First: Your Project

- Today is the start of your course project
- Goal: Take what we studied and apply it to a project you design yourself
- This is a team project: Teams must be between 3 and 5 students

Possible Topics

- Your goal is to add at least one major feature to the L language
- Some possible examples:
  - Adding type inference to L
  - Speeding up the L interpreter
  - Adding major language features to L
  - Type inference with novel error reporting
  - ...
- Your creativity is the limit

Deliverables & Time line

- Today: Start of project, form teams
- Nov. 13th 12:30pm: Email me a one page proposal for your project as pdf clearly describing what you want to do and list your team members
- Will receive feedback from proposal
- Dec. 11th 12:30pm: Project due. No late days.

Final Deliverables

- Report written in LaTeX (at least 15 pages) describing clearly what problem you are solving, what choices you made, challenges encountered and your results.
- All your source code in a tar.gz file compiling on Ubuntu
- You will be graded on size of chosen challenge, your solution and your written report
- Since every project is unique, you will get lots of feedback throughout
- If you are passionate about a PL project not related to L, or want to tackle something especially large with more people, etc: Ask!
- Any questions?

Introduction

- Recall for last time: We are inferring types
- Big idea: Replace all concrete type assumptions with type variables
- Collect constraints on these type variables
- Find most general solution for these constraints to deduce types
Quick Refresher

- Let's quickly look again at one example:

\[
\text{let } f = \lambda x.(f \ x) \text{ in } f
\]

- Type derivation:

\[
\begin{align*}
\Gamma[f ← a_1][x ← a_2] & \vdash f : a_1 \\
\Gamma[f ← a_1][x ← a_2] & \vdash x : a_2 \\
\Gamma[f ← a_1][x ← a_2] & \vdash (f \ x) : a_2 \\
\Gamma[f ← a_1] & \vdash \lambda x.(f \ x) : a_1 \\
\Gamma & \vdash \text{let } f = \lambda x.(f \ x) \text{ in } f : a_1
\end{align*}
\]

- Final Type: \(a_1 \rightarrow a_3\)

This yielded constraint system

\[
\begin{align*}
a_1 & \rightarrow a_3 \\
a_3 & \rightarrow a_1
\end{align*}
\]

More Efficient Type Inference

- Big Idea: Maintain equivalence classes of types directly

- Equivalence Class: Set of types that must be equal

- Specifically, if we process constraint of the form \(X = Y\), we know that \(X\) and \(Y\) are equal

- In this case, we want to union the equivalence classes of \(X\) and \(Y\)

- Also, if \(X\) and \(Y\) are function types of the form \(X_1 \rightarrow X_2\) and \(Y_1 \rightarrow Y_2\), we also want to union \(X_1\) and \(Y_1\) as well as \(X_2\) and \(Y_2\)

Union-Find

- To maintain equivalence classes directly, we will use the union-find algorithm

- Each set of types is called an equivalence class

- Each set has one element as its representative

- For type inference: If an equivalence contains a type constant or a function type, we will always use this type as the representative.

Union-Find Cont.

- In Union-Find, we have only two operations on equivalence classes:

  1. \(\text{Union}(s, t)\): This unions the equivalence classes of \(s\) and \(t\) into one equivalence class

  2. \(\text{Find}(s)\): This returns the representative of the equivalence class of which \(s\) is part of

- Example: Assume following two equivalence classes (representatives in red): \(\{\text{int}, \alpha\}, \{\beta \rightarrow \gamma, \text{int}\}\)

- Example: \(\text{Union}(\text{int}, \beta \rightarrow \gamma)\) results in new equivalence class \(\{\text{int}, \alpha, \beta \rightarrow \gamma\}\)

- Example: \(\text{Find}(\alpha) = \text{int}\)
Union-Find Representation

- We will represent equivalence classes as DAGs.
- Example: \{\beta \rightarrow \gamma, \alpha\}
- Conceptually, union will join the dotted areas of two equivalence classes
- And find will return the (red) representative in this class

Finding a Solution from the Union-Find DAG

- Question: Is this a possible solution for the type constraints?
- No! If a function type and a constant type ever end up in the same equivalence class, we know that the constraint system has no solution
- We also know constraint system has no solution if \texttt{Int} and \texttt{String} end up in the same EQ

Finding a Solution from the Union-Find DAG

- Example:
- How do we find solution for \alpha?
- \texttt{find(\alpha) = \beta \rightarrow \gamma}
- What about \beta?
- Every item is in its own EQ, therefore \texttt{find(\beta) = \beta}

Using Union-Find for solving Type Inference Constraints

- Initially, all type variables, functions and type constants are in their own equivalence class
- We then apply the following function to each equality in our type constraint:

```c
bool unify(m, n) {
    s = find(m); t = find(n);
    if(s == t) return true;
    if(s == s1 \rightarrow s2 && t == t1 \rightarrow t2) {
        union(s, t);
        return unify(s1, t1) && unify(s2, t2);
    }
    if(is_variable(s) || is_variable(t)) {
        union(s, t); return true;
    }
    return false; //No solution to type constraints
}
```

Union-Find Representation Cont.

