Introduction

- We will talk about type inference in L
- This will be your 4th (and last) programming project
- But the in a very different style then the first three

L and Type Inference

- First of all, observe that L is a superset of the toy languages we used for type inference in lecture
- But all the basic constructs are there, so everything we looked at in lecture applies right away.
- Therefore, let’s briefly focus on the aspects of L that are new.

Conditionals

- One construct we have not talked a lot about recently is conditionals:
  - Recall: Conditional in L
    \[ \text{if } p \text{ then } e_1 \text{ else } e_2 \]
  - Question: What is the typing rule for a conditional?
- First, observe that the conditional is well-typed only if \( p \) evaluated to an integer
  - and \( e_1 \) and \( e_2 \) evaluate to the same type

Conditionals Cont.

- Therefore, we have the following typing rule:
  \[
  \Gamma \vdash p : \text{Int} \\
  \Gamma \vdash e_1 : \tau \\
  \Gamma \vdash e_2 : \tau \\
  \Gamma \vdash \text{if } p \text{ then } e_1 \text{ else } e_2 : \tau
  \]
  - \text{Question: What is an L program that has no run-time error but cannot be typed under this rule?}
  - \text{Answer: let } x = 3 < (\text{if } 1 \text{ then } 1 \text{ else } "duck") \text{ in } x

Conditionals and Type Inference

- Now, let’s rewrite this rule to generate type constraints we can solve during type inference:
  \[
  \Gamma \vdash p : \text{Int} \\
  \Gamma \vdash e_1 : \tau \\
  \Gamma \vdash e_2 : \tau \\
  \Gamma \vdash \text{if } p \text{ then } e_1 \text{ else } e_2 : \tau
  \]
  - Here is typing rule for type inference:
  \[
  \Gamma \vdash p : \alpha_1 \\
  \Gamma \vdash e_1 : \alpha_2 \\
  \Gamma \vdash e_2 : \alpha_3 \\
  \alpha_1 = \text{Int} \\
  \alpha_2 = \alpha_3 \\
  \Gamma \vdash \text{if } p \text{ then } e_1 \text{ else } e_2 : \alpha_3
  \]
Lists

- Big feature in L that we have not yet discussed: Lists
- Lists are interesting because they introduce a new kind of type
- However, lists raise many interesting issues.
- First, consider the following expression in L: 10*"duck".
- This evaluates to the list [1, "duck"]
- But what is the type of this list?

Lists and Types

- First Idea: Have two list types, listInt and listString
- This way, we really only introduced two new type constants
- Now, what is the type of [3, 4]? listInt
- Benefit: Really easy. We did not have to do anything new

Lists and Types

- Question: Is just having listString and listInt expressive enough?
- This has two limitations:
  1. We cannot have a list that mixes integers and strings
  2. We cannot have, for example, a list of list of integers
- These issues are known as list polymorphism

Lists as Polymorphic Types

- Solution: Use the same idea we used for function types
- We will introduce a new kind of type called list type
- This list type will be parametric on the element type
- Notation: We will write a list with element type $\alpha$ as $\text{list}(\alpha)$

Lists as Polymorphic Types Example

- For example, what is the type of [[1, 2], [2, 3]]?
- Type is $\text{list(list(Int))}$
- Observe that there is a fundamental similarity to function types
- Recall from last time: $\alpha_1 \rightarrow \alpha_2$ is just in-fix notation for $\text{function}(\alpha_1, \alpha_2)$
- List is really just another kind of function type with only one argument
Type Checking with Polymorphic Lists

- For example, how can we type check the !e1?
- Here is the typing rule:
  \[ \Gamma \vdash e_1 : \text{list}(\tau) \]
  \[ \Gamma \vdash !e_1 : \tau \]
- But what if \( e_1 \) does not evaluate to a list?
- Typing rule in this case:
  \[ \Gamma \vdash e_1 : \tau \text{not list} \]
  \[ \Gamma \vdash !e_1 : \tau \]

Type Inference with Polymorphic Lists

- Consider again the typing rules for head of list:
  \[ \Gamma \vdash e_1 : \text{list}(\tau) \]
  \[ \Gamma \vdash !e_1 : \tau \]
  \[ \Gamma \vdash e_1 : \tau \text{not list} \]
  \[ \Gamma \vdash !e_1 : \tau \]
- Observe: Rules are not structural, i.e., more than one rule can apply for a given syntactic expression
- This is no problem in type checking, as long as the conditions under which each rule applies are disjoint

