

## Product types: Both x and y

New abstract syntax: PAIR, FST, SND

$$\begin{array}{c} \tau_1 \text{ and } \tau_2 \text{ are types} \\ \hline \tau_1 \times \tau_2 \text{ is a type} \end{array} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{PAIR}(e_1, e_2) : \tau_1 \times \tau_2}$$
$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{FST}(e) : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{SND}(e) : \tau_2}$$

Pair rules generalize to **product types** with many elements (“tuples,” “structs,” and “records”)

## Sum types: either x or y

New abstract syntax: LEFT, RIGHT, CASE

$\tau_1$  and  $\tau_2$  are types

---

$\tau_1 + \tau_2$  is a type

$\Gamma \vdash e : \tau_1 \quad \tau_2 \text{ is a type}$

---

$\Gamma \vdash \text{LEFT}_{\tau_2}(e) : \tau_1 + \tau_2$

$\Gamma \vdash e : \tau_2 \quad \tau_1 \text{ is a type}$

---

$\Gamma \vdash \text{RIGHT}_{\tau_1}(e) : \tau_1 + \tau_2$

$\Gamma \vdash e : \tau_1 + \tau_2$

$\Gamma\{x_1 \mapsto \tau_1\} \vdash e_1 : \tau \quad \Gamma\{x_2 \mapsto \tau_2\} \vdash e_2 : \tau$

---

$\Gamma \vdash \text{CASE } e \text{ OF LEFT}(x_1) \Rightarrow e_1 \mid \text{RIGHT}(x_2) \Rightarrow e_2 : \tau$

## Array types: Array of x

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

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

## Array types continued

Elimination:

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

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

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

## References (similar to C/C++ pointers)

Your turn! Given

<code>ref <math>\tau</math></code>	<code>REF(<math>\tau</math>)</code>
<code>ref <math>e</math></code>	<code>REF-MAKE(<math>e</math>)</code>
<code>!<math>e</math></code>	<code>REF-GET(<math>e</math>)</code>
<code><math>e1</math> := <math>e2</math></code>	<code>REF-SET(<math>e1, e2</math>)</code>

Write formation, introduction, and elimination rules.

**Wait for it ...**

# Reference Types

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

**Introduction:** 
$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{REF-MAKE}(e) : \text{REF}(\tau)}$$

**Elimination:** 
$$\frac{\Gamma \vdash e : \text{REF}(\tau)}{\Gamma \vdash \text{REF-GET}(e) : \tau}$$

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

# **New types are expensive**

## **Closed world**

- Only a designer can add a new type constructor

**A new type constructor (“array”) requires**

- Special syntax
- New type rules
- New internal representation (type formation)
- New code in type checker (intro, elim)
- New or revised proof of soundness



## Expense of array types

Formation:	$\frac{\tau \text{ is a type}}{\text{ARRAY}(\tau) \text{ is a type}}$
Introduction:	$\frac{\Gamma \vdash e_1 : \text{INT} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{AMAKE}(e_1, e_2) : \text{ARRAY}(\tau)}$
Elimination:	$\frac{\Gamma \vdash e_1 : \text{ARRAY}(\tau) \quad \Gamma \vdash e_2 : \text{INT}}{\Gamma \vdash \text{AAT}(e_1, e_2) : \tau}$
	$\frac{\Gamma \vdash e_1 : \text{ARRAY}(\tau) \quad \Gamma \vdash e_2 : \text{INT} \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{APUT}(e_1, e_2, e_3) : \tau}$
	$\frac{\Gamma \vdash e : \text{ARRAY}(\tau)}{\Gamma \vdash \text{ASIZE}(e) : \text{INT}}$

# Expense for programmers

**Monomorphism leads to code duplication**

**User-defined functions are monomorphic:**

```
(check-function-type swap
  ([array bool] int int -> unit))
(define unit swap ([a : (array bool)]
  [i : int]
  [j : int])
  (begin
    (set tmp      (array-at a i))
    (array-put a i (array-at a j))
    (array-put a j tmp)
    (begin)))
```

