

# Typelang

A language with types

# Overview

- Concepts
  - Types
  - Type system
  - Type inference and type checking
- Typelang (ML: lisp with types)
- Type rules (type checking rules)

# Why do we need types?

```
( define f (lambda (x y) (y x)))
```

Which of the following will throw a dynamic error?

- (f 2 3)
- (f 2 (lambda (x) (+ 1 x)))
- (f '2' (lambda (x) (+ 1 x)))

# What is Type

- **Type** is a property of program constructs such as expressions
- Divide program values into kinds -- it defines a set of values (range of variables) and a set of operations on those values
- Classes in OO languages are one instantiation of the modern notion of the type
  - fields and methods of a Java class are meant to correspond to values and operations

# Type as a Contract or Specification

- Contract between producer and consumer regarding what values to expect
  - Procedure: e.g., parameter should be a function
  - Client that calls the procedure needs to follow
- Specification
  - Ultra lightweight: type
  - Lightweight annotations: JML
  - Full mathematical formalism: Z, alloy

# Why Types

- Abstraction:
  - think in terms of a group of values
- Verification:
  - are we applying the operations correctly?
  - report as compiler warnings
- Documentation
  - this input parameter only can be integers
- Performance
  - language implementation optimize for types

2001

# Using Types to Analyze and Optimize Object-Oriented Programs

Amer Diwan

*University of Colorado at Boulder*

Kathryn S. McKinley

*University of Massachusetts - Amherst*

J. Eliot B. Moss

*University of Massachusetts - Amherst*

# Typed Languages

- **A language is typed**: association between expressions are valid only when the types match; Otherwise, the language is not (weakly) typed.

```
a = 9
b = "9"
c = concatenate(a, b) // produces "99"
d = add(a, b)         // produces 18
```

```
a = 9
b = "9"
c = concatenate( str(a), b)
d = add(a, int(b) )
```



# Typed Languages

- Java: **statically typed languages**, every variable name is bound both
  - to a type (at compile time, by means of a data declaration)
  - to an object.

# Typed Languages

- Python: **dynamically typed languages**
  - In a dynamically typed language, every variable name is (unless it is null) bound only to an object.
  - Names are bound to objects at execution time by means of assignment statements, and it is possible to bind a name to objects of different types during the execution of the program.

# Static and dynamic types

- **Static vs dynamic types**
  - Static: the type inferred at compile time
  - Dynamic: the type inferred at run-time
- **Sound static typing**: Dynamic type of an entity is a subset of Static type of the entity

# Type Systems and Type Rules

- A language type system specifies which operations are valid for which types
- A type system is a collection of rules that assign types to program constructs (more constraints added to checking the validity of the programs, violation of such constraints indicate errors)
- Type rules are defined on the structure of expressions
- Type rules are language specific

# Type Systems

- Purpose
  - Prevent execution errors
  - Determine whether programs are well behaved
- Where are type-systems implemented?
  - Compiler

# Execution Errors

- Execution errors
  - w/ symptoms
    - Illegal instruction faults, illegal memory reference faults
  - w/o symptoms
    - Data corruption

# Typing Rules

- Type rule, typing rule, type checking rules
- The typing rules use very concise notation
- They are very carefully constructed
- Virtually any change in a rule either:
  - Makes the type system unsound (bad programs are accepted as well typed)
  - Makes the type system less usable (perfectly good programs are rejected)

# Type Soundness

- Type soundness? We say a type system is sound if
  - **Well-typed programs are well behaved (free of execution errors)**
- It is a property of the type system
  - Intuitively, a sound type system can correctly predict the type of a variable at runtime
  - There can be many sound type rules, we need to use the most precise ones so it can be useful



# How to Specify a Typing Rule

- Logical rules about types

- *Assert a Fact*

(Fact A)

A

- *Imply: conditional assertion*

(B if A)

$$\frac{A}{B}$$

(C if A and B)

$$\frac{A \quad B}{C}$$

# Type Checking and Type Inferences

- **Type Checking** is the process of verifying fully typed programs
- **Type Inference** is the process of filling in missing type information

# Type Checking

- Type Checking
  - Static checking process to prevent unsafe and ill behaved program from ever running
  - Check if the program confirms to the type rules
- Type Checker
  - Algorithm or a tool that performs type checking
- Ill typed program
  - program has a type error – errors in the code that violates the type rules
- Well typed program
  - program has no type error and will pass the typechecker

