Typelang – A language with types

November 7, 2018

Overview

- Concepts
  - Types
  - Type system
  - Type inference
  - Type checking
- Typelang grammar
- Type rules (type checking rules)
  - Constant
  - Variable
  - Arithmetic
  - Compound
  - Let expression
  - Lambda expression
  - Ref
  - List

Why do we need types?

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

(f 2 3)

(f 2 (lambda (x) (+ 1 x))) ✓

(f '2' (lambda (x) (+ 1 x)) ✓

lambda(x)(+ 1 x) '2'

(+ 1 '2')

What is Type

- Type is a property of program constructs such as expressions
- It defines a set of values (range of variables) and a set of operations on those values
- Classes 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

What is Type

- Divide program values into kinds: contract between producer and consumer regarding what values to expect

- Each kind will define:
  - Range of the values
  - Operations on the values

Type as a Contract or Specification

- Contract/specification: agreement between two entities
  - 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?
  - static and compiler errors
- Documentation
  - this input parameter only can be integers
- Performance
  - language implementation optimize for types (python is slow, c can be faster)

Typed Languages

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

- Static vs dynamic types
  - Static: can be inferred at compile time
  - Dynamic: can be inferred at run-time

- Sound static typing: Dynamic type of an entity is a subset of Static type of the entity

reasoning system about types

It is always correct

Fruit
Apple
pear

pig.

Type Systems

- 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)
- A languages type system specifies which operations are valid for which types
- Type systems provide a concise formalization of the semantic checking rules
- Type rules are defined on the structure of expressions
- Type rules are language specific

Type Systems

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

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

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

- Typechecking
  - Static checking process to prevent unsafe and ill behaved program from ever running
  - Check if the program confirms to the type rules
- Typechecker
  - Algorithm or a tool that performs typechecking
- 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)

Typing Rules

- Type rule, typing rule, typechecking 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

Design Decisions for a PL: Safe? Typed?

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

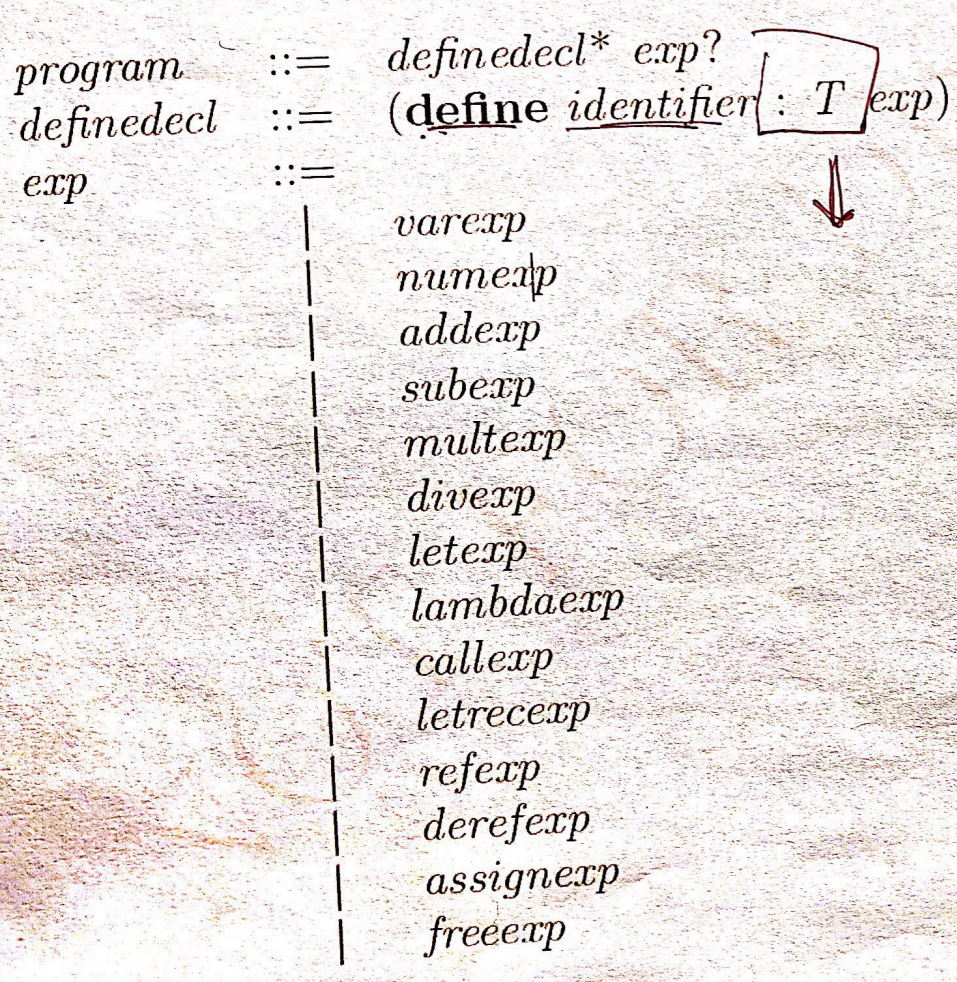
- Should languages be typed?
  - Debatable