## **Idea: Do it all with functions**

**Instead of syntax, use functions!**

- **No new syntax**
- **No new internal representation**
- **No new type rules**
- **One proof of soundness**
- **Programmers can add new types**

**Requires: more expressive function types**

## Better type for a swap function

```
(check-type  
  swap  
  (forall ('a) ([array 'a] int int -> unit)))
```

# 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**

`car` :  $\forall \alpha . \alpha \text{ list} \rightarrow \alpha$

`cdr` :  $\forall \alpha . \alpha \text{ list} \rightarrow \alpha \text{ list}$

`cons` :  $\forall \alpha . \alpha \times \alpha \text{ list} \rightarrow \alpha \text{ list}$

`'()` :  $\forall \alpha . \alpha \text{ list}$

`length` :  $\forall \alpha . \alpha \text{ list} \rightarrow \text{int}$

# 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

```
car      : (forall ('a) ([list 'a] -> 'a))
cdr      : (forall ('a) ([list 'a] -> [list 'a]))
cons     : (forall ('a) ('a [list 'a] -> [list 'a]))
' ()     : (forall ('a) (list 'a))
length  : (forall ('a) ([list 'a] -> int))
```

# Representing quantified types

## Two new alternatives for `tyex`:

```
datatype tyex
  = TYCON    of name                // int
  | CONAPP   of tyex * tyex list    // (list bool)
  | FUNTY    of tyex list * tyex    // (int int -> bool)
  | TYVAR    of name                // 'a
  | FORALL   of name list * tyex    // (forall ('a) ...)
```

# Programming with quantified types

## Substitute for quantified variables: “instantiate”

```
-> length
```

```
<procedure> : (forall ('a) ((list 'a) -> int))
```

```
-> [@ length int]
```

```
<procedure> : ((list int) -> int)
```

```
-> (length ' (1 2 3))
```

```
type error: function is polymorphic; instantiate before a
```

```
-> ([@ length int] ' (1 2 3))
```

```
3 : int
```



## Substitute what you like

-> length

<procedure> : (forall ('a) ((list 'a) -> int))

-> [@ length bool]

<procedure> : ((list bool) -> int)

-> ([@ length bool] ' (#t #f))

2 : int

## More instantiations

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

## Create your own!

Abstract over unknown type using `type-lambda`

```
-> (val id (type-lambda ['a]
                        (lambda ([x : 'a]) x )))
id : (forall ('a) ('a -> 'a))
```

`'a` is **type parameter** (an *unknown type*)

This feature is **parametric polymorphism**

## Polymorphic array swap

```
(check-type swap
  (forall ('a) ([array 'a] int int -> unit)))

(val swap
  (type-lambda ('a)
    (lambda ([a : (array 'a)]
      [i : int]
      [j : int])
      (let ([tmp ([@ Array.at 'a] a i)])
        (begin
          ([@ Array.put 'a] a i ([@ Array.at 'a] a j))
          ([@ Array.put 'a] a j tmp)))))))
```

# Power comes at notational cost

## Function composition

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

```
o : (forall ('a 'b 'c)
      ((('b -> 'c) ('a -> 'b) -> ('a -> 'c))))
```

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

## Instantiate by substitution

$\forall$  elimination:

- Concrete syntax  $(@ \ e \ \tau_1 \ \dots \ \tau_n)$
- Rule (note new judgment form  $\Delta, \Gamma \vdash e : \tau$ ):

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

Substitution is **in the book** as function `tysubst`

(Also in the book: `instantiate`)

## Generalize with type-lambda

∀ introduction:

- **Concrete syntax** (`type-lambda`  $[\alpha_1 \cdots \alpha_n]$   $e$ )
- **Rule (forall introduction):**

$$\frac{\begin{array}{c} \Delta\{\alpha_1 :: *, \dots \alpha_n :: *\}, \Gamma \vdash e : \tau \\ \alpha_i \notin \text{ftv}(\Gamma), \quad 1 \leq i \leq n \end{array}}{\Delta, \Gamma \vdash \text{TYLAMBDA}(\alpha_1, \dots, \alpha_n, e) : \forall \alpha_1, \dots, \alpha_n. \tau}$$

