Type soundness

If $\Gamma \vdash e : \text{int}$, and if $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$, and if $\Gamma$ and $\rho$ are consistent, then $v$ has the form $\text{NUM } n$ for some $n$

(Must generalize to all types $\tau$)
fun ty (IFX (e1, e2, e3)) = 
  if eqType (ty e1, booltype) then 
    let val (tau2, tau3) = (ty e2, ty e3) 
    in ... YOU FILL IN 1 ... 
    end 
  else 
    ... YOU FILL IN 2 ... 
  end 

| ty (SET (x, e)) = 
  let val tau_x = find (x, Gamma) 
    val tau_e = ty e 
  in ... YOU FILL IN 3 ... 
  end
fun ty (APPLY (f, actuals)) = 
    let val atys = map ty actuals 
    in case ty f 
        of FUNTY (formals, result) =>
            if eqTypes (atys, formals) then 
                ... YOU FILL IN 4 ... 
            else 
                ... YOU FILL IN 5 ... 
                | _ => ... YOU FILL IN 6 ... 
        end
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:

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

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

\[ \frac{\Gamma \vdash e : \text{ARRAY}(\tau)}{\Gamma \vdash \text{ASIZE}(e) : \text{INT}} \]
Monomorphism leads to code duplication

User-defined functions are monomorphic:

```
(check-function-type swap
  ([array bool] int 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)))
```
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 \( \mu \)Scheme: (forall (‘a1 ... ‘an) type)

Two ideas:

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

\[
\begin{align*}
\text{car} & : \forall \alpha . \alpha \text{ list} \rightarrow \alpha \\
\text{cdr} & : \forall \alpha . \alpha \text{ list} \rightarrow \alpha \text{ list} \\
\text{cons} & : \forall \alpha . \alpha \times \alpha \text{ list} \rightarrow \alpha \text{ list} \\
\text{‘}() & : \forall \alpha . \alpha \text{ list} \\
\text{length} & : \forall \alpha . \alpha \text{ list} \rightarrow \text{int}
\end{align*}
\]
Quantified types

