

Current Book Language

```
<tyexpr> ::= int  
          ::= bool  
          ::= (<tyexpr> -> <tyexpr>)
```

Types versus Type Expressions

<tyexpr>		<type>
int	expands to	int
bool	expands to	bool
(bool -> int)	expands to	(bool → int)
	etc.	

Datatype for types:

```
(define-datatype  
  type type?  
  (atomic-type (name symbol?))  
  (proc-type (arg-types (list-of type?))  
            (result-type type?)))  
  
(define int-type (atomic-type 'int))  
(define bool-type (atomic-type 'bool))
```

Implementing a Type Checker

```
;; type-of-expression : expr tenv -> type
;; signals an error if no type for exp
;;
(define (type-of-expression exp tenv)
  (cases expression exp
    (lit-exp ...)
    (true-exp ...)
    (false-exp ...)
    (var-exp ...)
    (primapp-exp ...)
    (proc-exp ...)
    (app-exp ...)
    (if-exp ...)
    (let-exp ...)
    (letrec-exp ...)))
```

Implementation: lit-exp case

- Example:

5

- The rule from previous lecture:

$$E \vdash \text{<num>} : \text{int}$$

- In Scheme:

(lit-exp (n) int-type)

Implementation: true-exp and false-exp case

- Example:

true

- The rule from previous lecture:

$E \vdash \text{<bool>} : \text{bool}$

- In Scheme:

(**true-exp** () **bool-type**)
(**false-exp** () **bool-type**)

Implementation: var-exp case

- Example:

... X ...

- The rule from previous lecture:

$$\{ \dots \langle \text{id} \rangle : T \dots \} \vdash \langle \text{id} \rangle : T$$

- In Scheme:

```
(var-exp (id) (apply-tenv tenv id))
;; where apply-tenv signals an error
;; if id is not in tenv
```

Implementation: if-exp case

if true then 5 else +(1,2)

- The rule from previous lecture:

$$\frac{E \vdash e_1 : \text{bool} \quad E \vdash e_2 : T_0 \quad E \vdash e_3 : T_0}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T_0}$$

- In Scheme:

```
(if-exp (test-exp then-exp else-exp)
  (let ((test-type (type-of-expr test-exp tenv))
        (then-type (type-of-expr then-exp tenv))
        (else-type (type-of-expr else-exp tenv)))
    ;; succeeds or signals an error:
    (check-equal-type! test-type bool-type)
    (check-equal-type! then-type else-type)
    then-type)
```

Implementation: proc-exp case

`proc(int x, bool y)if y then x else 0`

- The rule from previous lecture:

$$\frac{\{ \langle id \rangle_1 : T_1, \dots \langle id \rangle_n : T_n \} + E \vdash e : T_0}{E \vdash \text{proc}(T_1 \langle id \rangle_1, \dots T_n \langle id \rangle_n) e : (T_1 \times \dots T_n \rightarrow T_0)}$$

- In Scheme:

```
(proc-exp (texps ids body)
  (let* ((arg-tys (expand-tyexprs texps))
         (new-tenv (extend-tenv ids arg-tys tenv))
         (res-type (type-of-expr body new-tenv)))
    (proc-type arg-types res-type)))
```

Implementation: app-exp case

(**proc**(int x, int y)+(x,y) 6 7)

- The rule from previous lecture:

$$\frac{E \vdash e_0 : (T_1 \times \dots \times T_n \rightarrow T_0) \quad E \vdash e_1 : T_1 \quad \dots \quad E \vdash e_n : T_n}{E \vdash (e_0 \ e_1 \ \dots \ e_n) : T_0}$$

- In Scheme:

```
(app-exp (rator rands)
         (type-of-application
          (type-of-expression rator tenv)
          (types-of-expressions rands tenv))))
```

Implementation: app-exp case

```
(define (type-of-application rator-ty rand-tys)
  (cases type rator-ty
    (proc-type (arg-tys result-ty)
      (if (= (length arg-tys) (length rand-tys))
          (begin
            (check-equal-types! rand-tys arg-tys)
            result-ty)
          (error 'wrong-arg-count)))
    (else (error 'not-a-proc))))
```

Implementation: primapp-exp case

$+(1, 2)$

- The rule from previous lecture:

$$\frac{E \vdash e_1 : \text{num} \quad E \vdash e_1 : \text{num}}{E \vdash +(e_1, e_2) : \text{num}}$$