- Consider the following EQs:
- And now consider \texttt{union(\beta \rightarrow \gamma, \texttt{Int})}

Union-Find Representation

- We will represent equivalence classes as DAGs.
- Example: \{\beta \rightarrow \gamma, \alpha\}
- Conceptually, union will join the dotted areas of two equivalence classes
- And find will return the (red) representative in this class

Finding a Solution from the Union-Find DAG

- Assuming we end up with a consistent Union-find DAG, we can read the most general solution right of!
- For each type variable \(v\), simply return \texttt{find}(v)
- In other words, the representative of each equivalence class is the most general solution
- Question: Why do we always pick function types or type constants as representatives?
- Question: What happens if a function type and a type constant are in the same equivalence class?
Example

Consider the following system of type constraints:
\[ \alpha \rightarrow \text{Int} = \beta \]
\[ \gamma \rightarrow \text{Int} = \beta \]
\[ \gamma = \text{String} \]

Example Cont

\[ \alpha \rightarrow \text{Int} = \beta \]
\[ \gamma \rightarrow \text{Int} = \beta \]
\[ \gamma = \text{String} \]

Solution for \( \alpha \): find(\( \alpha \)) = \text{String}

Solution for \( \beta \): find(\( \beta \)) = \text{String} \rightarrow \text{Int}

Solution for \( \gamma \): find(\( \gamma \)) = \text{String}

Example 2

Consider the following system of type constraints:
\[ \alpha = \text{Int} \rightarrow \text{Int} \]
\[ \alpha = \text{String} \]

Conflict: Unify returns false when trying to unify \( \text{Int} \rightarrow \text{Int} \) and \( \text{String} \)

Conclusion: This system of type constraints is unsatisfiable

Union-Find

With this new approach, we can now only process each equality once.

However, for this to be efficient, union/find must be efficient.

Key result from algorithms: It is possible to build a data structure for union-find that can find a solution to our sets of type constraints in approximately linear time.

You can learn about this data structure in Advance Algorithms or Isil's class on automated logical reasoning

But for our purposes, we will just use this data structure

Type Inference

If we use Union-Find, we can make type inference practical on real programs

This style of polymorphic type inference we studied is known as Hindley-Milner type inference

Type inference is at the core of languages such as OCAML and Haskell

Type inference is increasingly moving to mainstream languages

- New C++11 standard
- Java 7

Type Inference and Errors

We saw that we can detect all errors easily when doing type inference

Specifically, every error resulted from unifying two equivalence classes that could not be unified.

Example: Trying to unify \( \text{String} \) and \( \alpha \rightarrow \text{Int} \)

But how do we report this error to programmers?
Error Reporting

- Consider again the example: \texttt{String} and \(\alpha \rightarrow \text{Int}\).
- Option 1: Output message: \texttt{String} and \(\alpha \rightarrow \text{Int}\) cannot be unified.
- Is this helpful?
- Obvious problems:
  - Not associated with any source location
  - Understanding typing errors requires understanding type inference

Error Reporting Cont.

- Improvement used in practice: Associate expression/source location with type constraint.
- Message can now at least contain the program expressions that evaluate to \texttt{String} and \(\alpha \rightarrow \text{Int}\).
- But the actual error in your program may be arbitrarily far from these locations!
- Typical OCaml error: "At line 37: Expected expression of type ‘\(a \to a\)’ but found expression of type ‘\(a \to b\)’
- To fix this, you need to understand all the reasoning steps that happened during type inference
- Most likely, the problem did not originate at line 37!

Type Annotations

- Most common technique for mitigating these difficulties:
  Allow type annotations
- Type annotations allow you explicitly declare types even though the compiler can infer them automatically
- Idea: If you encounter a type error you do not understand, you give the type you expect to the expressions involved in this error and re-run the type checker
- You will now get a new type error in a different location
- You repeat this process until you fixed your type error

Type Annotations Drawbacks

- However, this approach still has substantial drawbacks:
  - You often need many annotations to find the source of type errors
  - You can only annotate successfully if you understand polymorphic type inference
  - You often end up with a program that is almost completely type annotated!

Type Inference in the Real World

- Despite these difficulties, there are many real languages that support full type inference.
- Examples: OCaml, Haskell, F#
- Slogan on Type Inference: The ease of dynamic typing with the speed and guarantees of a static type system
- This claim is true, but real problems with explaining typing errors to programmers
- Explaining typing errors better is also an active research area!

Type Inference in the Real World Cont.

- Alternative approach taken by more main-stream languages recently: local type inference
- In local type inference, types are only inferred within one function, but must be fully annotated at function boundaries.
- Goal: Make it easier for programmers to diagnose type errors (and make type inference tractable in the imperative setting)
Example of local type inference

- C++ supports some forms of local type inference.
- First Example: templates
- A STL pair is templatized over the type of the first and second element
- You declare a pair as: pair<int, string> p(3, "duck");
- However, if you call a function that takes a pair, the compiler will infer the template type for you in some cases:
  - Example: edit_pair(p) instead of edit_pair<pair<int, string>>(p)

The new C++11 standard supports much more expressive local type inference
- This is done using the auto keyword
- Example using iterator: vector<int> v;
  ... for(vector<int>::iterator it = v.begin(); it != v.end(); it++) ...
- Example using iterator with new auto keyword:
  vector<int> v;
  ... for(auto it = v.begin(); it != v.end(); it++) ...

Type Inference in C++

- The auto keyword really just says "do type inference on this expression and figure the type out"
- Very convenient, local feature that is also creeping into languages such as C# and Java
- You will see more of this in the future

Conclusion

- We saw how to use Union-Find to make type inference scalable
- This formulation is one of the classic and elegant results in programming languages, known as Hindley-Milner type inference
- Type inference is most likely coming to your favorite language in the near future, if it is not already there!