How to Fix Type Inference with Lists

- First fix: Restrict language. In this case, disallow \( !e \) on anything but lists
  - This makes the typing rule structural for head of list
    - But now consider the L expression \( e1@e2 \)
    - Problem here: If \( e2 \) is of type \( \text{list}(X) \), then \( e1@e2 \) is of what type? \( \text{list}(\text{list}(X)) \)?
  - Conclusion: We cannot support type inference on lists with no structure

- First, we can only support "flat" lists in basic type inference.
  - Flat list: \([1, [2, 3]]\)
  - Not flat list: \([1, 2], 3\)
- Second, we need to restrict the language such that only one typing rule applies at the same time or use more sophisticated inference algorithm
  - However, even if we only allow flat lists, difficulties remain with \( e1@e2 \)
How to Fix Type Inference with Lists Cont.

- To see the problem, consider the L expression `let x = 2@3` in `1@x`.
- Here, in the expression `2@3` both 2 and 3 are of type `Int` and the result is of type `list(int)`.
- In the expression `1@x`, 1 is of type `Int` and `x` is of type `list(Int)` and the result is of type `list(int)`.
- To solve this, we either need to handle non-structural type inference rules or use two different operators for forming a list out of an atom and a list or two atoms.

On to the next project

- Now, let’s move on to what exactly you will be doing for the last programming project.
- Warning: This project is more independent and more difficult than the previous programming assignments.
- This means that it is your responsibility to start early and checkpoint your progress.

Project 4

- The first goal of this programming project is to add type inference to L.
- But this time, I will not give you a long manual/handout telling you exactly what kind of type inference you are supposed to implement.
- Instead, it will be your job to design a type inference system for L.

Big Picture

- I will provide a set of example programs that your type system must be able to successfully infer types for.
- You will design a type system that is expressive enough to perform successful type inference on at least these programs.
- Your type system must also be sound, i.e., it must prevent all run-time errors.

Must Requirements

- Specifically, you are required to do the following:
  - Design a type system that is sound, expressive enough to type the example programs and that can support type inference.
  - Prove preservation and progress of this type system, assuming type annotations.
  - Modify your type checking rules for type inference.
  - Implement a type inference algorithm.
  - Show that you can infer types for all examples.
  - Show some examples of error-containing and error-free programs on which you cannot do a successful type inference.
  - Discuss in detail your design choices and trade-offs when designing your type system.

Requirements Type Inference

- You have to implement polymorphic type inference for function types.
- But you don’t have to implement polymorphic lists.
- This means, you can just use type constants `IntList` and `StringList`.
- But, I will give extra credit if you can design a working polymorphic type system for lists in L.
- And definitely do not start on this until you have your basic algorithm designed and working!
To Complete

1. A report written in Latex:
   ▶ You will start with a high-level introduction to your type system and then present typing rules for all constructs in L and prove progress and preservation for your type system
   ▶ Give your type inference rules and algorithm for solving type constraints using union-find
   ▶ Explain how your algorithm performs successful type inference on the examples
   ▶ Show some examples of programs (both with real errors and no errors) on which your type inference fails
   ▶ Conclude with a discussion of the trade-offs you faced and why you made the choices you did

2. An implementation of your type inference algorithm in C++

The Report

▶ I expect your report to be lucid, well-written and complete
▶ You are welcome to use any references you can find, but you must cite them.
▶ Everything you write must have a point: I expect readable, clear and concise writing
▶ Why does this project include a report? You will have to report on your work all the time, no matter if you end up in academia or industry, and this is good practice!

The Implementation

▶ This time, please provide a clean implementation with comments
▶ Since everyone will implement a slightly different strategy, this will help us evaluate your project
▶ Your grade for the implementation component will take cleanliness and comments into account

Grading

▶ This project will count for 10% of your grade
▶ Half of your grade will be your written report, the other half your implementation
▶ Have fun!

Extra Credit Options

▶ Adding type annotations to L
▶ Changing/extending language semantics
▶ Improving performance
▶ Anything else cool you are interested in

Getting Started

▶ I will post starter code with a union-find library for you to use on the web site
▶ I also posted the set of examples you must handle
▶ But no handout, since everything else is up to you
Time Line:

- November 9th: 1-page project plan due. This includes:
  - All group member names
  - A list of everything you want to do for this project and a sketch on your plan.
  - In LaTeX
- One-on-one group meetings to be scheduled
- December 2nd: Project due