha_2}$$

By the inductive hypothesis and the constraint that $\alpha_2 = \alpha_3$, it follows that the resultant type will be an over-approximation of the concrete value, so preservation holds.

It follows from the constraints that $\alpha_2 = \alpha_3$ and that `p` must be an integer that no run-time errors can ever occur, since no incongruence can arise from the returned types, so progress holds.

Let & Identifiers:

$$\frac{\begin{array}{c}\Gamma \vdash S_1 : \alpha_1 \\ \alpha_1 = \alpha_2 \\ \Gamma[id \leftarrow \alpha] \vdash S_2 : \alpha_2\end{array}}{\Gamma \vdash \texttt{let}\ id = S_1\ \texttt{in}\ S_2 : \alpha_2} \qquad \frac{\Gamma \vdash \Gamma(id) = \alpha}{\Gamma \vdash id : \alpha}$$

The `let` rule will be an over-approximation of its concrete type because the inductive hypothesis ensures that S_2 must return either a valid type or an error. The *id* rule will be an over-approximation

of its concrete type because the nature of the inductive hypothesis ensures it must be of a valid type, so preservation holds.

The simple *id* rule will never incur a run-time error because of the precondition that the identifier have a designated type. The `let` rule will never incur a run-time error because of the precondition that S_2 with a mapped type for *id* return a valid type. So progress holds.

Lambda Statements:

$$\frac{\Gamma[x \leftarrow \alpha_1] \vdash S_1 : \alpha_2}{\Gamma \vdash \texttt{lambda}\ x : \alpha_1.\ S_1 : \alpha_1 \rightarrow \alpha_2}$$

It follows easily from the base case that $\alpha_1 \rightarrow \alpha_2$ will over-approximate the concrete value. So preservation holds assuming the base cases.

A run-time error is impossible with the lambda statement because of the required condition that the input will be of type α_1 and that the output will be of type α_2. So progress holds.

Expression Lists:

$$\frac{\Gamma \vdash S_1 : \alpha_1 \rightarrow \alpha_2 \quad \Gamma \vdash S_2 : \alpha_1}{\Gamma \vdash (S_1\ S_2) : \alpha_2}$$

It follows easily from the base cases, and from the lambda rules that α_2, the resultant type of the function, will be an over-approximate representation of the concrete type. So preservation holds.

Because of the assurance that the function is of the proper function type, and that S_1 is of the proper type, no run-time error can ever occur. So progress holds.

Since `fun` statements are parsed as `let` bindings plus `lambda` abstractions, additional typing rules for function definitions need not be specified.

3. **Explanation of Inference Algorithm**

The inference algorithm, implemented in C++, takes a parse-tree expression as input and passes it to an evaluator method. The evaluator method contains a suite of if/else statements that perform type inference on the different types of expressions in L. Returned in each if/else statement is a type variable that represents the type returned by the program. For example, if the program was simply:

```
1
```

the interpreter would return ConstantType(Int). Other type variables include String, ListInt, ListString, Nil, VariableType(x) and FunctionType(α_1, α_2). If the program contains a typing error, the evaluate function returns a Failure type along with an explanation. That is, if the program were

```
1 + "duck"
```

the interpreter would return ConstantType(Failure) and print a message along the lines of:

```
Typing Error:  Binary operations must be applied to objects of the same type
```

Each time a type is correctly identified, a message is printed indicating proper type inference. The algorithm itself is recursive, calling itself in case of nested expressions to evaluate each expression to a type variable.

4. **Example Programs With Failing Results**

   An example of a failure to perform type inference correctly unfortunately occurs in a program upon which the inferrer is graded:

   ```
   let f = lambda x.if x = 0 then 1 else x * (g x)
   in
   let g = lambda x.  (f (x - 1))
   in
   (f 6)
   ```

   returns a type variable of ConstantType(Failure), when it should return ConstantType(Int).

   Similarly, a program such as:

   ```
   lambda x.  lambda y.  x + y 6
   ```

   returns a type variable of ConstantType(Failure), when it should return FunctionType(Int $\rightarrow \alpha$)

5. **Trade-Offs**

   As demonstrated above, the type inferrer fails on certain let and lambda programs, not returning a proper function type in many circumstances. This is a side-effect of an improper use of union-find, as I elected to use a symbol-table algorithm rather than the union-find algorithm.