

## Type systems

Used to prevent run-time errors  
Specifically, run-time **type errors**

Come in two flavors:

- **Monomorphic** type system
  - Easy to compile to efficient machine code
  - Restrictive for programmers
- **Polymorphic** type system
  - More work to compile to efficient machine code
  - Freedom for programmers—good **reuse**

We study one monomorphic, two polymorphic

## What is a type?

Working definition of type:

- Property a value might have
- Set of values (having the property)
- Or simply a phrase in the type language

Mathematicians are more careful

- Watch out for Russell's Paradox!
- Is "type" a type?

## Examples

Things that are types:

- `int`
- `bool`
- `int * bool`
- `int * int -> int`

## More examples

Things that are not types:

- `list`
- `array`
- `ref (pointer)`
- `int int`

Things that are types:

- `int list`
- `bool array`
- `(int -> int) ref`

## Type constructors

Take zero or more types as arguments, produce type

Nullary type constructors: `int`, `bool`, `char`

- Also called **base types**

Unary type constructors: `list`, `array`, `ref`

Binary type constructor: `->`

More complicated type constructor:  
function in **C** or **Impcore**

## Rules for using constructors

Type formation rules for Typed Impcore

$$\frac{\tau \in \{\text{UNIT}, \text{INT}, \text{BOOL}\}}{\tau \text{ is a type}} \quad (\text{BASETYPES})$$

$$\frac{\tau \text{ is a type}}{\text{ARRAY}(\tau) \text{ is a type}} \quad (\text{ARRAYFORMATION})$$

"Little language" of **type-level expressions**

- New kind of abstract syntax
- Called "types" for short

## Connecting types to values

Some intuition about the meaning of types:

$\llbracket \text{INT} \rrbracket = \{ \text{NUM}(n) \mid n \text{ is an integer} \}$   
 $\llbracket \text{BOOL} \rrbracket = \{ \text{BOOL}(\#t), \text{BOOL}(\#f) \}$   
 $\llbracket \text{SYM} \rrbracket = \{ \text{SYM}(s) \mid s \text{ is a string} \}$   
 $\llbracket \tau_1 \times \dots \times \tau_n \rightarrow \tau \rrbracket = \text{set of functions in } \llbracket \tau_1 \rrbracket \times \dots \times \llbracket \tau_n \rrbracket \rightarrow \llbracket \tau \rrbracket$

## Why do we care about types?

Well-crafted type predicts run-time behavior.

Roughly speaking,

if  $e : \tau$  and  $e \Downarrow v$ , then  $v \in \llbracket \tau \rrbracket$

## Type Soundness

Informally, “well-typed programs don’t go wrong”

## Type rules I — Monomorphic type system

New languages: Typed Impcore and

Typed  $\mu$ Scheme

- Book has both languages
- Lecture notes use only Typed  $\mu$ Scheme
- Begin with monomorphic fragment of Typed  $\mu$ Scheme

## New syntax for a typed language

Explicit types on `define` and `lambda`:

- Argument types for `lambda`  
`(lambda ((int n) (int m)) (+ (* n n) (* m m)))`
- Argument and result types for `define`  
`(define int max ((int x) (int y)) (if (< x y) y x))`

In abstract syntax (ML code):

```

datatype exp = ...
  | LAMBDA of (name * ty) list * exp
  ...

datatype toplevel = ...
  | DEFINE of name * ty * ((name * ty list) * exp)

```

## Type judgments for monomorphic system

Two judgments

- $\tau$  is a type
- $\Gamma \vdash e : \tau$ : in type environment  $\Gamma$ , expression  $e$  has type  $\tau$

Type environment gives type of each variable

## Type soundness again

Full statement of type soundness:

if  $\forall x \in \text{dom } \Gamma : x \in \text{dom } \rho \wedge \rho(x) \in \llbracket \Gamma(x) \rrbracket$ , and if  $\Gamma \vdash e : \tau$ , and if  $\langle \rho, e \rangle \Downarrow v$ , then  $v \in \llbracket \tau \rrbracket$ .

(Ignores mutable locations)

Corollary: a well-typed program written in Typed  $\mu$ Scheme either terminates without error, loops forever, or fails with one of these errors: overflow, divide by zero, `car` or `cdr` of empty list.