# Static Type Checking and Dynamic Type Checking

- Dynamic checking?
  - Enforce good behavior by performing run time checks to rule out forbidden errors
  - Example: LISP
    - No static checking
    - No type system
- Type checking can disallow execution (similar to compile time errors)
- Type checking can block execution (similar to run time errors)

# Type Systems

- Expected properties of type systems?
  - Decidably verifiable
    - There exists a type-checking algorithm to ensure that a program is well behaved
  - Transparent
    - Programmer should be able to predict easily whether a program will type-check
  - Enforceable
    - Type declarations should be statically checked as much as possible otherwise dynamically checked

# Design Decisions for Types

- Should languages be typed?
  - Debatable

This study looked at bugs found in open source Javascript code. Looking through the commit history, they enumerated the bugs that would have been caught if a more strongly typed language (like Typescript) had been used. They found that a strongly typed language would have reduced bugs by 15%.

Does this make you want to avoid Python?

# To Type or Not to Type: Quantifying Detectable Bugs in JavaScript

Leveraging JavaScript project histories, we select a fixed bug and check out the code just prior to the fix. We manually add type annotations to the buggy code and test whether Flow and TypeScript report an error on the buggy code, thereby possibly prompting a developer to fix the bug before its public release. We then report the proportion of bugs on which these type systems reported an error. Evaluating static type systems against public bugs, which have survived testing and review, is conservative: it understates their effectiveness at detecting bugs during private development, not to mention their other benefits such as facilitating code search/completion and serving as documentation. Despite this uneven playing field, our central finding is that both static type systems find an important percentage of public bugs: both Flow 0.30 and TypeScript 2.0 successfully detect 15%!

# Design Decisions for Types

- Should languages be type safe?

"Type safe" usually refers to languages that ensure that an operation is working on the right kind of data at some point before the operation is actually performed. This may be at compile time or at run time.



# Design Decisions for Types

- Should languages be safe?
  - No in reality
  - C is deliberately unsafe
  - Run time checks are too expensive
  - Static analysis cannot always ensure

In assembler and C you can often treat a pointer as an integer, an integer as a string, a 2-byte integer as a 4-byte integer, or series of bytes as a structure. Used correctly, this can offer some performance benefit because data doesn't have to be copied or translated if it can be "re-interpreted" as a different type

# Review the Concepts

# TypeLang

- Grammar
- Type checking rules for
  - Constant
  - Atomic expression
  - Compound expression
  - Variable
  - Let expression
  - Lambda
  - List
  - References

# TypeLang Grammar: “Define”

<i>program</i>	<i>::=</i>	<i>definedecl* exp?</i>	<i>Programs</i>
<i>definedecl</i>	<i>::=</i>	<b>(define</b> <i>identifier</i> : <i>T</i> <i>exp</i> )	<i>Declarations</i>
<i>exp</i>	<i>::=</i>		<i>Expressions</i>
		<i>varexp</i>	<i>Variable expression</i>
		<i>numexp</i>	<i>Number constant</i>
		<i>addexp</i>	<i>Addition</i>
		<i>subexp</i>	<i>Subtraction</i>
		<i>multexp</i>	<i>Multiplication</i>
		<i>divexp</i>	<i>Division</i>
		<i>letexp</i>	<i>Let binding</i>
		<i>lambdaexp</i>	<i>Function creation</i>
		<i>callexp</i>	<i>Function Call</i>
		<i>letrecexp</i>	<i>Letrec</i>
		<i>refexp</i>	<i>Reference</i>
		<i>derefexp</i>	<i>Dereference</i>
		<i>assignexp</i>	<i>Assignment</i>
		<i>freeexp</i>	<i>Free</i>

# TypeLang: T

$T ::=$		<i>Types</i>
unit		<i>Unit Type</i>
num		<i>Number Type</i>
bool		<i>Boolean Type</i>
( $T^* \rightarrow T$ )		<i>Function Type</i>
Ref $T$		<i>Reference Type</i>

- Base Type
- Recursively-defined types, i.e. their definition makes use of other types: reference, function types

# Examples of Typelang: which one is correct, which one is incorrect?

- `$ (define pi : num 3.14159265359)`
- `$ (define r : Ref num(ref : num 2))`
- `$ (define u : unit (free (ref : num 2)))`
- `$ (define iden : (num -> num) (lambda (x : num) x))`
- `$ (define id : (num -> num) (lambda (x : (num -> num)) x))`
- `$ (define fi : num "Hello")`
- `$ (define f : Ref num(ref : bool #f))`
- `$ (define t : unit (free (ref : bool #t)))`

