Review

- Features learned: functions, tuples, lists, let expressions, options, records, datatypes, case expressions, type synonyms, pattern matching, exceptions, type variables, higher-order and anonymous functions, infix operators, type constructors, currying, lexical scope, closures

- Today:
  - Type inference

Kinds of typing

- Static vs. dynamic:
  - **Static**: type checking done by analysis of program
    - Compiler/interpreter verifies that type errors cannot occur
    - e.g., C, C++, FF, Haskell, Java, SML
  - **Dynamic**: type checking done by run-time
    - Run-time detects type errors and reports them. Usually requires keeping extra tag information for each value in memory.
    - e.g., JavaScript, LISP, Matlab, PHP, Python, Ruby
    - In practice, this can be a spectrum
      - e.g., `instanceof` in Java: some checking done at compile time, rest of checking done at run time

Kinds of typing

- Strong vs. weak:
  - **Strong**: type of a value is independent of how it’s used
    - Can’t pass a `string` where an `int` expected, etc.
    - e.g., SML, Haskell, Python, Java, Ruby
  - **Weak**: type of value is dependent on how it’s used
    - If a `string` is used where an `int` expected, it gets converted automatically to an `int`
      - e.g., C, C++, Perl, Python
    - Again, this can become a spectrum
      - e.g., Java + `operator converts objects to strings`
    - Troll alert: strong vs. weak is debated a lot; probably not helpful to degenerate into such debates

Typing quadrant

<table>
<thead>
<tr>
<th></th>
<th>Weak</th>
<th>Strong</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>C, C++</td>
<td>SML, Java, Haskell</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Perl, Assembly</td>
<td>Ruby, Python, Scheme</td>
</tr>
</tbody>
</table>

Kinds of typing

- Manifest vs. implicit:
  - **Manifest**: type information supplied in source code
    - e.g., C, C++, Java
  - **Implicit**: type information not supplied in source code
    - Implementation 1: Dynamic typing
      - e.g., LISP, Python, Ruby, PHP
    - Implementation 2: Type inference
      - e.g., Haskell, SML
    - Tradeoff: ease of implementation vs. run-time performance
      - Again, in practice this becomes a spectrum
        - e.g., SML sometimes needs type annotations for records
        - e.g., no reasonable language requires you to write to provide the type of `5` in `x:int = 5`
Type inference

- Goal is to reconstruct types of expressions based on known types of some symbols that occur in expressions
  - Type checkers have to do some of this anyway
  - Difference between inference and checking is really a matter of degree
- Best known in functional languages
  - Especially useful in managing the types of higher-order functions
  - But starting to appear in mainstream languages
    - e.g., C++11:
      - `auto x = e;` declares variable `x`, initialized with expression `e`, and type of `x` is automatically inferred
- Invented by Robin Milner for SML (though other people also deserve credit; see the notes)

Is type inference hard?

- The algorithm used in ML is quite clever yet
  - relatively easy to implement and
  - relatively easy to explain
- Difficulty of doing type inference for any particular language is often hard to determine
  - Not particularly related to strong vs. weak or how permissive type system is
  - Type system that accepts everything is easy to infer types for
  - Type system that accepts nothing is easy to infer types for
  - Designing type inference for a particular language can be quite hard
    - Difficult to balance:
      - expressivity of type system
      - possibility of inferring all types without requiring annotations

HM type inference

- Overview:
  - Determine types of bindings in order
    - Use types of earlier binding to infer later
    - (which is why you can’t use later bindings in file)
  - For each binding, solve constraints to determine type
    - e.g., if inferencer sees `x+1`, concludes `x` must have type `int`
    - First step is to assign preliminary types to all subexpressions in binding
    - Second step is to generate constraints from those preliminary types
    - Third step is to solve system of constraints for type of binding
- Let’s do examples before seeing precise description of algorithm…

Example 1

Recall: `5+x` is really syntactic sugar for `+(5,x)`, because `+` is infix

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>R</td>
</tr>
<tr>
<td>+(5,x)</td>
<td>S</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
</tr>
<tr>
<td>x</td>
<td>U</td>
</tr>
</tbody>
</table>

R,S,T,U are type variables used during inference
Example 1

- fun g(x) = 5 + x;
val g = fn : int -> int

Step 2: Generate constraints

Function binding constraint: in binding fun \( f \) \( x = e \), if the type of \( f \) is \( A \), the type of \( x \) is \( B \), and the type of \( e \) is \( C \), then \( A = B \rightarrow C \)

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>( R )</td>
</tr>
<tr>
<td>( x )</td>
<td>( U )</td>
</tr>
</tbody>
</table>

Constraint: \( R = U \rightarrow S \)

Step 3: Solve constraints

- \( R = U \rightarrow S \)
- \( int \times int \rightarrow int = T \rightarrow S \)
- \( T = int \times U \)

Example 1

- fun g(x) = 5 + x;
val g = fn : int -> int

Step 2: Generate constraints

Function application constraint: if the type of \( f \) is \( A \), the type of \( e \) is \( B \), and the type of \( f(e) \) is \( C \), then \( A = B \rightarrow C \)

