TypeLang: a language with types

April 7, 2021

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 (λ (x y) (y x)))
- Which of the following will throw a dynamic error?
  - (f 2 3) // throw a dynamic error
  - (f 2 (λ (x) (+ 1 x)))
  - (f '2' (λ (x) (+ 1 x)) // throw a dynamic error

What is Type

- **type** is a property of program constructs such as variables and expressions
- Divide program values into kinds: it defines a set of values (range of variables) and a set of operations on these values
- Classes in OO languages define new types
  - 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 this "contact"

- 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

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
```


Figure: Example for not(weakly) typed language


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

Figure: Example for strongly typed language

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**
  - Static: 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
  - Prevent execution errors
  - Determine whether programs are well-behaved

- 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

- Decidably verifiable
  - There exists a type-checking algorithm to automatically ensure that a program is well behaved
- Transparent
  - Programmer should be able to predict easily whether a program will correctly type-check

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 System 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
- There can be many sound type rules, we need to use the most precise ones so it can be useful

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 and Type Inferences

- 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

- Type checking can disallow execution (similar to compile time errors)
- Type checking can block execution (similar to run time errors)
- Dynamic checking?
  - Enforce good behavior by performing run time checks to rule out forbidden errors
  - Example: LISP

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 active 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 (strongly typed language)?
  - 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.

TypeLang

- Grammar
- Adding type declarations for firstly introduced values/variables
  - Let expression
  - Define expression
  - Lambda expression
  - List expressions
  - RefLang expressions
- Type checking rules

TypeLang: T

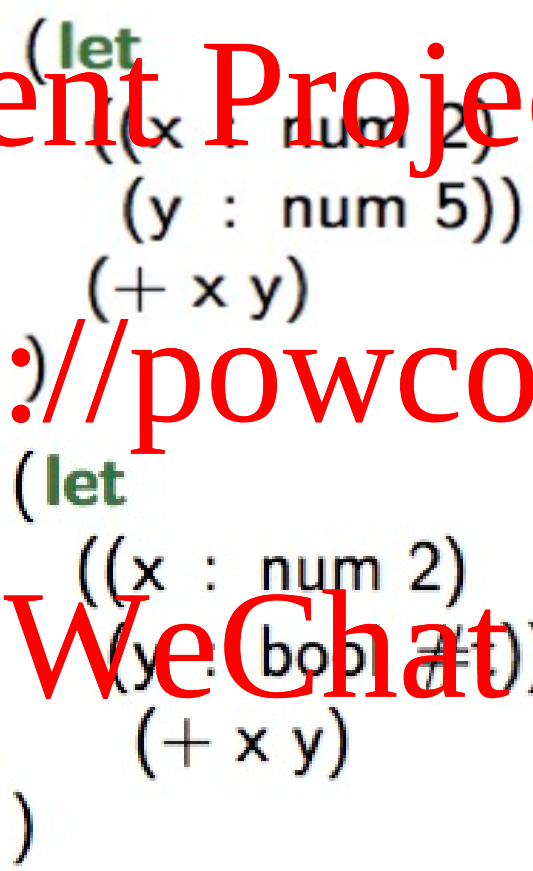
T	::=	*Types*
	unit	*Unit Type*
	num	*Number Type*
	bool	*Boolean Type*
	(T^* -> T)	*Function Type*
	Ref T	*Reference Type*
type	::= ...	*Types*
	String	*String Type*
	(T , T)	*Pair Type*
	List<T>	*List Type*

- Base type
- Recursively-defined types, i.e. their definition makes use of other types

Typing "Define"

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

Typing LetExp



Typing λ-function and calls

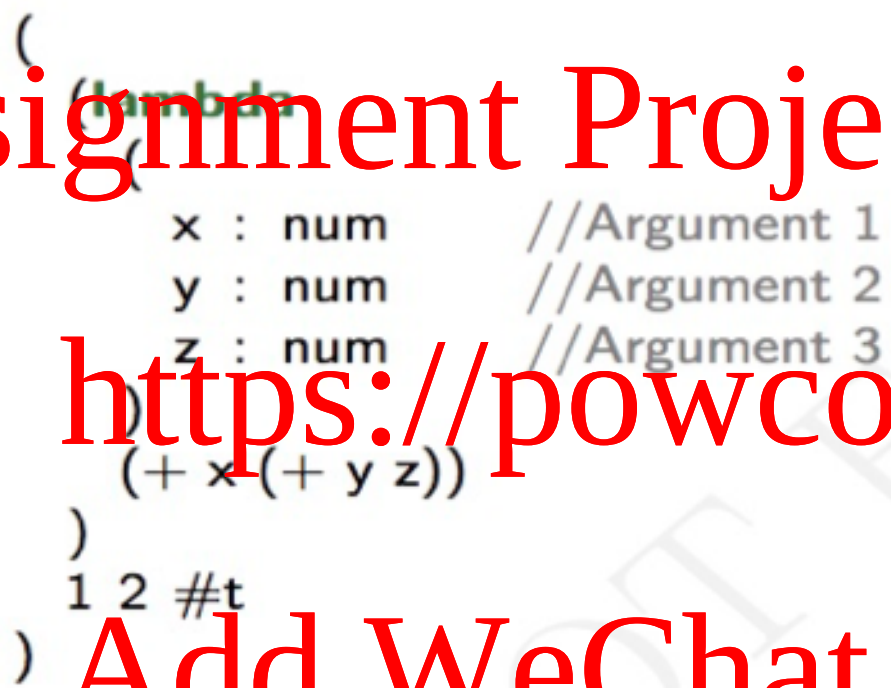
$$lambdaexp \quad ::= \quad (\textbf{lambda}\ (\{identifier\ :\ T\}^*)\ exp) \qquad\qquad Lambda$$


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

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


- Type for this function is, num num num $->$ num
- 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 λ-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 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))
)

- Declares r as reference to a reference with value number 5 and evaluation of the program returns 5

Typing Pair and List

- 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 Pair and List

- 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?

Typing Pair and List

- For a pair expression, do the types for the first and second element has to be same?
- No.

Example Scripts:

```
(cons 1 2)
$ (1 2)
(cons 1 #t)
$ (1 #t)
(cons 1 "hello")
$ (1 "hello")
```

Typing Pair and List

- How about a list where elements are pairs?
- Recall, that elements of the list has to be of the same type.

Example Scripts:


```
(list : (num, bool) (cons 1 #t))
$ ((1 #t))
(list : (num, bool) (cons 1 #t) (cons 2 #f))
$ ((1 #t) (2 #f))
(list : (num, bool) (cons 1 #t) (cons 2 #f) (cons 2 "hello"))
$ Type error: The 2 expression should have type (number bool) found (number stri
```

More Examples of Typelang

- $ (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)) //incorrect
- $ (define fi : num "Hello") //incorrect
- $ (define f : Ref num (ref : bool #f)) //incorrect
- $ (define t : unit (free (ref : bool #t)))

Typing Checking 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)

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

TypeChecking Rules for Constant

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

$$\frac{}{n : \textbf{num}}\ (\text{Num})$$

- 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:

$$\frac{}{\text{(Num) } n : \textbf{bool}}$$

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

Type Checking Rules for Atomic Expressions

- Atomic: no subexpressions

(NumExp)
(NumExp) : **num**

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

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

Type Environment

- **typing environment** (or type environment)
  - tenv |– e:T should be read as assuming the type environment 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

- Value environment (in Varlang)
  - What should be the value of a variable x?

- Similarly, type environment
  - What should be the type of a variable?
  - Map that provides operation to lookup the type

Type rule for constants

$$\frac{}{tenv \vdash n : \texttt{num}} \text{(Num)}$$

$$\frac{}{tenv \vdash b : \texttt{bool}} \text{(Num)}$$

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

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 Checking 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**.

(ADDEXP)

$$\frac{tenv \vdash e_i : \mathbf{num}, \forall i \in 1..n}{tenv \vdash (AddExp\ e_0\ e_1\ \ldots\ e_n) : \mathbf{num}}$$

- if subexpressions e_0 to e_n have type **num**, then the expression (AddExp $e_0, e_1, \ldots, e_n$) will have type **num** as well

Type Checking for Compound Expressions

- This typechecking rule establishes a contract between producers of values in this context (expressions e_0, e_1, ..., 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.
- Note that the rule does not mention situations where expressions e_0, e_1, ..., e_n might produce a dynamic error. If expressions e_0, e_1, ..., e_n fail to produce a numerical value the addition expression provides no guarantees.

Typing MultExp, SubExp, and DivExp

(MultExp)

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

(SubExp)

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

(DivExp)

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

Division Rethink

- This is an example of a situation where the type system being developed is insufficient to detect and remove certain errors, i.e., the divide-by-zero errors

$$\text{(DivExp)}\quad \frac{tenv \vdash e_i : \texttt{num}, \forall i \in 1..n}{tenv \vdash (\mathit{DivExp}\ e\ e_1\ \ldots\ e_n) : \texttt{num}}$$

Variable Typing

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

Type environment

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

Type rule for VarExp

$$\frac{get(tenv, var) = t}{tenv \vdash (VarExp\ var) : t}$$

Type rule for add expression

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

- 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 ::= (let (*identifier* : *T* *exp*) ... *exp*) *Let expression*

Typelang requires that programmer specifies the types of identifiers in let expression.


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

Type rules for LetExpr

(LetExp)

$$\frac{\begin{array}{c} tenv \vdash e_i : t_i, \forall i \in 0..n \\ tenv_n = (ExtendEnv\ var_n\ t_n\ tenv_{n-1}) \ldots \\ tenv_0 = (ExtendEnv\ var_0\ t_0\ tenv) \\ tenv_n \vdash e_{body} : t \end{array}}{tenv \vdash (LetExp\ var_0, \ldots, var_n, t_0, \ldots, t_n, e_0, \ldots, e_n, e_{body}) : t}$$

Typing λ-function and calls

- Body (consumer of parameter values and producer of the result value)
- Caller (producers of parameter values and consumer of the result value)

Typing λ-function and calls

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


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

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


- Type for this function is, num num num $->$ num
- 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 λ-function and calls

(LAMBDAEXP)

$$\frac{tenv_n = (ExtendEnv\ var_n\ t_n\ tenv_{n-1}) \ldots \quad tenv_0 = (ExtendEnv\ var_0\ t_0\ tenv) \quad tenv_n \vdash e_{body} : t}{tenv \vdash (LambdaExp\ var_0 \ldots var_n\ t_0 \ldots t_n\ e_{body}) : (t_0 \ldots t_n -> t)}$$

(CALLEXP)

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

Summary

- Concepts
- Typelang: introduce types for different types of expressions
  - Syntax
  - Semantics: Type Checking Rules: examples, formal notations
- Further Reading: Rajan's Chapter 10, Sebester Chapter 6