# TypeChecking Rules for Constant

- TypeLang assert that all numeric values (constants) have type **num**

- Notation:

(Num)  
n : **num**

- Parts:
  - Name of the rules
  - Before ‘:’ expression or program
  - After ‘:’ type
- Reads: n has a type of **num**

# TypeChecking Rules for Constant

- TypeLang assert that all Boolean values (constants) have type **bool**

- Notation:

(Num)

$n : \mathbf{bool}$

- Reads:  $n$  has a type of **bool**



# TypeChecking Rules for Atomic Expressions

- Atomic: no subexpressions  
(NumExp)

(NumExp n) : **num**

Producer: produce the expression – promise to produce type **num**

Consumer: use this expression – expect type **num**

# Type Environment

- **Typing environment** (or type environment)
  - $\text{tenv} \vdash e:T$  should be read as assuming the type environment  $\text{tenv}$ , the expression  $e$  has type  $t$
  - A record of the types of variables during the processing of program fragments
  - In compiler it is just a symbol table

# Type environment

- Notations
  - $M : A$  *has-type*
  - $A <: B$  *subtype-of*
  - $A = B$  *type equivalence*

# Type Environment

- Value environment (in Varlang)
  - What should be the value of a variable  $x$ ?
- Similarly, type environment
  - Defines the surrounding context
  - What should be the type of a variable?
  - Map that provides operation to lookup the type

# TypeChecking for Compound Expressions

- Conditional assertion: if subexpressions of the addition expression always produce values of type **num**, then the addition expression will produce a value of type **num**.

$$\begin{array}{c} (\text{ADDEXP}) \\ \hline \text{tenv} \vdash e_i : \text{num}, \forall i \in 1..n \\ \hline \text{tenv} \vdash (\text{AddExp } e_0 \ e_1 \ \dots \ e_n) : \text{num} \end{array}$$

- if subexpressions  $e_0$  to  $e_n$  have type **num**, then the expression  $(\text{AddExp } e_0, e_1, \dots, e_n)$  will have type **num** also

# TypeChecking for Compound Expressions

- This typechecking rule establishes a contract between producers of values in this context (expressions  $e_0, e_1, \dots, e_n$ ) and the consumer of these values (the addition expression).
- It also clearly states the conditions under which the addition expression is going to produce a numerical value.
- Notice that the rule does not mention situations where expressions  $e_0, e_1, \dots, e_n$  might produce a dynamic error. If expressions  $e_0, e_1, \dots, e_n$  fail to produce a numerical value the addition expression provides no guarantees.

# TypeChecking for Compound Expressions

- Conditional assertion: if subexpressions of the addition expression always produce values of type **num**, then the addition expression will produce a value of type **num**.

$$\begin{array}{c} (\text{ADDEXP}) \\ \frac{tenv \vdash e_i : \text{num}, \forall i \in 1..n}{tenv \vdash (\text{AddExp } e_0 \ e_1 \ \dots \ e_n) : \text{num}} \end{array}$$

- if subexpressions  $e_0$  to  $e_n$  have type **num**, then the expression  $(\text{AddExp } e_0, e_1, \dots, e_n)$  will have type **num** also

# TypeChecking for Compound Expressions

- This typechecking rule establishes a contract between producers of values in this context (expressions  $e_0, e_1, \dots, e_n$ ) and the consumer of these values (the addition expression).
- It also clearly states the conditions under which the addition expression is going to produce a numerical value.
- Notice that the rule does not mention situations where expressions  $e_0, e_1, \dots, e_n$  might produce a dynamic error. If expressions  $e_0, e_1, \dots, e_n$  fail to produce a numerical value the addition expression provides no guarantees.



