CS 334 lecture 9
Type Checking
-check declared types for names against uses

Type Inference
-compute types for names that are consistent with uses.
3 Outcomes

1) Find types:
   \[ \text{fun } f(x) = x + 1; \]
   \[ f: \text{int} \rightarrow \text{int} \]

2) Type error:
   \[ \text{fun } f(x) = f + x; \]

3) Unconstrained types:
   \[ \text{fun } f(x) = x; \]
   \[ f: \text{a} \rightarrow \text{a} \]
Hindley–Milner
- 1970s.

Step 2: Generate Constraints
- Function Defs:
  \[ \lambda : C \]
  \[ \mu : D \]
  \[ X : A \]
  \[ e : B \]

\[ C = A \rightarrow B \]

Step 3
1. \( f = \mu \rightarrow S \)
2. \( t = \text{int} \rightarrow S \)
3. \( \text{int} \rightarrow (\text{int} \rightarrow \text{int}) = \mu \rightarrow T \)
Step 3: Unification

- Find satisfying assignment to variables in equations to make all equations simultaneously true.

- Rules to generate more eqns:
  1) if \( A \rightarrow B = C \rightarrow D \), then \( A = C \) and \( B = D \).
  2) if \( X = A \) and \( C = 0 \), then \( [A/x]C = [A/x]0 \).
Twice

\[ \lambda : C \]
\[ / \]
\[ \times : A \quad e : B \]
\[ C = A \rightarrow B \]

(From Above)

\[ e : F \]
\[ \angle : E \]
\[ e_1 : D \quad e_2 : E \]
\[ D = E \rightarrow F \]

\[ r = z \rightarrow s \]
\[ z = v \times u \]
\[ v = z \rightarrow s \]
\[ v = u \rightarrow z \]

\[ (a \mapsto a) \_ a \mapsto a \]

\[ t \rightarrow s = \mu \rightarrow \tau \]
\[ \mu = \tau \]
\[ s = \tau \]
\[ v = u \rightarrow \mu \]
\[ \tau = (u \rightarrow u) \_ u \]
\[ s = \mu \]
\[ r = (u \rightarrow u) \_ u \mapsto s \]
\[ f \mapsto t = (u \rightarrow u) \_ u \mapsto \mu \]
(From Above)

1) if $A \rightarrow B = C \rightarrow D$, then $A = C$ and $B = D$.

2) if $x = A$ and $C = 0$, then $[A/x]C = [A/x]D$. 
Limitations

1. What to report in error if unification fails?
   - Localize problems...
   - Order unification
   - Heuristics
   - Minimal edit distance

2. Big-O time?
   \( O(2^n) \)
Implementing Parametric Polymorphism

Version 1 Lists

Chars: 1 byte

```
| a | → | b | → | nil |
```

1 byte 4 bytes

Int3: 4 bytes

```
1 → 2 → nil
```

4 bytes

Reals

```
1.1 → 2.2 → nil
```

8 bytes
polymorphic funs: single code sequence
    that works for all possible types.
length: 'a list -> int.

**Version 2 Cists**
- Uniform data representation.
- Boxing

```plaintext
char:
  | \121 |
  \(\text{\text{\textbackslash n}}\)\n
int:
  | 1 |
  | 2 |
  \(\text{\text{\textbackslash n}}\)\n
real:
  | 1.1 |
  | 2.2 |
  \(\text{\text{\textbackslash n}}\)
```
+ uniform data rep ⇒
  single code sequence works
  on all type.
- more work to maintain rep.
C++ Templates

1. No uniform rep.
2. Doing ops on values of type T.
3. Local var of type T.
Template Instantiation

- Generate new code for each version separately
- Type check each version separately.

<table>
<thead>
<tr>
<th>MC</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>✓</td>
</tr>
<tr>
<td>Size</td>
<td></td>
</tr>
<tr>
<td>Speed</td>
<td>✓</td>
</tr>
<tr>
<td>Flexibility</td>
<td></td>
</tr>
<tr>
<td>Ease of use</td>
<td>✓</td>
</tr>
</tbody>
</table>
ML Overloading

+, * for int & reals.

+: int -> int -> int

: real -> real -> real.