Typed Languages

program data is typed.
operations are only appropriate for some types.
*type error*: applying operation to inappropriate data.

*static typing*: analysis before execution.
*latent, dynamic typing*: data is tagged, tags are checked during execution — overhead; also, may limit abstraction.
*untyped execution*: whatever hardware and representation produce, no checks.
Analysis

1. set of types, expressed value $v$ has type $t$.
2. analyzer assigns type $t$ to each expression $e$. Type system is sound if $t$ also results from execution of $e$.
3. analyzer inspects; detects potential type errors.
4. analyzer might correct, prevent execution, etc.

Diagram:
- source
- front end
- tree
- analyzer
- translator
- target
First Example

type-exp: 'int'

| int-type-exp() |
| 'bool' |
| bool-type-exp() |
| '(' type-exp/'*'/ '->' type-exp ')' |

proc-type-exp(arg-texps result-texp)

types in programs and type expressions equivalent. expressed value is of type int iff it is an integer, of type bool iff it is a boolean, and of the procedure type iff it is a procval with matching argument types, argument numbers, and result type.
### Describing Scheme's Expressed Values

<table>
<thead>
<tr>
<th>Type</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\text{int} \to \text{bool}))</td>
<td>\text{even?}</td>
</tr>
<tr>
<td>((\text{int} \times \text{int} \to \text{int}))</td>
<td>+</td>
</tr>
<tr>
<td>((\text{int} \to (\text{int} \to \text{int})))</td>
<td>((\lambda \text{x} \to (\text{int} \to \text{int})))</td>
</tr>
<tr>
<td>(((\text{int} \to \text{int}) \times \text{int} \to \text{bool}))</td>
<td>((\lambda \text{f} \text{x} \to \text{bool}))</td>
</tr>
</tbody>
</table>

Types are of the values of the expression, not of the expressions themselves.
Strongly Statically Typed

program passing checker will not make type error.

type errors:

3. applying integer or boolean to argument.
4. applying procedure or primitive to wrong number of arguments.
5. applying primitive expecting integer to something else.
6. using non-boolean as test in conditional expression.

division by zero is not a type error — cannot be detected statically.
type-of-expression

(define type-of-expression
  (lambda (exp tenv) ..

exp: an expression.
tenv: a type environment, maps each variable to a type.
type-of-expression assigns to exp a type t so that if exp is executed in an environment agreeing with tenv then the resulting value has type t.
if we know the types of all variable values we should be able to deduce the expression value type — this will be t.
Typing Rules

(type-of-expression number tenv) = int
(type-of-expression id tenv) =
    (apply-env tenv id)

if (type-of-expression r tenv) =
    (t₁ * t₂ * .. * tₙ -> t)
and (type-of-expression n₁ tenv) = t₁
and (type-of-expression n₂ tenv) = t₂
...
and (type-of-expression nₙ tenv) = tₙ
then (type-of-expression (r n₁ n₂ .. nₙ) tenv) = t
Typing Rule for Conditionals

\[
\begin{align*}
\text{(type-of-expression } & \text{ test-exp tenv}) = \text{ bool} \\
\text{(type-of-expression } & \text{ true-exp tenv}) = t \\
\text{(type-of-expression } & \text{ false-exp tenv}) = t \\
\text{(type-of-expression} & \text{ if test-exp then true-exp else false-exp tenv}) = t
\end{align*}
\]

test value must be boolean, alternatives' values must have same type, which is assigned as result type.
Typing Rule for Procedures

\[(\text{type-of-expression } \text{exp} 
\quad [x_1 = t_1, \ldots, x_n = t_n] + \text{tenv}) = t)\]

\[(\text{type-of-expression } \text{proc } (x_1, \ldots, x_n) \text{ exp} 
\quad \text{tenv}) = (t_1 \ast \ldots \ast t_n \rightarrow t)\]

i.e., to compute the type of a procedure expression, we need to extend the type environment with the types of the procedure parameters (bound variables)
Approaches