# Typing MultExp, SubExp, and DivExp

(MULTEXP)

$$\frac{tenv \vdash e_i : \text{num}, \forall i \in 1..n}{tenv \vdash (\text{MultExp } e_0 \ e_1 \ \dots \ e_n) : \text{num}}$$

(SUBEXP)

$$\frac{tenv \vdash e_i : \text{num}, \forall i \in 1..n}{tenv \vdash (\text{SubExp } e_0 \ e_1 \ \dots \ e_n) : \text{num}}$$

(DIVEXP)

$$\frac{tenv \vdash e_i : \text{num}, \forall i \in 1..n}{tenv \vdash (\text{DivExp } e_0 \ e_1 \ \dots \ e_n) : \text{num}}$$

# Division Rethink

This is an example of a situation where the type system being developed is insufficient to detect and remove certain classes of errors, e.g. the divide-by-zero errors. How to modify it?

$$\frac{\text{(DIVEXP)} \quad tenv \vdash e_i : \mathbf{num}, \forall i \in 1..n}{tenv \vdash (DivExp \ e_0 \ e_1 \ \dots \ e_n) : \mathbf{num}}$$

# Variable typing

- What should be the type of a variable expression  $x$ ?
- What should be a typechecking rule for a variable expression?

w.r.t **Type Environment**

# Type Environment

$$\text{get}(\text{tenv}, v') = \begin{cases} \text{Error} & \text{tenv} = (\text{EmptyEnv}) \\ t & \text{tenv} = (\text{ExtendEnv } v \ t \ \text{tenv}') \\ & \text{and } v = v' \\ \text{get}(\text{tenv}', v') & \text{Otherwise.} \end{cases}$$

# Typechecking rule for VarExp

$$\begin{array}{c} (\text{VAREXP}) \\ \dfrac{\text{get}(\text{tenv}, \text{var}) = t}{\text{tenv} \vdash (\text{VarExp } \text{var}) : t} \end{array}$$

# Type Environment Continue

(NUM)  
 $tenv \vdash n : \text{num}$

(NUM)  
 $tenv \vdash b : \text{bool}$

(NUMEXP)  
 $tenv \vdash (\text{NumExp } n) : \text{num}$

The type environment doesn't play a major role because values produced (and thus types) of these expressions are not dependent on the context.

# Type Environment Continue

$$\begin{array}{c} \text{(ADDEXP)} \\ \frac{tenv \vdash e_i : \text{num}, \forall i \in 1..n}{tenv \vdash (\text{AddExp } e_0 \ e_1 \ \dots \ e_n) : \text{num}} \end{array}$$

- type environment used to perform typechecking of subexpressions is the same as that of the addition expression
- variables and their types stored in the type environment are not affected by the addition expression

# Typing LetExp

$letexp ::= (\text{let } ((identifier : T \text{ exp})^+) \text{ exp})$  *Let expression*

- Typelang requires that programmer specify types of identifiers in let expression

```
(let  
  ((x : num 2))  
  x  
)
```



# Typing LetExp

```
(let  
  ((x : num 2)  
    (y : num 5))  
  (+ x y)  
)
```

- Declares two variables x and y with values 2 and 5
- Establishes contract b/w producers of these values and consumer (+ x y)
- The consumer (+ x y) relies on typechecking rule AddExp

```
(let  
  ((x : num 2)  
    (y : bool #t))  
  (+ x y)  
)
```

- This variable fails to typecheck
- AddExp cannot add a number and a boolean

# Typing LetExp

(LETEXP)

$$\frac{\begin{array}{l} \text{tenv} \vdash e_i : t_i, \forall i \in 0..n \\ \text{tenv}_n = (\text{ExtendEnv } \text{var}_n \ t_n \ \text{tenv}_{n-1}) \ \dots \\ \text{tenv}_0 = (\text{ExtendEnv } \text{var}_0 \ t_0 \ \text{tenv}) \\ \text{tenv}_n \vdash e_{\text{body}} : t \end{array}}{\text{tenv} \vdash (\text{LetExp } \text{var}_0, \dots, \text{var}_n, t_0, \dots, t_n, e_0, \dots, e_n, e_{\text{body}}) : t}$$

# Typing Lambda – Function and Calls

- Type for a function?
  - Contract b/w
    - body (consumer of parameter values and producer of the result value)
  - and
    - Caller (producers of parameter values and consumer of the result value)

# Typing Lambda – Function and Calls

$\text{lambdaexp} ::= (\text{lambda } (\{ \text{identifier} : T \}^*) \text{exp}) \quad \text{Lambda}$

```
(lambda
  (
    x : num    //Argument 1
    y : num    //Argument 2
    z : num    //Argument 3
  )
  (+ x (+ y z))
)
```

- Declares a function with three arguments x, y and z
- Type for this function is,
- $\text{num num num} \rightarrow \text{num}$
- Return type is num as well
- Type checks!

```
(
  (lambda
    (
      x : num    //Argument 1
      y : num    //Argument 2
      z : num    //Argument 3
    )
    (+ x (+ y z))
  )
  1 2 3
)
```

- Declares the same function and also calls it by passing integer parameters 1, 2 and 3 for arguments x, y and z
- Type checks!

