4.4 Type Inference

- Type declarations aren't always necessary.
- In our toy typed language, they are *always* optional.
- Consider:

```
letrec fact(n) =
  if zero?(n)
    then 1
    else *(n, (fact sub1(n)))
```

- What is redundant (inferable) about the types?
4.4 Type Inference

- **zero?** is defined as `int->bool`, so `n` must be `int`
- **then** returns 1, so **fact** must return **int**
- so we have inferred all the types

```plaintext
letrec int fact(int n) =
  if zero?(n)
  then 1
  else *(n, (fact sub1(n)))
```
**Unification**

- Treat programs as formulas over *type variables*.
- Solve for variables using algebra.
- This method is called *unification* (ML, Prolog)
- Another example:

  \[
  \text{proc}(? f, ? x) \ (f + (1, x) \ \text{zero}?(x))
  \]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>( tf )</td>
</tr>
<tr>
<td>( x )</td>
<td>( tx )</td>
</tr>
<tr>
<td>((f + (1, x) \ \text{zero}?(x)))</td>
<td>( t1 )</td>
</tr>
<tr>
<td>(+ (1, x))</td>
<td>( t2 )</td>
</tr>
<tr>
<td>( \text{zero}?(x) )</td>
<td>( t3 )</td>
</tr>
</tbody>
</table>
Unification

proc(? f, ? x) (f +(1,x) zero?(x))

f tf
x tx
(f +(1,x) zero?(x)) t1
+(1,x) t2
zero?(x) t3

• Type of whole expression = (tf * tx -> t1)

• Solve for tf, tx, t1:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type_Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f +(1,x) zero?(x))</td>
<td>tf = (t2*t3-&gt;t1)</td>
</tr>
<tr>
<td>+(1,x)</td>
<td>(int<em>int-&gt;int) = (int</em>tx-&gt;t2)</td>
</tr>
<tr>
<td>zero?(x)</td>
<td>(int-&gt;bool) = (tx-&gt;t3)</td>
</tr>
</tbody>
</table>
# Unification

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(f + (1, x) \text{ zero?}(x))$</td>
<td>$tf = (t2*t3-&gt;t1)$</td>
</tr>
<tr>
<td>$(1, x)$</td>
<td>$(\text{int<em>int}-&gt;\text{int}) = (\text{int</em>tx}-&gt;t2)$</td>
</tr>
<tr>
<td>$\text{zero?}(x)$</td>
<td>$(\text{int-&gt;bool}) = (\text{tx}-&gt;t3)$</td>
</tr>
</tbody>
</table>

• Therefore:

$$tx = \text{int} \quad t3 = \text{bool} \quad t2 = \text{int}$$

$$tf = (\text{int }\ast\text{ bool }\rightarrow \text{ t1})$$

$$t1 = ?$$

• Body = $((\text{int }\ast\text{ bool }\rightarrow \text{ t1})\ast\text{ int }\rightarrow \text{ t1})$

• I.e., body is *polymorphic in t1*. 
Unification

• So we need:  (1) data structure for type variables
(2) unification algorithm

• Name all variables \( t_1, t_2, t_3, \ldots \) (“serial number”)

• Variable has an (initially empty) container filled by unification:

\[
\begin{align*}
\text{(define-datatype type type?} & \text{ (atomic-type (name symbol?)))} \\
\text{(proc-type} & \text{ (arg-types (list-of type?)))} \\
\text{(result-type type?))} \\
\text{(tvar-type} & \text{ (serial-number integer?)}) \\
\text{(container vector?)))}
\end{align*}
\]
Unification: \texttt{check-equal-type!}((t_1, t_2))

1. If \texttt{t}_1 and \texttt{t}_2 are atomic types (\texttt{bool}, \texttt{int}), succeed if they have the same name; else fail. (Note that constants like \texttt{1} and \texttt{true} are implicitly typed: \texttt{int} and \texttt{bool}, respectively.)

2. If \texttt{t}_1 is a type var, check that its contents are the same as the contents of \texttt{t}_2 (and vice-versa) using \texttt{check-tvar-equal-type!} (see next page). If either is an (unassigned) variable, set its contents to the contents of the other: \textit{type inference}!

3. If \texttt{t}_1 and \texttt{t}_2 are procedure types, check \# of args equal and recur on args.

4. Otherwise, fail.
(define check-tvar-equal-type!
 (lambda (tvar ty exp)
   (if (tvar-non-empty? tvar)
       (check-equal-type!
         (tvar->contents tvar) ty exp)
       (begin
         (check-no-occurrence! tvar ty exp)
         (tvar-set-contents! tvar ty)))))

Need `check-no-occurrence!` to avoid, eg.,

t1 = (int -> t1).