**type checking**
programmer must supply types of bound variables, type checker deduces rest and checks consistency.

**type inference**
type checker attempts to infer the types of bound variables by checking their use in the program. This is (mostly) possible for a well-designed language.
Types

Programming Language Essentials
2nd edition
Chapter 4.2 Type Checking
Extended Language (1)

program: expr

expr: number | id
    | 'true' | 'false'
    | true-exp () | false-exp ()
    | primitive '(' expr/',' ')' '
    | 'if' expr 'then' expr 'else' expr
    | 'let' (id '=?' expr)* 'in' expr
    | 'proc' '(' (type-exp id)/',' ')' expr
    | proc-exp (arg-texps ids body)
    | '(' expr expr* ')
    | 'letrec' ( type-exp id
        '(' (type-exp id)/',' ')' '=?' expr *)
        'in' expr

letrec-exp (resultts names argtss idss bodies body)
Extended Language (2)

type-exp: 'int'

<table>
<thead>
<tr>
<th>int-type-exp()</th>
</tr>
</thead>
<tbody>
<tr>
<td>'bool'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bool-type-exp()</th>
</tr>
</thead>
<tbody>
<tr>
<td>(' type-exp/''*'' '-&gt;' type-exp ')</td>
</tr>
</tbody>
</table>

proc-type-exp(arg-texps result-texp)

primitive: '+' | '-' | '*' | 'add1' | 'sub1' |
| 'zero?'
Usage

proc (int x) add1(x)

letrec
    int fact (int x) =
        if zero?(x) then 1
        else *(x, (fact sub1(x)))
    in fact

both of these have the same type:

(int -> int)
Representing Types

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

(define int-type (atomic-type 'int))
(define bool-type (atomic-type 'bool))
expand-type-expression

(define expand-type-expression
  (lambda (texp)
    (cases type-exp texp
      (int-type-exp () int-type)
      (bool-type-exp () bool-type)
      (proc-type-exp (arg-texps result-texp)
        (proc-type
          (expand-type-expressions arg-texps)
          (expand-type-expression result-texp)
        )
      )
    )
  )
)

(define expand-type-expressions
  (lambda (texps) (map expand-type-expression texp))
)

converts from expression to type representation.
type-to-external-form

(define type-to-external-form
  (lambda (ty)
    (cases type ty
      (atomic-type (name) name)
      (proc-type (arg-types result-type)
        (append
         (arg-types-to-external-form arg-types)
         '(:->)
         (list (type-to-external-form result-type))
        )
      ))
    )
  )
)

converts from type representation to the string for an expression.
arg-types-to-external-form

(define arg-types-to-external-form
  (lambda (types)
    (if (null? types)
        '()
        (if (null? (cdr types))
            (list (type-to-external-form (car types)))
            (cons
              (type-to-external-form (car types))
              (cons '*
                (arg-types-to-external-form (cdr types))
              )
            )
        )
  )
)

deals with types for procedure arguments.
type-of-expression

(define type-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp)
        (type-of-expression exp (empty-tenv)))))

(define type-of-expression
  (lambda (exp tenv)
    (cases expression exp
      (lit-exp (number) int-type)
      (true-exp () bool-type)
      (false-exp () bool-type)
      (var-exp (id) (apply-tenv tenv id)) ..
Type Environment

(define-datatype type-environment type-environment?
  (empty-tenv-record)
  (extended-tenv-record
    (syms (list-of symbol?))
    (vals (list-of type?))
    (tenv type-environment?)
  ) )

