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

```
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:

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

Expression	Type Variable
`f`	`tf`
`x`	`tx`
`(f +(1,x) zero?(x))`	`t1`
`+(1,x)`	`t2`
`zero?(x)`	`t3`

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`**:

Expression	Type Equation
`(f +(1,x) zero?(x))`	`tf = (t2*t3->t1)`
`+(1,x)`	`(int*int->int) = (int*tx->t2)`
`zero?(x)`	`(int->bool) = (tx->t3)`

Unification

Expression	Type Equation
`(f +(1,x) zero?(x))`	`tf = (t2*t3->t1)`
`+(1,x)`	`(int*int->int) = (int*tx->t2)`
`zero?(x)`	`(int->bool) = (tx->t3)`

- Therefore:

```
tx = int          t3 = bool        t2 = int

tf = (int * bool -> t1)

t1 = ?
```

- Body = `((int * bool -> t1)* int -> t1)`
- *I.e.*, body is *polymorphic in* `t1`.

Unification

- So we need: (1) data structure for type variables

  (2) unification algorithm
- Name all variables *t1*, *t2*, *t3*, ... ("serial number")
- Variable has an (intially empty) container filled by unification:

```
(define-datatype type type?
  (atomic-type (name symbol?))
  (proc-type
    (arg-types (list-of type?))
    (result-type type?))
  (tvar-type
    (serial-number integer?)
    (container vector?)))
```

Unification: `check-equal-type!(t1,t2)`

1. If **`t1`** and **`t2`** are atomic types (**`bool`**, **`int`**), suceed if they have the same name; else fail. (Note that constants like **`1`** and **`true`** are implicitly typed: **`int`** and **`bool`**, respectively.)

2. If **`t1`** is a type var, check that its contents are the same as the contents of **`t2`** (and vice-versa) using **`check-tvar-equal-type!`** (see next page). If either is an (unassigned) variable, set its contents to the contents of the other: *type inference!*

3. If **`t1`** and **`t2`** 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-occurence!`** to avoid, eg.,

`t1 = (int -> t1)`.