## Type rules for variables

### Variable by lookup

$$\frac{x \in \text{dom}\Gamma}{\Gamma \vdash x : \Gamma(x)} \quad (\text{VAR})$$

### Types match in assignment

$$\frac{x \in \text{dom}\Gamma \quad \Gamma(x) = \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \text{SET}(x, e) : \tau} \quad (\text{SET})$$

## Type rules for control

### Boolean condition; matching branches

$$\frac{\Gamma \vdash e_1 : \text{BOOL} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{IF}(e_1, e_2, e_3) : \tau} \quad (\text{IF})$$

### Boolean condition; loop produces no useful result

$$\frac{\Gamma \vdash e_1 : \text{BOOL} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{WHILE}(e_1, e_2) : \text{UNIT}} \quad (\text{WHILE})$$

N.B. Body must be well typed, but type is irrelevant

## Product types: both x and y

### New abstract syntax: PAIR, FST, SND

$$\frac{\tau_1 \text{ and } \tau_2 \text{ are types}}{\tau_1 \times \tau_2 \text{ is a type}} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{PAIR}(e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{FST}(e) : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{SND}(e) : \tau_2}$$

Generalize to **product types** with many elements  
("tuples," "structs," "records")

At run time, identical to **cons**, **car**, **cdr**

## Sum types: either x or y

### New abstract syntax: LEFT, RIGHT, CASE

$$\frac{\tau_1 \text{ and } \tau_2 \text{ are types}}{\tau_1 + \tau_2 \text{ is a type}}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_2 \text{ is a type}}{\Gamma \vdash \text{LEFT}_{\tau_2}(e) : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash e : \tau_2 \quad \tau_1 \text{ is a type}}{\Gamma \vdash \text{RIGHT}_{\tau_1}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma\{x_1 \mapsto \tau_1\} \vdash e_1 : \tau \quad \Gamma\{x_2 \mapsto \tau_2\} \vdash e_2 : \tau}{\Gamma \vdash \text{CASE } e \text{ OF LEFT}(x_1) \Rightarrow e_1 \mid \text{RIGHT}(x_2) \Rightarrow e_2 : \tau}$$

Generalize to "union" of  $n$  alternatives

## Arrow types: function from x to y

### Simulate multi-argument function with tuple

$$\frac{\tau_1, \dots, \tau_n \text{ and } \tau \text{ are types}}{\tau_1 \times \dots \times \tau_n \rightarrow \tau \text{ is a type}} \quad (\text{ARROWFORMATION})$$

### Eliminate with application:

$$\frac{\Gamma \vdash e : \tau_1 \times \dots \times \tau_n \rightarrow \tau \quad \Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n}{\Gamma \vdash \text{APPLY}(e, e_1, \dots, e_n) : \tau}$$

### Introduce with **lambda**:

$$\frac{\Gamma\{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Gamma \vdash \text{LAMBDA}(x_1 : \tau_1, \dots, x_n : \tau_n, e) : \tau_1 \times \dots \times \tau_n \rightarrow \tau}$$

## Array types: array of x

$$\text{Formation: } \frac{\tau \text{ is a type}}{\text{ARRAY}(\tau) \text{ is a type}}$$

$$\text{Introduction: } \frac{\Gamma \vdash e_1 : \text{INT} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{ARRAY-MAKE}(e_1, e_2) : \text{ARRAY}(\tau)}$$

$$\text{Elimination: } \frac{\Gamma \vdash e_1 : \text{ARRAY}(\tau) \quad \Gamma \vdash e_2 : \text{INT}}{\Gamma \vdash \text{ARRAY-GET}(e_1, e_2) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \text{ARRAY}(\tau) \quad \Gamma \vdash e_2 : \text{INT} \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{ARRAY-SET}(e_1, e_2, e_3) : \tau}$$

$$\frac{\Gamma \vdash e : \text{ARRAY}(\tau)}{\Gamma \vdash \text{ARRAY-LENGTH}(e) : \text{INT}}$$