Review the Concepts

Typelang

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

TypeLang Grammar: "Define"



Programs
Declarations
Expressions
Variable expression
Number constant
Addition
Subtraction
Multiplication
Division
Let binding
Function creation
Function Call
Letrec
Reference
Dereference
Assignment
Free

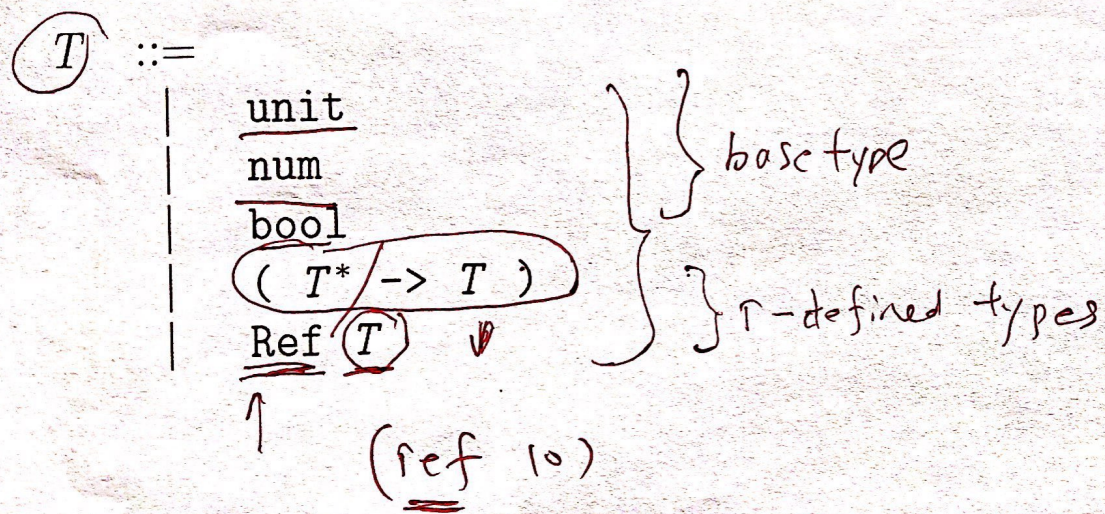
Examples of Typelang: which one is correct, which one is incorrect?

- $ (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))) ✓

(ref : T value)

((num -> num) -> (num -> num))