(define empty-tenv empty-tenv-record)
(define extend-tenv extended-tenv-record)

identical to environment, but unfortunately with a buried semantic error message.
(define apply-tenv
  (lambda (tenv sym)
    (cases type-environment tenv
      (empty-tenv-record ()
        (eopl: error 'apply-tenv "Unbound variable ~s" sym))
      (extended-tenv-record (syms vals env)
        (let ((pos (list-find-position sym syms)))
          (if (number? pos)
            (list-ref vals pos)
            (apply-tenv env sym)
          ))))))
Typing Rule for Conditionals

\[
\begin{align*}
\text{(type-of-expression } \texttt{test-exp} \text{ tenv)} &= \text{ bool} \\
\text{(type-of-expression } \texttt{true-exp} \text{ tenv)} &= t \\
\text{(type-of-expression } \texttt{false-exp} \text{ tenv)} &= t \\
\text{(type-of-expression} \\
\text{ if } \texttt{test-exp} \text{ then } \texttt{true-exp} \text{ else } \texttt{false-exp} \text{ tenv}) &= t
\end{align*}
\]

test value must be boolean, alternatives' values must have same type, which is assigned as result type.
(define type-of-expression
  (lambda (exp tenv)
    (cases expression exp ..
      (if-exp (test-exp true-exp false-exp)
        (let ((test-type
               (type-of-expression test-exp tenv))
               (false-type
               (type-of-expression false-exp tenv))
               (true-type
               (type-of-expression true-exp tenv))
               (check-equal-type!
                test-type bool-type test-exp)
               (check-equal-type!
                true-type false-type exp)
               true-type)) ..
  )
)

(define check-equal-type!)
    (lambda (t1 t2 exp)
        (if (not (equal? t1 t2))
            (eopl:error 'check-equal-type!
                "Types didn't match: ~s != ~s in~%~s"
                (type-to-external-form t1)
                (type-to-external-form t2)
                exp
        ) ) ) )

equal? can recursively compare trees.
Typing Rule for Procedures

\[
\text{type-of-expression } \exp
\]

\[
\begin{array}{l}
[x_1 = t_1, \ldots, x_n = t_n] + \text{tenv} = t
\end{array}
\]

\[
\text{type-of-expression}
\]

\[
\begin{array}{l}
\text{proc } (t_1 x_1, \ldots, t_n x_n) \exp
\end{array}
\]

\[
\text{tenv} = (t_1 * \ldots * t_n \rightarrow t)
\]

the procedure expression is defined with parameter types.
type-of-expression (proc)

(define type-of-expression
  (lambda (exp tenv)
    (cases expression exp ..
      (proc-exp (texps ids body)
        (type-of-proc-exp texps ids body tenv)) ..
    )
  )

(define type-of-proc-exp
  (lambda (texps ids body tenv)
    (let*
      ((arg-types (expand-type-expressions texps))
       (result-type (type-of-expression body
                     (extend-tenv ids arg-types tenv)))
      )
      (proc-type arg-types result-type)
    )
  )


Typing Rule for Application

\[(\text{type-of-expression } r \text{ tenv}) = (t_1 \ast t_2 \ast \ldots \ast t_n \rightarrow t)\]

\[(\text{type-of-expression } n_1 \text{ tenv}) = t_1\]

\[(\text{type-of-expression } n_2 \text{ tenv}) = t_2\]

\[\ldots\]

\[(\text{type-of-expression } n_n \text{ tenv}) = t_n\]

\[(\text{type-of-expression } (r \ n_1 \ n_2 \ldots \ n_n) \text{ tenv}) = t\]

this can include primitives if they are specified as procedures.
(define type-of-expression
  (lambda (exp tenv)
    (cases expression exp ..
      (primapp-exp (prim rands)
        (type-of-application
          (type-of-primitive prim)
          (types-of-expressions rands tenv)
          prim rands exp))
      (app-exp (rator rands)
        (type-of-application
          (type-of-expression rator tenv)
          (types-of-expressions rands tenv)
          rator rands exp)) ..
(define type-of-primitive
  (lambda (prim)
    (cases primitive prim
      (add-prim () ; (int * int -> int)
        (proc-type
          (list int-type int-type)
          int-type)
      ) ) ..)
(define type-of-application
  (lambda (rator-type rand-types rator rands exp)
    (cases type rator-type
      (proc-type (arg-types result-type)
        (if (= (length arg-types) (length rand-types))
          (begin
            (for-each
              check-equal-type!
              rand-types arg-types rands
            )
            result-type
          )
          (eopl:error 'type-of-expression ..))
        (else (eopl:error 'type "not a procedure"))
      ))
  )
)
Typing Rule for let

\[(\text{type-of-expression} \ e_1 \ \text{tenv}) = t_1\]