- In Scheme (completely different):

```
(primapp-exp (prim rands)
             (type-of-application
              (type-of-primitive prim)
              (types-of-expressions rands tenv)))
```

Implementation: primapp-exp case

```
(define (type-of-primitive prim)
  (cases primitive prim
    (add-primitive ())
      (proc-type (list int-type int-type)
                 int-type))
    ...))
```

Implementation: let-exp case

```
let x = 5
  f = proc(int y)false
in (f x)
```

- In Scheme:

```
(let-exp (ids rands body)
  (let* ((rand-tys (types-of-exprs rands tenv))
         (body-tenv (extend-tenv ids rand-tys
                                  tenv)))
    (type-of-expression body body-tenv)))
```

Implementation: letrec-exp case

```
letrec int f(int x) = (g +(x,1) false)
    int g(int y, bool b) = if b then (f y) else y
in (g 10 true)
```

- In Scheme:

```
(letrec-exp (res-texps proc-ids texpss idss bodies
                       body)
  (let*((arg-tyss (expand-tyexprss texpss))
        (res-tys (expand-tyexprs res-texps))
        (proc-tyss (map proc-type arg-tyss res-tys)))
    (new-tenv (extend-tenv proc-ids proc-tyss
                           tenv))))  
...)
```

Implementation: letrec-exp case

```
letrec int f(int x) = (g +(x,1) false)
    int g(int y, bool b) = if b then (f y) else y
in (g 10 true)
```

- In Scheme:

```
(letrec-exp (res-texps proc-ids texpss idss bodies
body)
...
(for-each
(lambda (ids arg-tys body res-ty)
(check-equal-type! res-ty
(type-of-expr body
(extend-tenv ids arg-tys new-tenv))))
idss arg-tyss bodies res-tys)
(type-of-expression body new-tenv))
```

Type-Checking Expressions

- What is the type of the following expression?

`proc(x)+(x,1)`

- **Answer:** Yet another trick question; it's not an expression in our typed language, because the argument type is missing
- But it seems like the answer *should* be (`int → int`)

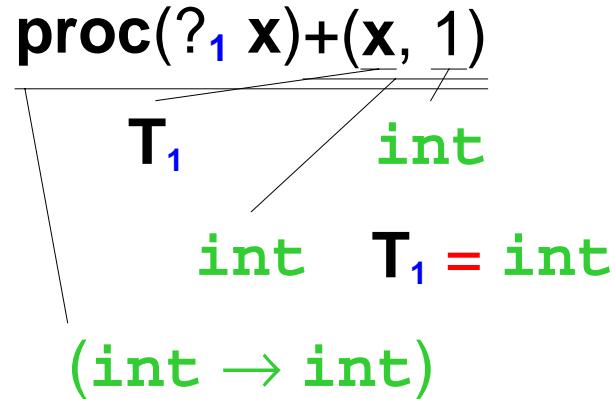
Type Inference

- **Type inference** is the process of inserting type annotations where the programmer omits them
- We'll use explicit question marks, to make it clear where types are omitted

proc (? **x**)+(x,1)

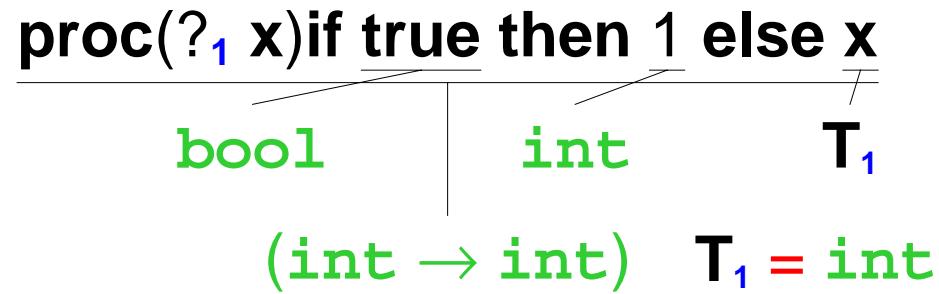
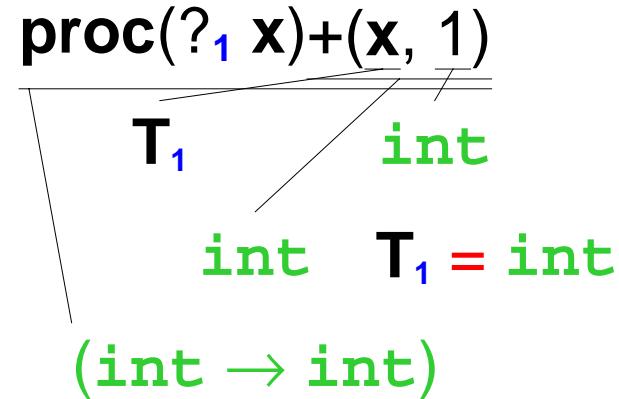
<tyexpr> ::= **int**
::= **bool**
::= (**<tyexpr>** -> **<tyexpr>**)
::= ?