TypeLang: T



Types
Unit Type
Number Type
Boolean Type
Function Type
Reference Type

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

Additional Types

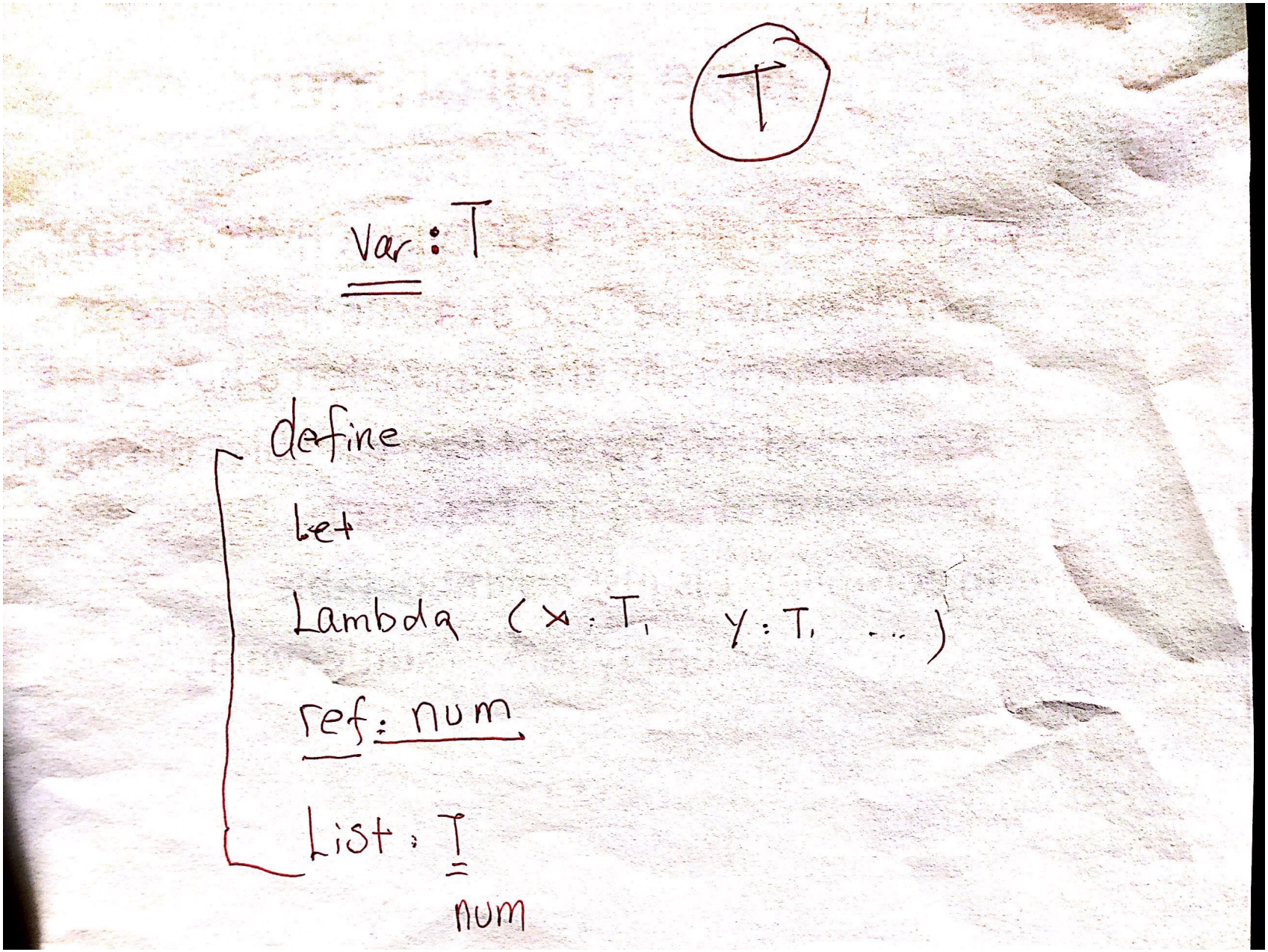
```
T  ::=
     | unit
     | num
     | bool
     | ( T* -> T )
     | Ref T
```

Types
Unit Type
Number Type
Boolean Type
Function Type
Reference Type

```
type ::= ...
       | String
       | ( T , T )
       | List < T >
```

Types
String Type
Pair Type
List Type

Additional Types



How to Specify a Typing Rule

- Logical rules
- Assert a Fact

(Fact A)

A

- Imply: conditional assertion

(B if A)

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

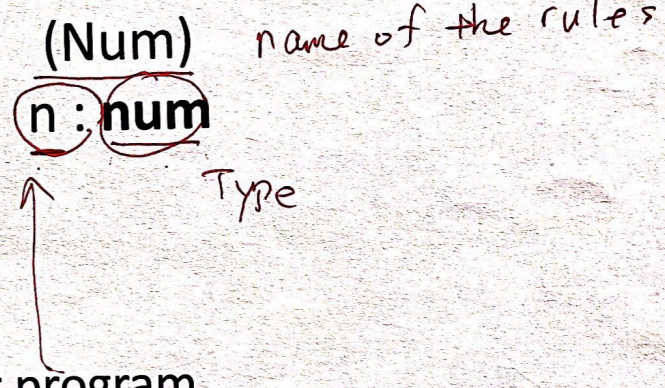
(C if A and B)

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

TypeChecking Rules for Constant

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

- Notation:



- Parts:
  - Name of the rules
  - Before ':' expression or program
  - After ':' type

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

TypeChecking Rules for Atomic Expressions

- Atomic: no subexpressions

(NumExp)

(NumExp n) : **num**

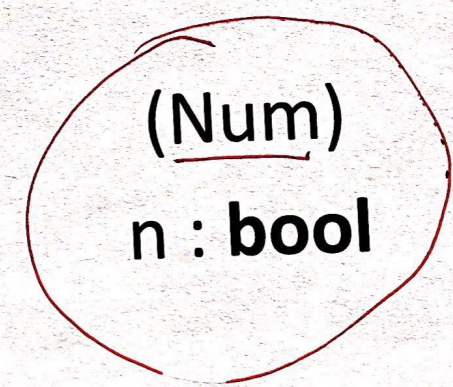
Producer: produce the expression – promise to produce type **num**

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

TypeChecking Rules for Constant

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

- Notation:



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

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

tenv |- M : A

Type Environment

- Typing environment (or type environment)
  - tenv |- should be read as assuming the type environment tenv, the expression e has type t
  - A record of the types of free variables during the processing of program fragments
  - In compiler it is just a symbol table
  - Γ |- M : A, M has type A in static typing environment Γ