.. 

\[(\text{type-of-expression} \ e_n \ \text{tenv}) = t_n\]

\[(\text{type-of-expression} \ \text{body} \ [x_1 = t_1, \ldots, x_n = t_n]+\text{tenv}) = t\]

\[(\text{type-of-expression} \ \text{let} \ x_1 = e_1 \ .. \ x_n = e_n \ \text{in} \ \text{body} \ \text{tenv}) = t\]
(define type-of-expression
  (lambda (exp tenv)
    (cases expression exp ..
      (let-exp (ids rands body)
        (type-of-let-exp ids rands body tenv)) ..
    ))

(define type-of-let-exp
  (lambda (ids rands body tenv)
    (let ((tenv-for-body
          (extend-tenv ids
            (types-of-expressions rands tenv)
            tenv)))
      (type-of-expression body tenv-for-body))))
letrec

letrec
  \( t_1 \ p_1 (t_{11} \ x_{11}, \ldots, t_{1n} \ x_{1n}) = e_1 \)
  \( t_2 \ p_2 (t_{21} \ x_{21}, \ldots, t_{2m} \ x_{2m}) = e_2 \)
  \ldots \)

in body

\( p_i \) has \( n_i \) parameters and result type \( t_i \).

parameter \( x_{ij} \) has type \( t_{ij} \).

\( (t_{i1} \ast t_{i2} \ldots \ast t_{im} \rightarrow t_i) \)

need to check in an environment where the variables are in scope.
Typing Rule for letrec (1)

\[
\begin{align*}
\text{(type-of-expression } & e_1 \text{ tenv}_1) = t_1 \\
\text{(type-of-expression } & e_2 \text{ tenv}_2) = t_2 \\
\ldots
\text{(type-of-expression } & \text{ body tenv}_{\text{body}}) = t \\
\text{(type-of-expression } & \text{ letrec} \\
\quad \quad t_1 p_1 (t_{11} x_{11}, \ldots t_{1n} x_{1n}) = e_1 \\
\quad \quad t_2 p_2 (t_{21} x_{21}, \ldots t_{2m} x_{2m}) = e_2 \\
\ldots \\
\quad \quad \text{in body tenv}) = t
\end{align*}
\]

Typing Rule for \texttt{letrec} (2)

body has the procedures in scope; therefore

\[
\text{tenv}_{\text{body}} = [p_1 = (t_{11} * t_{12} \ldots \rightarrow t_1), \\
p_2 = (t_{21} * t_{22} \ldots \rightarrow t_2), \ldots \\
]\text{tenv}
\]

i-th procedure body $e_i$ also has procedures in scope, plus the parameters $x_{ij}$ with types $t_{ij}$:

\[
\text{tenv}_i = [x_{i1} = t_{i1}, x_{i2} = t_{i2}, \ldots] \text{tenv}_{\text{body}}
\]

unlike \texttt{let}, all procedure result types need to be known to compute each procedure body.
type-of-expression (letrec)

(define type-of-expression
  (lambda (exp tenv)
    (cases expression exp ..
      (letrec-exp
        (resultts names texpss idss bodies body)
        (type-of-letrec-exp
          (resultts names texpss idss bodies body)
          tenv)) ..