## Type checking

**Problem:** given  $\Gamma$  and  $e$ , find  $\tau$  such that  $\Gamma \vdash e : \tau$ .

**Solution:** follow type rules

```

fun typeof (e, Gamma) =
  let fun ty (VAR x) = find(x, Gamma)
      | ty (SET (x, e)) =
          let val tau_x = find(x, Gamma)
              val tau_e = ty e
          in if tau_x = tau_e then
              tau_x
            else
              raise TypeError ...
          end
  end

```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

19

## More type-checking code

**Function body checked in new environment:**

```

| ty(LAMBDA (formals, body)) =
  let val Gamma' =
      List.foldl
        (fn ((n, ty), g) => bind(n, ty, g))
        Gamma formals
      val bodytype = typeof(body, Gamma')
      val formaltypes =
          map (fn (n, ty) => ty) formals
    in funtype(formaltypes, bodytype)
    end

```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

20

## Types in a monomorphic language

Every new type constructor requires:

- Special-purpose abstract syntax
- New type rules
- New internal representation for formation rules
- New code in type checker for intro, elim rules
- Repeat proof of type soundness

There is a better way: **polymorphism**

©Copyright 2002 Norman Ramsey. All Rights Reserved.

21

## Code duplication in a monomorphic language

User-defined functions **less powerful than abstract syntax**

Every type requires its own function:

```

(define int lengthI ((list int) l)
  (if (null? l) 0 (+ 1 (lengthI (cdr l)))))
(define int lengthB ((list bool) l)
  (if (null? l) 0 (+ 1 (lengthB (cdr l)))))
(define int lengthS ((list sym) l)
  (if (null? l) 0 (+ 1 (lengthS (cdr l)))))

```

There is a better way: **polymorphism**

©Copyright 2002 Norman Ramsey. All Rights Reserved.

22

## Quantified types

Heart of polymorphism:  $\forall \alpha_1, \dots, \alpha_n. \tau$ .

In Typed  $\mu$ Scheme: (forall ('a1 ... 'an) type)

Two ideas:

- **Type variable** 'a stands for an unknown type
- **Quantified type** (with forall) enables substitution

```

length :  $\forall \alpha. \alpha \text{ list} \rightarrow \text{int}$ 
cons   :  $\forall \alpha. \alpha \times \alpha \text{ list} \rightarrow \alpha \text{ list}$ 
car    :  $\forall \alpha. \alpha \text{ list} \rightarrow \alpha$ 
cdr    :  $\forall \alpha. \alpha \text{ list} \rightarrow \alpha \text{ list}$ 
'()    :  $\forall \alpha. \alpha \text{ list}$ 

```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

23

## Instantiation

Use a quantified type by **substituting for type variables**

```

-> (val length-int (@ length int))
length-int : (function ((list int)) int)
-> (val cons-bool (@ cons bool))
cons-bool : (function (bool (list bool))
              (list bool))
-> (val cdr-sym (@ cdr sym))
cdr-sym : (function ((list sym)) (list sym))
-> (val empty-int (@ '() int))
() : (list int)

```

©Copyright 2002 Norman Ramsey. All Rights Reserved.

24

## Instantiation as substitution

Simply substitute type arguments for type variables

```
-> map
<procedure> :
  (forall ('a 'b)
    (function ((function ('a) 'b)
              (list 'a))
              (list 'b))))
-> (@ map int bool)
<proc> : (function ((function (int) bool)
                    (list int))
              (list bool))
```

© Copyright 2002 Norman Ramsey. All Rights Reserved.

25

## Create polymorphic values at home

Abstract over unknown type using `type-lambda`

Example: polymorphic identity:

```
-> (val id (type-lambda ('a)
                      (lambda (('a x)) x)))
id : (forall ('a) (function ('a) 'a))
type-lambda in term becomes forall in type
Inside (type-lambda ('a) ...), type variable 'a
is legitimate type—nature unknown
```

© Copyright 2002 Norman Ramsey. All Rights Reserved.

26

## More do-it-yourself polymorphism

Example: function composition

```
-> (val o (type-lambda ('a 'b 'c)
                    (lambda (((function ('b) 'c) f)
                              ((function ('a) 'b) g))
                              (lambda (('a x)) (f (g x))))))
o : (forall ('a 'b 'c)
     (function ((function ('b) 'c)
                       (function ('a) 'b))
               (function ('a) 'c))))
```

Or  $o : \forall \alpha, \beta, \gamma. (\beta \rightarrow \gamma) \times (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$

© Copyright 2002 Norman Ramsey. All Rights Reserved.

