

Implementing Type Checking

Declaring types:

Integers

`<type-exp> ::= int`

Booleans

`<type-exp> ::= bool`

Functions

`<type-exp> ::= ({<type-exp>}*,(*) -> <type-exp>)`

Types of errors:

1. Applying an integer or boolean to an argument.
2. Applying a procedure to the wrong number of arguments.
3. Applying a primitive expecting an integer to a non-integer.
4. Using a non-boolean as the test in a conditional expression.

The Language

As used previously:

Expressions: `lit-exp`, `var-exp`, `if-exp`, `let-exp`, `app-exp`, and `primapp-exp`

Primitives: `+`, `-`, `*`, `add1`, `sub1`, `zero?`

Add or replace:

```
<expression> ::= proc ({<type-exp> <identifier>}*(,)) <expression>
```

```
<expression> ::= letrec {  
    <type-exp> <identifier>  
    ({<type-exp> <identifier>}*(,)) = <expression>  
}*(,)  
in <expression>
```

```
<expression> ::= true
```

```
<expression> ::= false
```

Type Environments

When checking types, we don't care what the values of the variables are, just their types.

We'll use a *type environment* to map variables to types
(just like we used environments to bind variables to values)

- `create-empty-tenv`
- `extend-tenv`
- `apply-tenv`

Determining Types of Expressions:

Number literals are ints:

(type-of-expression \underline{n} *tenv*) = int

Variables—look them up in the type environment:

(type-of-expression \underline{id} *tenv*) = (apply-tenv *tenv* *id*)

Functions:

if (type-of-expression \underline{rator} *tenv*) = ($t_1 * t_2 * \dots t_n \rightarrow t$)

and (type-of-expression $\underline{rand_1}$ *tenv*) = t_1

and (type-of-expression $\underline{rand_2}$ *tenv*) = t_2

...

and (type-of-expression $\underline{rand_n}$ *tenv*) = t_n

then (type-of-expression ($\underline{rator rand_1 rand_2 \dots rand_n}$) *tenv*) = t

If expression: Is the test boolean? Do the two expressions have the same type?

if (type-of-expression $\underline{test-exp}$ *tenv*) = bool

and (type-of-expression $\underline{then-exp}$ *tenv*) = t

and (type-of-expression $\underline{else-exp}$ *tenv*) = t

then (type-of-expression $\underline{if test-exp then then-exp else else-exp}$ *tenv*) = t

Our Internal Type Representation

```
(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))
```

A Few Helper Functions

Expand a *type expressions* (declarations) to a *type*

```
(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)))
```

A Few Helper Functions

Printing out types

```
(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)))))))

(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))))))))
```

The Implementation

```
(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))

      (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))

      ...
```

The Implementation (cont'd)

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

      (proc-exp (texps ids body)
        (type-of-proc-exp texps ids body tenv))

      (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))

      (let-exp (ids rands body)
        (type-of-let-exp ids rands body tenv))

      ...
```

The Implementation (cont'd)

```
(define type-of-expression
  (lambda (exp tenv)
    (cases expression exp
      ...
      (letrec-exp (result-texps proc-names texpss idss bodies
                   letrec-body)
        (type-of-letrec-exp
         result-texps proc-names texpss idss bodies
         letrec-body tenv))
    )))
```

Auxiliary Functions for the Implementation

Map type-of-expression onto a list:

```
(define types-of-expressions
  (lambda (rands tenv)
    (map (lambda (exp) (type-of-expression exp tenv)) rands)))
```

Make sure two types are equal (error if not):

```
(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))))
```

Auxiliary Functions for the Implementation (cont'd)

Type of an application expression—make sure rator and rands match declaration:

```
(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
              (string-append
                "Wrong number of arguments in expression ~s:"
                "~%expected ~s~%got ~s")
                exp
                (map type-to-external-form arg-types)
                (map type-to-external-form rand-types))))))
      (else
        (eopl:error 'type-of-expression
          "Rator not a proc type:~%~s~%had rator type ~s"
          rator (type-to-external-form rator-type))))))
```

Auxiliary Functions for the Implementation (cont'd)

Type of a let expression—build local type environment (local variables) and check the body:

```
(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))))
```

Type of a proc expression—build a local type environment (formals) and check the body:

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

Auxiliary Functions for the Implementation (cont'd)

*Type of a letrec expression—build **recursive** local environment (formals) and check the body:*

```
(define type-of-letrec-exp
  (lambda (result-texps proc-names texps idss bodies
          letrec-body tenv)
    (let ((arg-typess
          (map
            (lambda (texps)
              (expand-type-expressions texps))
            texps))
          (result-types
            (expand-type-expressions result-texps)))
      (let ((the-proc-types
            (map proc-type arg-typess result-types))
            (tenv-for-body ;^ type env for all proc-bodies
              (extend-tenv proc-names the-proc-types tenv)))
        (for-each
          (lambda (ids arg-types body result-type)
            (check-equal-type!
              (type-of-expression
                body
                (extend-tenv ids arg-types tenv-for-body))
              result-type
              body))
            idss arg-typess bodies result-types)
          (type-of-expression letrec-body tenv-for-body))))))
```