# Typing Lambda – Function and Calls

```
(  
  (lambda  
    (  
      x : num    //Argument 1  
      y : num    //Argument 2  
      z : num    //Argument 3  
    )  
    (+ x (+ y z))  
  )  
1 2 #t  
)
```

- Won't typecheck
- #t is of bool type not a num type

# Typing Lambda – Function and Calls

(LAMBDAEXP)

$$\frac{\begin{array}{l} \text{tenv}_n = (\text{ExtendEnv } \text{var}_n \ t_n \ \text{tenv}_{n-1}) \dots \\ \text{tenv}_0 = (\text{ExtendEnv } \text{var}_0 \ t_0 \ \text{tenv}) \quad \text{tenv}_n \vdash e_{\text{body}} : t \end{array}}{\text{tenv} \vdash (\text{LambdaExp } \text{var}_0 \ \dots \ \text{var}_n, t_0 \ \dots \ t_n, e_{\text{body}}) : (t_0 \ \dots \ t_n \multimap t)}$$

(CALLEXP)

$$\frac{\text{tenv} \vdash e_{\text{op}} : (t_0 \ \dots \ t_n \multimap t) \quad \text{tenv} \vdash e_i : t_i, \forall i \in 0..n}{\text{tenv} \vdash (\text{CallExp } e_{\text{op}} \ e_0 \ \dots \ e_n) : t}$$

# Typing Refs

- `refexp : '(' 'ref' ':' T exp ')'`
- `(ref : num 2)`
  - Allocates a memory location of type number with value 2
- `(ref : Ref num (ref : num 2))`
  - Allocates a memory location of type to a reference which its content is 2.

# Typing Refs

- (let  
  (  
    (r : Ref Ref num (ref : Ref num (ref : num 5)))  
  )  
  (deref (deref r))  
)



# Typing Refs

- (let  
  (  
    (r : Ref Ref num (ref : Ref num (ref : num 5)))  
  )  
  (deref (deref r))  
)
  - Declares r as reference to a reference with value number 5 and evaluation of the program returns 5

# Typing Other Expressions

<i>exp</i>	<i>::=</i>	<i>...</i>	<i>Expressions</i>
		<i>strconst</i>	<i>String constant</i>
		<i>boolconst</i>	<i>Boolean constant</i>
		<i>lessexp</i>	<i>Less</i>
		<i>equalexp</i>	<i>Equal</i>
		<i>greaterexp</i>	<i>Greater</i>
		<i>ifexp</i>	<i>Conditional</i>
		<i>carexp</i>	<i>Car</i>
		<i>cdrexp</i>	<i>Cdr</i>
		<i>consexp</i>	<i>Cons</i>
		<i>listexp</i>	<i>List constructor</i>
		<i>nullexp</i>	<i>Null</i>

# Additional Types

$T ::=$

- | unit
- | num
- | bool
- | (  $T^* \rightarrow T$  )
- | Ref  $T$

*Types*

- Unit Type*
- Number Type*
- Boolean Type*
- Function Type*
- Reference Type*

*type* ::= ...

- | String
- | (  $T, T$  )
- | List  $< T >$

*Types*

- String Type*
- Pair Type*
- List Type*

# Typing Lists

- `listexp : ((' 'list' ':' T exp* '')`
- Examples
  - `(list : num 1 2 3)`
    - Constructs a list with elements 1, 2 and 3 of type number.
    - Typechecks?
  - `(list : num 1 2 #t)`
    - Typechecks?

# Typing Lists

- More examples
  - (null? (cons 1 2))
    - Typechecks?
    - (cons 1 2) : constructs a pair type (num, num)
  - (list : List<num> (list : num 2))
    - Typechecks?
  - (list : List<num> (list : bool #t))
    - Typechecks?

# Summary

- Concept maps: types, type systems, type rules, execution errors
- Typelang:
  - Syntax: variable, list will specify types
  - Semantics:
    - Type Checking Rules: examples, formal notations
- Further Reading: Rajan's Chapter 10, Sebestener Chapter 6