

Program Failures

Two types of failures:

- **Fail-stop** failures abort the program.
 - Easy to debug
 - Hardware exceptions (divide by 0, dereference nil...)
 - Software checks (array bounds check, function arity check, operator position really has a function...)
- **Fail-continue** failures continue after error.
 - Hard to debug
 - unchecked reference outside array bounds is typical error

Safe Languages

Safe languages have only fail-stop failures.

Untyped λ -calculus is safe since **no** errors.

Scheme is a safe language: price lots of checks at run time.

ML is a safe language: less expressive than Scheme, but fewer checks at run time. Checks at compile time.

Modula-3 is unsafe, but only where marked.

C is unsafe: pointer arithmetic and casts.

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`

Typed lambda calculus

First-order typed lambda calculus:

$exp \Rightarrow var$

- | $\lambda var : type . exp$
- | $exp exp$
- | $const$

$type \Rightarrow base\text{-}type$ (e.g., `int`)

- | $type \rightarrow type$

Type rules:

- type of `var` is the type introduced at binding
- type of $M_1 M_2$, if $M_1 : \tau_1 \rightarrow \tau_2$ and $M_2 : \tau_1$, is τ_2
- type of $\lambda v : \tau_1 . M$, if M has type τ_2 , is $\tau_1 \rightarrow \tau_2$
- type of `const` is some basic type

Type judgments

Two-part judgement: (Assumptions, assertion) means “Given the assumptions, the assertion is correct”. Write with infix symbol:

Assumptions \vdash **assertion**

Type judgments:

Given some assumptions, τ is a well-formed type.

$\Gamma \vdash \tau$ is a type

Given assumptions about the types of all free variables in expression e , the expression has type τ .

$\Gamma \vdash e : \tau$ in type environment Γ , expression e has type τ

Type rules for typed λ -calculus

Variable by lookup

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

Application (function call) arg has right type

$$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau} \quad (\text{APP})$$

Abstraction (function definition) body checks if argument has declared type

$$\frac{\Gamma\{x \mapsto \sigma\} \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma. M) : \sigma \rightarrow \tau} \quad (\text{ABS})$$

Type checking example

For any type σ ,

$$\Gamma\{y \mapsto \sigma\} \vdash ((\lambda x : \sigma. x)y) : \sigma$$

On board...

Actually more rules: need rules to say an environment is well-formed. Can only look up variable in well-formed environment.

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

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

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

“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:

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

Types as sets.

Types as Partial Equivalence Relations (symmetric, transitive) between expressions.

Why do we care about types?

Well-crafted type predicts run-time behavior.

Roughly speaking,

if $e : \tau$ and $e \Downarrow v$, then $v \in [[\tau]]$
 or
 if $e : \tau$ and $e \Downarrow v$, then $v : \tau$

Type Soundness

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

Type rules I — Monomorphic type system

New languages: Typed Impcore and Typed μ Scheme

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

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

Type environment gives type of each variable

Type soundness again

Full statement of type soundness:

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

(Ignores mutable locations)

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

Abbreviations: name or structure?

type rectangularComplex = real * real;
 type polarComplex = real * real;

Are different type names different types?

ML: structure rather than name except in datatype, exception constructors.

C: structure except for structs.

Pascal: name almost everywhere.

datatype rectangularComplex = R of real * real;
 datatype polarComplex = P of real * real;

μ Scheme 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

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

Boolean condition; loop produces no useful result

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

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

Product types: both x and y

New abstract syntax: PAIR, FST, SND

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

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

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

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

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

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

Generalize to “union” of n alternatives

Arrow types: function from x to y

Simulate multi-argument function with tuple

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

Eliminate with application:

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

Introduce with `lambda`:

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

Array types: array of x

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

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

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

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

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

Recursive types

ML datatype not just a sum type
datatype 'a list = nil | :: of 'a * 'a list

recursive definition:

For all types 'a: 'a list = $\mu l . \text{unit} + ('a \times l)$

Type rules:

(unwind)
$$\frac{\Gamma \vdash e : \tau[\mu t. \tau / t]}{\Gamma \vdash e : \mu t. \tau}$$
 (wind)
$$\frac{\Gamma \vdash e : \mu t. \tau}{\Gamma \vdash e : \tau[\mu t. \tau / t]}$$

Untyped lambda calculus is typable with all terms
having type $\mu f. f \rightarrow f$

Type checking

Problem: given Γ and e , find τ 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
```

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
```

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**

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**

Quantified types

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

In Typed μ 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 : $\forall \alpha. \alpha \text{ list} \rightarrow \alpha \text{ list}$

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

Second-order typed lambda-calculus

exp \Rightarrow

var	variable
$\lambda \text{ var} : \text{type} . \text{exp}$	functional abstraction
exp exp	application
const	constant
$\Lambda \alpha . \text{exp}$	type abstraction
exp type	type application

type \Rightarrow

tycon	type constructor: int, bool, ...
tyvar	type variable: α, β, \dots
$\forall \alpha . \text{type}$	
type \rightarrow type	

Second-order typed lambda-calculus

New environment Δ keeps track of tycon, α :

$\Delta = \{int :: *, bool :: *, list :: * \Rightarrow *, \alpha :: *\}$

New rules:

Abstracting over type
$$\frac{\Delta\{\alpha \mapsto *\}, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \Lambda \alpha. e : \forall \alpha. \tau}$$

Applying to type
$$\frac{\Delta, \Gamma \vdash e : \forall \alpha. \tau \quad \Delta \vdash \tau :: *}{\Delta, \Gamma \vdash e \sigma : \tau[\alpha \mapsto \sigma]}$$

Rules for well-formedness of types:

judgements $\Delta \vdash \tau :: *$

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)
```

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))
```

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

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

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:

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

Δ is kind environment (remembers α_i 's are types)

What about type formation?

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: $\text{int} :: *$, $\text{list} :: * \Rightarrow *$, $\text{pair} :: * \times * \Rightarrow *$

Kinding rules for types

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

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

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

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

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 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

- 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 decl modifies Γ, ρ

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

New type decl modifies Δ

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

New datatype decl modifies Δ, Γ, ρ

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

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

Evaluating typed calculi

Theorem (“Erasure”):

Can evaluate typed λ -calculus (1st- and 2nd-order) by **erasing all types and Λ 's**, and evaluating as for untyped λ -calculus

An interpreter would:

- infer types and Λ 's for all variables and functions
- discard types, use eval on untyped syntax