Type Inference



- Create a new type variable for each `?`
- Change type comparison to install type equivalences

Type Inference



Type Inference: Impossible Cases

`proc(?1 x)if x then 1 else x`

$T_1 \quad | \quad \text{int} \quad T_1$

no type: T_1 can't be both `bool` and `int`

Type Inference: Many Cases

$$\frac{\mathbf{proc}(\mathbf{?}_1 \mathbf{y})\mathbf{y}}{\mathbf{T}_1}$$

$(\mathbf{T}_1 \rightarrow \mathbf{T}_1)$

- Sometimes, more than one type works

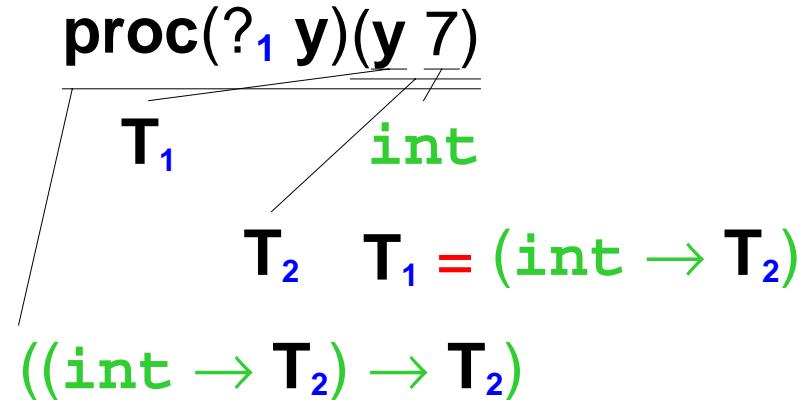
- $(\mathbf{int} \rightarrow \mathbf{int})$
- $(\mathbf{bool} \rightarrow \mathbf{bool})$
- $((\mathbf{int} \rightarrow \mathbf{bool}) \rightarrow (\mathbf{int} \rightarrow \mathbf{bool}))$

so the type checker leaves variables in the reported type

Type Inference: Function Calls

$$\frac{(\mathbf{proc}(\mathbf{?}_1 \mathbf{y})\mathbf{y} \quad \mathbf{proc}(\mathbf{?}_2 \mathbf{x})+(\mathbf{x}, 1))}{\begin{array}{c} (\mathbf{T}_1 \rightarrow \mathbf{T}_1) \qquad \qquad (\mathbf{int} \rightarrow \mathbf{int}) \\ \qquad \qquad \qquad (\mathbf{int} \rightarrow \mathbf{int}) \\ \mathbf{T}_1 = (\mathbf{int} \rightarrow \mathbf{int}) \end{array}}$$

Type Inference: Function Calls



- In general, create a new type variable record for the result of a function call

Type Inference: Cyclic Equations

`proc(?1 x)(x x)`

no type: T_1 can't be $(T_1 \rightarrow \dots)$

- T_1 can't be `int`
- T_1 can't be `bool`
- Suppose T_1 is $(T_2 \rightarrow T_3)$
 - T_2 must be T_1
 - So we won't get anywhere!

Type Inference: Cyclic Equations

`proc(?1 x)(x x)`

no type: T_1 can't be $(T_1 \rightarrow \dots)$

- When installing a type equivalence, make sure that the new type for T doesn't already contain T

Implementing Type Inference

- Extend `type` datatype with `tvar-type` variant

```
(define-datatype  
  type type?  
  ...  
  (tvar-type (serial-number integer?)  
             (container vector?)))
```

- Create a new type variable record for each ?
 - Initial container value is "don't know", '()
- Create a new type variable record for each application
- Change `check-equal-type!` to read and set type variable containers