Product types: Both x and y

New abstract syntax: PAIR, FST, SND

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_1 + \tau_2 \\ \Gamma\{x_1 \mapsto \tau_1\} \vdash e_1 : \tau \qquad \Gamma\{x_2 \mapsto \tau_2\} \vdash e_2 : \tau\end{array}}{\Gamma \vdash \mathsf{CASE}\ e\ \mathsf{OF}\ \mathsf{LEFT}(x_1) \Rightarrow e_1 \mid \mathsf{RIGHT}(x_2) \Rightarrow e_2 : \tau}$$

Array types: Array of x

Formation:

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

Introduction:

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

Array types continued

Elimination:

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

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

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

References (similar to C/C++ pointers)

Your turn! Given

`ref` τ	REF(τ)
`ref e`	REF-MAKE(e)
`!e`	REF-GET(e)
`e1 := e2`	REF-SET$(e1, e2)$

Write formation, introduction, and elimination rules.

Wait for it . . .

Reference Types

Formation:

$$\frac{\tau \text{ is a type}}{\mathsf{REF}(\tau) \text{ is a type}}$$

Introduction:

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

Elimination:

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

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

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:

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

Introduction:

$$\frac{\Gamma \vdash e_1 : \mathsf{INT} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{AMAKE}(e_1, e_2) : \mathsf{ARRAY}(\tau)}$$

Elimination:

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

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

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

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: $\forall \alpha_1, \ldots, \alpha_n \,.\, \tau$.

In Typed μScheme: `(forall ('a1 ... 'an) type)`

Two ideas:

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

`car`	$: \forall \alpha \,.\, \alpha\ \texttt{list} \to \alpha$
`cdr`	$: \forall \alpha \,.\, \alpha\ \texttt{list} \to \alpha\ \texttt{list}$
`cons`	$: \forall \alpha \,.\, \alpha \times \alpha\ \texttt{list} \to \alpha\ \texttt{list}$
`'()`	$: \forall \alpha \,.\, \alpha\ \texttt{list}$
`length`	$: \forall \alpha \,.\, \alpha\ \texttt{list} \to \texttt{int}$

Quantified types

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

In Typed μScheme: `(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 τ_1 $\cdots$ τ_n`)`
- **Rule (note new judgment form** $\Delta, \Gamma \vdash e : \tau$**):**

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

Substitution is in the book as function `tysubst`

(Also in the book: `instantiate`**)**

Generalize with type-lambda

$\forall$ **introduction:**

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

$$\frac{\Delta\{\alpha_1 :: *, \ldots \alpha_n :: *\}, \Gamma \vdash e : \tau \qquad \alpha_i \notin \mathrm{ftv}(\Gamma), \quad 1 \leq i \leq n}{\Delta, \Gamma \vdash \textsf{TYLAMBDA}(\alpha_1, \ldots, \alpha_n, e) : \forall \alpha_1, \ldots, \alpha_n . \tau}$$

Δ is **kind environment** (remembers α_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', whic
```

How can we know which types are OK?

Let's classify type constructors

Is a type: `int`, `bool`

$$\texttt{int} :: *$$

$$\texttt{bool} :: *$$

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

$$\texttt{list} :: * \Rightarrow *$$

$$\texttt{array} :: * \Rightarrow *$$

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

$(\texttt{int} :: *, \texttt{bool} :: *)$

Type constructors of arrow kinds are "types in waiting"

$(\texttt{list} :: * \Rightarrow *, \texttt{array} :: * \Rightarrow *, \texttt{pair} :: * \times * \Rightarrow *)$

The kinding judgment

$\Delta \vdash \tau :: \kappa$ "Type τ has kind κ"

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

Replaces one-off type-formation rules

Kind environment Δ tracks type constructor names and kinds.

Use `asType` in code!

Kinding rules for types

$$\frac{\mu \in \operatorname{dom} \Delta \qquad \Delta(\mu) = \kappa}{\Delta \vdash \mathsf{TYCON}(\mu) :: \kappa} \;\textsc{KindIntroCon}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau :: \kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa \\ \Delta \vdash \tau_i :: \kappa_i, \qquad 1 \leq i \leq n\end{array}}{\Delta \vdash \mathsf{CONAPP}(\tau, [\tau_1, \ldots, \tau_n]) :: \kappa} \;\textsc{KindApp}$$

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 μScheme, none of the above! Just

- New functions
- New primitive type constructor in Δ

You'll do arrays both ways

Kinds of primitive type constructors

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

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

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

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

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

What can a programmer add?

Typed Impcore:

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

Typed μScheme:

- **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

Δ maps names (of tycons and tyvars) to kinds

Γ maps names (of variables) to types

ρ 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

- Δ maps names (of tycons and tyvars) to **kinds**
- Γ maps names (of variables) to **types**
- ρ maps names (of variables) to **values** or **locations**

New `val` def modifies Γ, ρ

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

New `type` def modifies Δ

```
type 'a transformer = 'a list * 'a list
```

means $\Delta\{\mathit{transformer} :: * \Rightarrow *\}$

New `datatype` def modifies Δ, Γ, ρ

```
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

$\Delta\{\texttt{tree} \mapsto * \Rightarrow *\}$,

$\Gamma\{\texttt{NODE} \mapsto \forall\text{'a . 'a tree * 'a * 'a tree} \rightarrow \text{'a tree},$

$\quad \texttt{EMPTY} \mapsto \forall\text{'a . 'a tree}\}$,

$\rho\{\texttt{NODE} \mapsto \lambda(l, x, r) . \cdots, \texttt{EMPTY} \mapsto 1\}$