    (define type-of-letrec-exp
      (lambda (resultts names texpss idss bodies body)
        (let* ((arg-typess ..
          (result-types ..
            (proc-types ..
              (tenv-body ..
                (type-of-expression body tenv-body)
              )
            )
          )
        )
      ))
"
(define type-of-letrec-exp
  (lambda (resultts names texpss idss bodies body)
    (let* ((arg-typess
        (map
          (lambda (texpss)(expand-type-expressions texpss))
          texpss
        )
      (result-types
        (expand-type-expressions resultts
      )
    (proc-types
      (map proc-type arg-typess result-types
    )
    (tenv-body
      (extend-tenv names proc-types tenv)
    )
  )
)
)
(define type-of-letrec-exp
  (lambda (resultts names texpss idss bodies body)
    (let* ((arg-typess ..
            (result-types ..
            (proc-types ..
            (tenv-body ..
              (for-each
                (lambda (ids argts body result-type)
                  (check-equal-type!
                    (type-of-expression body
                      (extend-tenv ids argts tenv-body))
                    result-type
                    body
                    )
                  )))
              idss arg-typess bodies result-types
            )
        ))
    (type-of-expression body tenv-body))))
How to execute

$ scheme48 -I eopl.image
> ,load ../3/circular-environment.scm
> ,load 4.2.scm
> ,load ../3/boilerplate.scm
> ,load boilerplate.scm
> (type-check "
    letrec
        int fact (int x) =
            if zero?(x) then 1
            else *(x, (fact sub1(x)))
    in fact
"
'(int -> int)

$ cd 4; make 4.2
Chapter 4.3 Enforcing Abstract Boundaries
Type Encapsulation

Types are introduced to hide value representation. Using values in a way based on a representation is a type error.

```
(define s (extend-subst 'x 1 (empty-subst)))
```

Even if substitutions are implemented as procedures:

```
(apply-subst s 'z) ; always legal
(s 'z) ; type error!
```
Extended Language

expr: 'lettype' id '=' type-exp
    ( type-exp id
    '('* (type-exp id)/',',' ')') '=' expr )*
    'in' expr

lettype-exp (name texp
    resultts names argtss idss bodies
    body)

type-exp: id
    tid-type-exp (id)
Usage: Integer Simulation

lettype myint = int
    myint zero () = 1
    myint succ (myint x) = add1(x)
    myint pred (myint x) = sub1(x)
    bool iszero? (myint x) = zero?(sub1(x))
in ..

(succ (zero)) ; ok
add1((zero)) ; type error
(iszero? (zero)) ; ok
(iszero? 1) ; type error
lettype ff = (int -> int)
    ff zero-ff () = proc (int k) 0
    ff extend-ff (int k, int val, ff old-ff) = 
        proc (int k1)
            if zero?(-(k1,k))
            then val
            else (apply-ff old-ff k1)
    int apply-ff (ff f, int k) = (f k)
lettype ff = (int -> int)
    ff zero-ff () = proc (int k) 0
    ff extend-ff (int k, int val, ff old-ff) = proc (int k1)
        if zero?(-(k1,k))
        then val
        else (apply-ff old-ff k1)
    int apply-ff (ff f, int k) = (f k)
in let ff1 (extend-ff 1 11
    (extend-ff 2 22
    (zero-ff)))
in (apply-ff ff1 2)
Usage: Environment Simulation

lettype ff = (int -> int)
    ff zero-ff () = proc (int k) 0
    ff extend-ff (int k, int val, ff old-ff) = proc (int k1)
        if zero?(-(k1,k))
        then val
        else (apply-ff old-ff k1)
    int apply-ff (ff f, int k) = (f k)
in let ff1 (extend-ff 1 11
    (extend-ff 2 22
        (zero-ff)))
in (ff1 2) ; type error
lettype \( tid = t \)

\[
\begin{align*}
    t_1 \ p_1 (t_{11} \ x_{11}, \ldots, t_{1n} \ x_{1n}) &= e_1 \\
    t_2 \ p_2 (t_{21} \ x_{21}, \ldots, t_{2m} \ x_{2m}) &= e_2 \\
    \ldots
\end{align*}
\]

in \textit{body}

check implementation in type environment with \( tid \) bound to \( t \), i.e., representation visible.

check \textit{body} in type environment with \( tid \) bound to a new atomic type, i.e., representation unknown.
type-environment