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

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`:**

```plaintext
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` → `length`

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

`length` → `[@ length int]`

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

`length '(:1 2 3)`

Type error: function is polymorphic; instantiate before application

`[@ 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)
Abstract over unknown type using \textit{type-lambda}

\[
\rightarrow \text{(val id (type-lambda } [\text{'a}] \\
\quad \text{(lambda } ([x : \text{'a}] \ x ))))
\]

\textit{id} : (forall (\text{'a}) (\text{'a} \rightarrow \text{'a}))

\textit{'a} is type parameter (an \textit{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.at-put 'a] a i ([@ Array.at 'a] a j))
               ([@ Array.at-put 'a] a j tmp))))))
Power comes at notational cost

Function composition

\[ \to (\text{val } o \ (\text{type-lambda } ['a 'b 'c] \ (\lambda ([f : ('b -> 'c)] \ [g : ('a -> 'b)]) \ (\lambda ([x : 'a]) (f (g x))))))) \]

\[ o : (\forall ('a 'b 'c) \ ((('b -> 'c) ('a -> 'b) -> ('a -> 'c))) \]

Aka \[ o : \forall \alpha, \beta, \gamma . (\beta \to \gamma) \times (\alpha \to \beta) \to (\alpha \to \gamma) \]
Instantiate by substitution

\forall \text{ elimination:}
- Concrete syntax \((\mathbin{@} e \, \tau_1 \, \cdots \, \tau_n)\)
- Rule (note new judgment form \(\Delta, \Gamma \vdash e : \tau\)):

\[
\Delta, \Gamma \vdash e : \forall \alpha_1, \ldots, \alpha_n. \tau
\]

\[
\Delta, \Gamma \vdash \text{TYAPPLY}(e, \tau_1, \ldots, \tau_n) : \tau[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]
\]

Substitution is in the book as function \text{tysubst}

(Also in the book: instantiate)
Generalize with type-lambda

\forall introduction:
• Concrete syntax (\texttt{type-lambda [\alpha_1 \cdots \alpha_n] e})
• Rule (forall introduction):

\[ \Delta\{\alpha_1 :: *, \ldots \alpha_n :: *\}, \Gamma \vdash e : \tau \]
\[ \alpha_i \notin \text{ftv}(\Gamma), \quad 1 \leq i \leq n \]
\[ \Delta, \Gamma \vdash \text{TY\text{LAM}BDA}(\alpha_1, \ldots, \alpha_n, e) : \forall \alpha_1, \ldots, \alpha_n. \tau \]

\(\Delta\) is kind environment (remembers \(\alpha_i\)'s are types)
You can’t trust code

Type checking guarantees expressions are OK

What about types?

→ (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

(User’s types not blindly trusted)
When is a type well formed?

A well-formed type has the right kind

• “Types classify terms”
• “Kinds classify types”
What kinds do

Kinds classify type expressions into:

- **types** that classify terms (e.g., int)
- **type constructors** that build types (e.g., list)
- **nonsense** that means nothing (e.g., int int)

Use judgment

\[ \Delta \vdash \tau :: \kappa \]

Where

- \( \kappa \) Roughly, “type” or “type constructor”
- \( \Delta \) Kind of each type constructor, type variable
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 *)
Use kinds to give arities

Examples: int :: *, list :: * ⇒ *, pair :: * × * ⇒ *

Non-Examples: int int and bool × list have no kind because they are nonsense.

*Kinds classify type expressions just as types classify terms*
The kinding judgment

\[ \Delta \vdash \tau :: \kappa \quad \text{“Type } \tau \text{ has kind } \kappa \text{”} \]

\[ \Delta \vdash \tau :: * \quad \text{Special case: “} \tau \text{ is a type” (asType)} \]

Replaces one-off type-formation rules

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

Use asType in code!
Kinding rules for types

\[
\begin{align*}
\mu \in \text{dom} \Delta & \quad \Delta(\mu) = \kappa \\
\Delta \vdash \text{TYCON}(\mu) :: \kappa & \quad \text{KINDINTROCON} \\
\Delta \vdash \tau :: \kappa_1 \times \cdots \times \kappa_n \Rightarrow \kappa & \\
\Delta \vdash \tau_i :: \kappa_i, \quad 1 \leq i \leq n & \\
\Delta \vdash \text{CONAPP}(\tau, [\tau_1, \ldots, \tau_n]) :: \kappa & \quad \text{KINDAPP}
\end{align*}
\]

These two rules replace all formation rules.

(Check out book functions \texttt{kindof} and \texttt{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

\[ \Delta(\text{int}) = * \]

\[ \Delta(\text{bool}) = * \]

\[ \Delta(\text{list}) = * \Rightarrow * \]

\[ \Delta(\text{option}) = * \Rightarrow * \]

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

\[ \Delta(\text{array}) = \text{You fill in} \]

\[ \Delta(\text{unit}) = \text{You fill in} \]
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

```ml
val x = 33
```

New type def

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

New datatype def

```ml
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 Γ{x : int}, ρ{x \mapsto 33}

New type def modifies Δ
  type 'a transformer = 'a list * 'a list
means Δ{transformer :: * \Rightarrow *}

New datatype def modifies Δ, Γ, ρ
  datatype color = RED | GREEN | BLUE
means Δ{color :: *}, Γ{RED : color, GREEN : color, BLUE : color},
      ρ{RED \mapsto 0, GREEN \mapsto 1, BLUE \mapsto 2}
Exercise: Three environments

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

means
\[ \Delta\{\text{tree} \mapsto * \Rightarrow *\}, \]
\[ \Gamma\{\text{NODE} \mapsto \forall 'a . 'a \text{ tree} * 'a * 'a \text{ tree} \rightarrow 'a \text{ tree}, \]
\[ \quad \text{EMPTY} \mapsto \forall 'a . 'a \text{ tree}\}, \]
\[ \rho\{\text{NODE} \mapsto \lambda(l,x,r) \ldots , \text{EMPTY} \mapsto 1\} \]