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f ) ( e )</td>
<td>( T )</td>
</tr>
<tr>
<td>( 5 )</td>
<td>( S )</td>
</tr>
<tr>
<td>( x )</td>
<td>( U )</td>
</tr>
</tbody>
</table>

Constraint: \( int \times int \rightarrow int = T \rightarrow S \)

Example 1

- fun g(x) = 5 + x;
val g = fn : int -> int

Step 2: Generate constraints

All constraints:

- \( R = int \times int \rightarrow int = T \rightarrow S \)
- \( T = int \times U \)

Example 1

- fun g(x) = 5 + x;
val g = fn : int -> int

Step 3: Solve constraints

- \( R = U \rightarrow S \)
- \( int \times int \rightarrow int = int \times U \rightarrow S \)
- \( T = int \times U \)
Example 1

Step 3: Solve constraints

\[-\text{fun } g(x) = 5 + x;\]
\[-\text{val } g = \text{fn } : \text{int} \rightarrow \text{int}\]

\[
R = U \rightarrow S
\]
\[
\text{int} \times \text{int} \rightarrow \text{int} = \text{int} \times U \rightarrow S
\]
\[
\text{int} = U
\]
\[
\text{int} = S
\]
**Example 1**

```plaintext
- fun g(x) = 5 + x;
val g = fn : int -> int
```

Step 3: Solve constraints

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>R</td>
</tr>
</tbody>
</table>

Done: type of g is int -> int

**Example 2**

```plaintext
- fun apply(f,x) = f x;
val apply = fn : ('a -> 'b) * 'a -> 'b
```

Step 1: Assign preliminary types to all subexpressions

### Step 2: Generate constraints

- Function binding constraint: in binding `fun f x = a`, if the type of `f` is `A`, the type of `x` is `B`, and the type of `a` is `C`, then `A = B --> C`
- Function application constraint: if the type of `f` is `A`, the type of `a` is `B`, and the type of `(f a)` is `C`, then `A = B --> C`

**Example 3**

```plaintext
- apply(g,3);
val it = 8 : int
```

Step 1: Assign preliminary types to all subexpressions

### Step 2: Generate constraints

- Type of `apply` is polymorphic
- So essentially the same as Example 3

**Example 4**

```plaintext
- apply(not,false);
val it = true : bool
```

**Algorithm for constraint generation**

- Let’s make the algorithm for generating constraints precise
- Formalize and generalize what we just did in the examples
- HM does type checking and constraint generation at the same time
  - We’ll decouple and just look at constraint generation
  - Not hard to mix the two since you understand them independently
  - Either way, you end up with the same types
- For simplicity, we’ll focus on just a very small subset of SML...
Lambda calculus

• The lambda calculus is a subset of ML that contains only three kinds of expressions:

  - No integers, no pattern matching, etc.
  - The “assembly language” of functional programming
  - Doing type inference for all of ML is conceptually no different than doing it for the lambda calculus

<table>
<thead>
<tr>
<th>x</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1 e2</td>
<td>Function application</td>
</tr>
<tr>
<td>fn x =&gt; e</td>
<td>Anonymous functions</td>
</tr>
</tbody>
</table>

Constraint generation

Step 1:

- Assign a unique type variable (e.g., R, S, T, …)
  - one to each variable \( x \) bound by a function in \( e \)
  - one to every subexpression \( e' \) in \( e \)
- Define two functions that return type variables
  - \( D(x) \) returns the type variable assigned to variable \( x \) bound by a function in \( e \)
  - \( U(e') \) returns the type variable assigned to subexpression \( e' \)

Step 2: Generate constraints:

- \( U(x) = D(x) \) for each occurrence of a variable \( x \) as a subexpression
- \( U(e1) \cdot U(e2) \rightarrow U(e1 \cdot e2) \) for each occurrence of a subexpression \( e1 \cdot e2 \)
  - This is really just the “function application constraint” we saw earlier
- \( U(fn x => e) = D(x) \rightarrow U(e) \) for each occurrence of a subexpression \( fn x => e \)
  - This is really just the “function binding constraint” we saw earlier

Return those constraints as output of algorithm

Constraint generation

Step 1:

Example:

- Input: \( fn x => (fn y => x) \)
- Type variables:
  - \( R = D(x) \) for binding \( x \) in \( fn x \)
  - \( S = D(y) \) for binding \( y \) in \( fn y \)
  - \( T \rightarrow U(fn x => (fn y => x)) \) for occurrence of \( fn x \) in \( fn y \)
  - \( X \rightarrow U(fn y => x) \) for occurrence of \( fn y \) in \( x \)
  - \( Y \rightarrow U(x) \) for occurrence of \( x \)

Step 2:

Example (continued):

- \( fn y => x \) occurs as subexpression, so generate constraint \( U(fn y \rightarrow x) = D(x) \)
  - From step 1, have \( U(x) = Y \) and \( D(x) = R \)
  - So constraint is \( Y \rightarrow R \)
- \( fn y \rightarrow x \) occurs as subexpression, so generate constraint \( U(fn y \rightarrow x) = D(x) \rightarrow U(fn y \rightarrow x) \)
  - From step 1, have \( U(x) = Y \), and \( U(fn y \rightarrow x) \rightarrow X \) and \( D(y) = S \)

Constraint generation