(define-datatype type-environment type-environment?
  (empty-tenv-record)
  (extended-tenv-record
    (syms (list-of symbol?))
    (vals (list-of type?))
    (tenv type-environment?))
  (typedef-record
    (name symbol?)
    (definition type?)
    (tenv type-environment?)))
)

tid is only searched in typedef-record.
(define find-typedef
  (lambda (tenv0 sym)
    (let loop ((tenv tenv0))
      (cases type-environment tenv
        (empty-tenv-record ()
          (eopl: error 'apply-tenv ..))
        (extended-tenv-record (syms vals tenv)
          (loop tenv))
        (typedef-record (name type tenv)
          (if (eqv? name sym) type
            (loop tenv)
            (loop tenv)))))
apply-tenv

(define apply-tenv
  (lambda (tenv sym)
    (cases type-environment tenv
      (empty-tenv-record ..
      (extended-tenv-record ..
      (typedef-record (name type tenv)
        (apply-tenv tenv tenv sym)
      )
    )
  )
) ) ) ) )
(define expand-type-expression
  (lambda (texp)
    (cases type-exp texp
      (int-type-exp () int-type)
      (bool-type-exp () bool-type)
      (proc-type-exp (arg-texps result-texp)
        (proc-type
          (expand-type-expressions arg-texps)
          (expand-type-expression result-texp))
        ) ) ) )
)

(define expand-type-expressions
  (lambda (texps) (map expand-type-expression texps)))
)

converts from expression to type representation.
(define expand-type-expression
  (lambda (texp tenv)
    (cases type-exp texp
      (tid-type-exp (id) (find-typedef tenv id)
        (int-type-exp () int-type)
        (bool-type-exp () bool-type)
        (proc-type-exp (arg-texps result-texp)
          (proc-type
            (expand-type-expressions arg-texps tenv)
            (expand-type-expression result-texp tenv)))))
  
  (define expand-type-expressions
    (lambda (texps tenv)
      (map
        (lambda (texp) (expand-type-type-expression text tenv)
          texps)))
lettype \( \text{tid} = t \)
\[
\begin{align*}
\ t_1 \ p_1 \ (t_{11} \ x_{11}, \ldots, t_{1n} \ x_{1n}) &= e_1 \\
\ t_2 \ p_2 \ (t_{21} \ x_{21}, \ldots, t_{2m} \ x_{2m}) &= e_2 \\
\ \ldots
\end{align*}
\]
in \textit{body}

\( \text{tenv}_{\text{implementation}} = [\text{tid} = t]\text{tenv} \)

\( \text{tenv}_{\text{client}} = [\text{tid} = \text{fresh}]\text{tenv} \)
Type Environments

lettype \( \textit{tid} = t \)

\[ t_1 \ p_1 \ (t_{11} \ x_{11}, \ldots, t_{1n} \ x_{1n}) = e_1 \]

\[ t_2 \ p_2 \ (t_{21} \ x_{21}, \ldots, t_{2m} \ x_{2m}) = e_2 \]

\[ \ldots \]

in \textit{body}

\( \textit{tenv}_{\text{implementation}} = [\textit{tid} = t] \textit{tenv} \)

\( \textit{tenv}_i = [x_{i1} = t_{i1}, \ldots] \)

\[ [p_1 = (t_{11} \ast t_{12} \ldots \rightarrow t_1), \]

\[ p_2 = (t_{21} \ast t_{22} \ldots \rightarrow t_2), \]

\[ \ldots ] \textit{tenv}_{\text{implementation}} \]

all types expanded over \( \textit{tenv}_{\text{implementation}} \)
lettype \( \text{tid} = t \)

\[
\begin{align*}
  t_1 \ p_1 \ (t_{11} \ x_{11}, \ldots, t_{1n} \ x_{1n}) &= e_1 \\
  t_2 \ p_2 \ (t_{21} \ x_{21}, \ldots, t_{2m} \ x_{2m}) &= e_2 \\
  \ldots
\end{align*}
\]

in \( \text{body} \)