Gamma

Type environment

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

Type Environment Continue

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

- 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

Type Environment Continue

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

3

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

#t

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

3

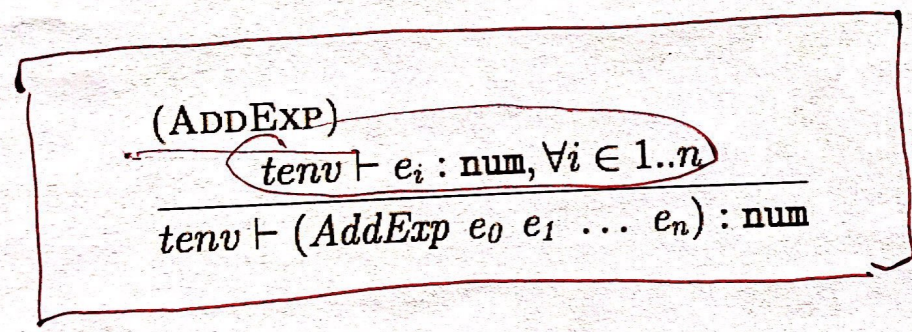
The type environment doesn't play a major role because values produced (and thus types) of these expressions are not dependent on the context.

Context-Sensitivity

Connect dots { Value Environment & Scope
PDA

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**.



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

- if subexpressions e_0 to e_n have type **num**, then the expression (AddExp e_0 , e_1 , . . . , 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, . . . , 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, . . . , 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 classes of errors, e.g. the divide-by-zero errors. How to modify it?

NUM_NonZero

NUM_Zero

✓ (DivExp)

$$\frac{tenv \vdash e_i : \mathbf{num}, \forall i \in 1..n \qquad e_1 \neq 0 \; e_n \neq 0}{tenv \vdash (DivExp\ e_0\ e_1\ \ldots\ e_n) : \mathbf{num}}$$

② $tenv \vdash e_0 : NUM\_Zero$

$e_1 \; \ldots \; e_i : NUM\_NonZero$

$tenv \vdash DivExp(\,) : NUM\_Zero$

① $tenv \vdash e_i : NUM\_NonZero \quad \forall i \in 1..n$

$tenv \vdash (DivExp\ e_0\ \ldots\ e_n) : NUM\_NonZero$

Variable typing

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

x y

w.r.t **Type Environment**

Type Environment

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

tenv = tenv' ∪ (v ↦)

Typechecking rule for VarExp

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

Typing LetExp

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

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

Typing 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

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

```
(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,
- *num num num -> 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)

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

(CALLEXP)

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

t0 t1

Call

(1 2 3)

(defined (lambda (x) (+ x 1)))

(d #t)

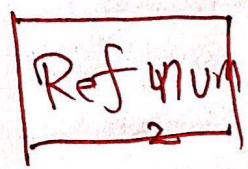
num -> num

((d) 2 3)

num num -> num

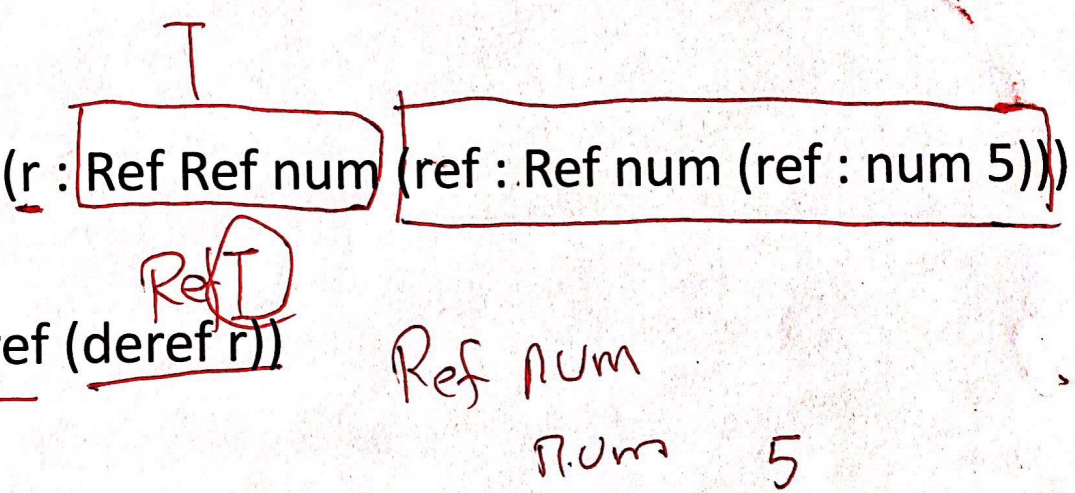
Typing Refs

- 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 Other Expressions

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

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, Sebester Chapter 6