27

## Type rules for polymorphism

Instantiate by substitution:

$$\frac{\Delta, \Gamma \vdash e : \forall \alpha_1, \dots, \alpha_n. \tau}{\Delta, \Gamma \vdash \text{TYAPPLY}(e, \tau_1, \dots, \tau_n) : \tau[\alpha_1 \mapsto \tau_1, \dots, \alpha_n \mapsto \tau_n]}$$

Introduce by type abstraction:

$$\frac{\Delta\{\alpha_1 \mapsto *, \dots, \alpha_n \mapsto *\}, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \text{TYLAMBDA}(\alpha_1, \dots, \alpha_n, e) : \forall \alpha_1, \dots, \alpha_n. \tau}$$

$\Delta$  is kind environment (remembers  $\alpha_i$ 's are types)

What about type formation?

© Copyright 2002 Norman Ramsey. All Rights Reserved.

28

## Generalizing type formation: Kinds

Each type constructor has a kind:

$$\frac{}{* \text{ is a kind}} \quad (\text{KINDFORMATIONTYPE})$$

$$\frac{\kappa_1, \dots, \kappa_n \text{ are kinds} \quad \kappa \text{ is a kind}}{\kappa_1 \times \dots \times \kappa_n \Rightarrow \kappa \text{ is a kind}} \quad (\text{KINDFORMATIONARROW})$$

Examples: `int :: *`, `list :: *  $\Rightarrow$  *`, `pair :: *  $\times$  *  $\Rightarrow$  *`

© Copyright 2002 Norman Ramsey. All Rights Reserved.

29

## Kinding rules for types

$$\frac{\mu \in \text{dom } \Delta}{\Delta \vdash \text{TYCON}(\mu) :: \Delta(\mu)} \quad (\text{KINDINTROCON})$$

$$\frac{\alpha \in \text{dom } \Delta}{\Delta \vdash \text{TYVAR}(\alpha) :: \Delta(\alpha)} \quad (\text{KINDINTROVAR})$$

$$\frac{\Delta \vdash \tau :: \kappa_1 \times \dots \times \kappa_n \Rightarrow \kappa \quad \Delta \vdash \tau_i :: \kappa_i, \quad 1 \leq i \leq n}{\Delta \vdash \text{CONAPP}(\tau, [\tau_1, \dots, \tau_n]) :: \kappa} \quad (\text{KINDAPP})$$

$$\frac{\Delta\{\alpha \mapsto *\} \vdash \tau :: *}{\Delta \vdash \text{FORALL}(\alpha, \tau) :: *} \quad (\text{KINDALL})$$

© Copyright 2002 Norman Ramsey. All Rights Reserved.

30

## Three environments

- $\Delta$  maps names (of tycons and tyvars) to kinds
- $\Gamma$  maps names (of variables) to types
- $\rho$  maps names (of variables) to values or locations

### New val decl

```
val x = 33
```

### New type decl

```
type 'a queue = 'a list * 'a list
```

### New datatype decl

```
datatype void = VOID of void
```

## Three environments revealed

- $\Delta$  maps names (of tycons and tyvars) to kinds
- $\Gamma$  maps names (of variables) to types
- $\rho$  maps names (of variables) to values or locations

### New val decl modifies $\Gamma, \rho$

```
val x = 33 means  $\Gamma\{x : \text{int}\}, \rho\{x \mapsto 33\}$ 
```

### New type decl modifies $\Delta$

```
type 'a queue = 'a list * 'a list  
means  $\Delta\{\text{queue} :: * \Rightarrow *\}$ 
```

### New datatype decl modifies $\Delta, \Gamma, \rho$

```
datatype void = VOID of void
```

```
means  $\Delta\{\text{void} :: *\}, \Gamma\{\text{VOID} : \text{void} \rightarrow \text{void}\}, \rho\{\text{VOID} \mapsto \langle \text{closure} \rangle\}$ 
```