\[
\text{tenv}_{\text{client}} = [\text{tid} = \text{fresh}]\text{tenv}
\]

\[
\text{tenv}_{\text{body}} = [p_1 = (t_{11} * t_{12} \ldots \rightarrow t_1), \\
p_2 = (t_{21} * t_{22} \ldots \rightarrow t_2), \\
\ldots]\text{tenv}_{\text{client}}
\]

all types expanded over \( \text{tenv}_{\text{client}} \)
Expansions

Types have to be expanded over the environment to which they are then added:

```scheme
(define extend-tenv-with-typedef-exp
  (lambda (typename texp tenv)
    (typedef-record typename
      (expand-type-expression texp tenv) tenv
    ) ) )

(define extend-tenv-with-type-exps
  (lambda (ids texps tenv)
    (extended-tenv-record ids
      (expand-type-expressions texps tenv) tenv
    ) ) )
```
For the client, a type definition is bound to a totally new atomic type.

```
(define fresh-type
  (let ((counter 0))
    (lambda (s)
      (set! counter (+ counter 1))
      (atomic-type
        (string->symbol
          (string-append
            (symbol->string s)
            (number->string counter)
          )
        )
      )
    )
  )
)
(define type-of-lettype-exp
  (lambda (name texp
             resultts names argtss idss bodies body tenv)
    (let* ((the-new-type (fresh-type type-name))
           (rhs-texps (map proc-type-exp argtss resultts))
           (tenv-for-implementation
            (extend-tenv-with-typedef-exp name texp tenv))
           (tenv-for-client
            (extend-tenv-with-typedef
             name the-new-type tenv))
           (tenv-for-proc
            (extend-tenv-with-type-exps
             names rhs-texps tenv-for-implementation))
           (tenv-for-body
            (extend-tenv-with-type-exps
             names rhs-texps tenv-for-client))
    ) ..)
type-of-lettype-exp (2)

(define type-of-lettype-exp
  (lambda (name texp
    resultts names argtss idss bodies body tenv)
  (let* ( .. )
    (for-each
      (lambda (ids argts body resultt)
        (check-equal-type!
          (type-of-expression
            body
            (extend-tenv-with-type-exps
              ids argts tenv-for-proc)
          )
        (expand-type-expression resulttt tenv-for-proc)
        body
        )
      )
      idss argtss bodies resultts)
  (type-of-expression body tenv-for-body)
)) )

How to execute (1)

$ scheme48 -I eopl.image
> ,load ../3/circular-environment.scm
> ,load 4.3.scm
> ,load ../3/boilerplate.scm
> ,load boilerplate.scm
> (type-check "
    lettype myint = int
      myint zero () = 1
      myint succ (myint x) = add1(x)
      myint pred (myint x) = sub1(x)
    bool iszero? (myint x) = zero?((sub1(x))
    in (succ (zero))"
')
'myint1

$ cd 4; make 4.3
> (type-check "
 lettype myint = int
   myint zero () = 1
   myint succ (myint x) = add1(x)
   myint pred (myint x) = sub1(x)
   bool iszero? (myint x) = zero?(sub1(x))
in add1((zero))
")

Error reported by check-equal-type!:
Types didn't match: myint2 != int in
(app-exp (var-exp zero) ())
How to execute (3)

> (type-check "
  lettype ff = (int -> int)
    ff zero-ff () = proc (int k) 0
    ff extend-ff (int k, int val, ff old-ff) =
      proc (int k1) if zero?(-(k1,k))
        then val
        else (apply-ff old-ff k1)
    int apply-ff (ff f, int k) = (f k)
  in let ff1 = (extend-ff 1 11
                (extend-ff 2 22 (zero-ff))
  in (apply-ff ff1 2)
"
'int
How to execute (4)

> (type-check "

lettype ff = (int -> int)
    ff zero-ff () = proc (int k) 0
    ff extend-ff (int k, int val, ff old-ff) =
        proc (int k1) if zero?(-(k1,k))
            then val
        else (apply-ff old-ff k1)
    int apply-ff (ff f, int k) = (f k)
in let ff1 = (extend-ff 1 11

    (extend-ff 2 22 (zero-ff)))
in (ff1 2)
"

Error reported by type-of-expression:
Rator not a proc type:
(var-exp ff1) had rator type ff4
Types

Programming Language Essentials
2nd edition
Chapter 4.4 Type Inference
Changed Language (1)

program: expr

expr: number | id | 'true' | 'false'
    | primitive '(', expr *, ')'
    | 'if' expr 'then' expr 'else' expr
    | 'let' (id '=' expr)*) 'in' expr
    | 'proc' '(' (opt-type-exp id)/',', ')' expr

proc-exp (opt-arg-texps ids body)

| '(' expr expr* ')
| 'letrec' ( opt-type-exp id
    | '(', (opt-type-exp id)/',', ') 'id' expr *)
    | 'in' expr

letrec-exp (opt-resultts
    names opt-argtss idss bodies
    body)
lettype and type-exp for procedures do not permit optional type expressions.
Usage (1)

letrec
  ? even(? odd, ? x) =
    if zero?(x) then 1 else (odd sub1(x))
  in letrec
  ? odd(? x) =
    if zero?(x) then 0
    else (even odd sub1(x))
  in (odd 13)

checker returns int and infers the rest.
Usage (2)

letrec
  ? even(? odd, int x) =
    if zero?(x) then 1 else (odd sub1(x))
in letrec
  bool odd(? x) =
    if zero?(x) then 0
    else (even odd sub1(x))
in (odd 13)

user can fill in some types — correctly.
Type Variables