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

## What have we gained?

No more introduction rules:

- Instead, use polymorphic functions

No more elimination rules:

- Instead, use polymorphic functions

But, **we still need formation rules**



# You can't trust code

## User's types not blindly trusted:

```
-> (lambda ([a : array]) (Array.size a))
```

```
type error: used type constructor 'array' as a type
```

```
-> (lambda ([x : (bool int)]) x)
```

```
type error: tried to apply type bool as type constructor
```

```
-> (@ car list)
```

```
type error: instantiated at type constructor 'list', which
```

## How can we know which types are OK?

# Let's classify type constructors

**Is a type:** `int, bool`

`int :: *`

`bool :: *`

**Takes a type (to make a type):** `array, list`

`list :: *  $\Rightarrow$  *`

`array :: *  $\Rightarrow$  *`

These labels are called **kinds**

# Type formation through kinds

Each type constructor has a **kind**, which is either:

- $*$ , or
- $\kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa$

Type constructors of kind  $*$  **classify terms**

`(int :: *, bool :: *)`

Type constructors of arrow kinds are “**types in waiting**”

`( list :: *  $\Rightarrow$  *, array :: *  $\Rightarrow$  *, pair :: *  $\times$  *  $\Rightarrow$  *)`

# The kinding judgment

$\Delta \vdash \tau :: \kappa$  “Type  $\tau$  has kind  $\kappa$ ”

$\Delta \vdash \tau :: *$  Special case: “ $\tau$  is a type” (`asType`)

**Replaces** one-off type-formation rules

*Kind environment*  $\Delta$  tracks type constructor names and kinds.

**Use** `asType` in code!

## Kinding rules for types

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

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

These two rules **replace all formation rules.**

(Check out book functions `kindof` and `asType`)

## Designer's burden reduced

To extend Typed Impcore:

- New syntax
- New type rules
- New internal representation
- New code
- New soundness proof

To extend Typed  $\mu$ Scheme, none of the above! Just

- New functions
- New primitive type constructor in  $\Delta$

You'll do arrays both ways

# Kinds of primitive type constructors

$$\Delta(\text{int}) = *$$

$$\Delta(\text{bool}) = *$$

$$\Delta(\text{list}) = * \Rightarrow *$$

$$\Delta(\text{option}) = * \Rightarrow *$$

$$\Delta(\text{pair}) = * \times * \Rightarrow *$$

# What can a programmer add?

## Typed Impcore:

- **Closed world** (no new types)
- Simple formation rules

## Typed $\mu$ Scheme:

- **Semi-closed world** (new type variables)
- How are types formed (from other types)?

## Standard ML:

- **Open world** (programmers create new types)
- How are types formed (from other types)?



# How ML works: 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 def**

```
val x = 33
```

**New type def**

```
type 'a transformer = 'a -> 'a
```

**New datatype def**

```
datatype color = RED | GREEN | BLUE
```

# 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 def` modifies  $\Gamma, \rho$

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

**New** `type def` modifies  $\Delta$

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

**New** `datatype def` modifies  $\Delta, \Gamma, \rho$

`datatype color = RED | GREEN | BLUE`

means  $\Delta\{\text{color} :: *\}, \Gamma\{\text{RED} : \text{color}, \text{GREEN} : \text{color}, \text{BLUE} : \text{color}\},$   
 $\rho\{\text{RED} \mapsto 0, \text{GREEN} \mapsto 1, \text{BLUE} \mapsto 2\}$

## Exercise: Three environments

```
datatype 'a tree
    = NODE of 'a tree * 'a * 'a tree
    | EMPTY
```

means

$$\Delta\{\text{tree} \mapsto * \Rightarrow *\},$$
$$\Gamma\{\text{NODE} \mapsto \forall 'a. 'a \text{ tree} * 'a * 'a \text{ tree} \rightarrow 'a \text{ tree},$$
$$\text{EMPTY} \mapsto \forall 'a. 'a \text{ tree}\},$$
$$\rho\{\text{NODE} \mapsto \lambda(l, x, r). \dots, \text{EMPTY} \mapsto 1\}$$