```
(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?)))
```

container either contains ‘()` or the type once it is known.
(define expand-optional-type-expression
  (lambda (otexp tenv)
    (cases optional-type-exp otexp
      (no-type-exp () (fresh-tvar))
      (a-type-exp (texp)
        (expand-type-expression texp texp tenv))
    )))

replaces expand-type-expression as required by grammar.
(define fresh-tvar
  (let ((serial-number 0))
    (lambda ()
      (set! serial-number (+ 1 serial-number))
      (tvar-type serial-number (vector '())))))

(define tvar->contents
  (lambda (ty)
    (vector-ref (tvar-type->container ty) 0)))

(define tvar-set-contents!
  (lambda (ty val)
    (vector-set! (tvar-type->container ty) 0 val)))
tvar

(define tvar-non-empty?
  (lambda (ty)
    (not (null?
      (vector-ref (tvar-type->container ty) 0))))))

(define tvar-type->container
  (lambda (ty)
    (cases type ty
      (tvar-type (sn vec) vec)
      (else (error 'tvar-type->container
        "Not a tvar-type: ~s" ty)))))
type-of-application

(define type-of-application
  (lambda (rator-type actual-types rator rands exp)
    (let ((result-type (fresh-tvar)))
      (check-equal-type!
        rator-type
        (proc-type actual-types result-type)
        exp
        result-type)))

check-equal-type tries to unify the arguments by setting the tvar.
check-equal-type!

1. if arguments are equal, done.
2. if one argument is `tvar` call `check-tvar-equal-type`!
3. if arguments are atomic types they must be equal.
4. if arguments are procedures they must accept the same number of arguments. Recurse on all types.
5. error.
check-tvar-equal-type!

1. if tvar contains type, recurse and set equal.
2. if tvar is empty, avoid cycles and set equal.
How to execute

$ scheme48 -I eopl.image
> ,load ../3/circular-environment.scm
> ,load 4.4.scm
> ,load ../3/boilerplate.scm
> ,load boilerplate.scm
> (type-check "
   letrec
     ? even(? odd, ? x) =
       if zero?(x) then 1 else (odd sub1(x))
   in letrec
     ? odd(bool x) =
       if zero?(x) then 0 else (even odd sub1(x))
   in (odd 13)
"

Type mismatch: int doesn't match bool in (primapp-exp (zero-test-prim) ((var-exp x)))

$ cd 4; make 4.4