

## Type Systems

- A **type** is a **name** for a well-defined set of values and operations on those values
  - **Examples:**
    - **int** with values  $\{\dots, -2, -1, 0, 1, 2, \dots\}$  and operations  $\{+, -, *, /, \dots\}$
    - **boolean** with values  $\{false, true\}$  and operations  $\{\&\&, //, !\}$
  - A **type system** associates types with variables and other objects in a program
  - **Statically typed languages** associate a single type with a variable throughout the run-time life-span of the variable
  - **Dynamically typed languages** allow types of variables to change during execution
  - **Statically typed languages** have type rules operating on the abstract syntax — “static semantics”
- Type correctness as part of syntax: **context-sensitive!**

## Modular C Programming Example ...

```

/* cube.h */
extern double cube(double x);

-----
/* cube.c */
#include <stdio.h>
double cube(double x) {
    double r = x * x * x;
    printf("cube: %f --> %f\n", x, r);
    return r; }

-----
/* cubing.c */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[]) {
    int k = atoi(argv[1]);
    printf("cubing: %d --> %d\n", k, cube(k));
    return 0;
}

```

## Undetected Run-Time Type Error in C

```

#include <stdio.h>                                // union,c

union utag {int a;
             float p;
             } u;

void main() {
    float x = 2.0;
    u.a = 2135329191;
    printf("%d %f\n", u.a, x + u.p);
}

2135329191    263901874997436424049023275123576143872.000000

```

- interpretation as **float** values is **not well-defined** for **int** values
- interpretation as **float** values is *possible* for **int** values
- **unexpected values can produce undetected misbehaviour**

## Detected Run-Time Type Error in Java

```

class Point { protected double _x, _y;
    public Point(double x, double y) { _x = x; _y = y; }
    public String toString() { return "(" + _x + ", " + _y + ")"; }
}

class Point3 extends Point { protected double _z;
    public Point3(double x, double y, double z) { super(x,y); _z = z; }
    public void up(double dz) { _z += dz; }
    public String toString() { return "(" + _x + ", " + _y + ", " + _z + ")"; }
}

class DownCastError {
    public static void main(String[] args) {
        Point p = new Point( 2.0, 3.0);
        Point q = new Point3( 2.0, 3.0, 4.0);
        ((Point3)q).up( 0.7 ); System.out.println("q = " + q);
        ((Point3)p).up( 0.7 ); System.out.println("p = " + p);
    }
}

q = ( 2.0, 3.0, 4.7)
java.lang.ClassCastException: Point
    at DownCastError.main(DownCastError.java:18)

```

## Static versus Dynamic Typing

- Compile-time type checking: **Static typing**
- Run-time type checking: **Dynamic typing**
- All type errors will be *detected*: **strongly typed** languages
- A program is **type safe** if it is known to be free of type errors
- A language is **type safe** if all its programs are type safe

**Warning:** *Quite some mix-up in the conclusions in the textbook!*

- For every language, all its programs pass all its statical tests:
  - For a dynamically typed language like LISP, at least some programs contain type errors, otherwise no dynamic checking would be necessary: LISP may be strongly typed, but is **not type safe**. (p. 51)
  - Java is strongly typed, but in some aspects **only with dynamic type checks**: Java is **not type safe!** (p. 233)
  - **Haskell is strongly statically typed**, and therefore **type safe!** (p. 233)

**Oberon** type guards correspond to Java down-casts.

## Type Languages

At each point in a program, there is a **type language**  $\mathcal{T}$

- most languages include implementation-oriented **primitive types** like `int`, `bool`, `float`, `char`
- $\mathcal{T}_{\text{Jay}} = \{\text{int}, \text{boolean}\}$
- **type definitions** extend the type language
- **type constructors** produce infinite type languages:
  - **C, Java:** `*`, `[]` only
  - **Haskell:** `_->_`, `[]`, `(_,_)`, `Maybe _`, `IO _`, `Ratio _`, `Array _ _`, and **user-defined type constructors**, e.g.: `Map _ _`, `Set _`, `Graph _`
  - **Java 1.5:** Has “**generics**”, i.e., parametric polymorphism similar to Haskell

## Type System Principles

Assume a constant type language  $\mathcal{T}$ .

- At each point in a program, there is a **type context** (or **typing environment**)  $\Gamma$ , mapping visible identifiers to types.

Usually, this is a (finite) partial function:

$$\Gamma : \text{Identifier} \mapsto \mathcal{T}$$

- Given a type context  $\Gamma$ ,
  - a **declaration**, if **valid**, produces a new type context  $\Gamma'$

- an **expression**  $e$  may **have a type**  $t$

$$\Gamma \vdash e : t$$

(Then,  $e$  is **well-typed**, with type  $t$ )

- a **statement** may be **well-typed**

## Example Jay Program

```
void main () {
// compute result = the factorial of integer n
int n, i, result;
n = 8;
i = 1;
result = 1;
while ( i < n ) {           // (i < n)   : boolean
    i = i + 1;              // (i = 1)   : int
    result = result * i;    // (result * i) : int
}
}
```

## Jay: Extracting the Context from the Declarations

Given a type context  $\Gamma$ , a **declaration**, if **valid**, produces a new type context  $\Gamma'$ .

- Abstract syntax:
 

```
class Declaration { Variable v; Type t; }
class Declarations extends Vector {}
```
- Java type *TypeMap* implements *Identifier*  $\leftrightarrow \mathcal{T}$
- Jay has only one declaration block: can start from empty context:

$$\text{typing} : \text{Declarations} \rightarrow \text{TypeMap}$$

$$\text{typing} (\text{Declarations } d) = \bigcup_{i \in \{1, \dots, n\}} \{d_i.v \mapsto d_i.t\}$$

- *TypeMap* *typing* (*Declarations* *d*) {
 

```
TypeMap map = new TypeMap();
for (int i=0; i<d.size(); i++)
  map.put (((Declaration)(d.elementAt(i))).v,
           ((Declaration)(d.elementAt(i))).t);
return map;
}
```

## Jay in Haskell: Extracting the Context from the Declarations

Given a type context  $\Gamma$ , a **declaration**, if **valid**, produces a new type context  $\Gamma'$ .

- Abstract syntax:
 

```
data Program = MkProgram [Declaration] Block
data Declaration = MkDecl Variable Type
type Declaration' = (Variable, Type)
```
- Haskell type *Map* *Variable* *Type* implements *Identifier*  $\leftrightarrow \mathcal{T}$
- Jay has only one declaration block: can start from empty context:

$$\text{typing} : \text{Declarations} \rightarrow \text{TypeMap}$$

$$\text{typing} (\text{Declarations } d) = \bigcup_{i \in \{1, \dots, n\}} \{d_i.v \mapsto d_i.t\}$$

- *typing* = *Map.fromList* . *map* ( $\backslash$  (*MkDecl* *v* *ty*) -> (*v*,*ty*))
   
-- = *foldr* ( $\lambda$  (*MkDecl* *v* *ty*)  $\rightarrow$  *Map.insert* *v* *ty*) *Map.empty*
  
  
*typing'* = *Map.fromList* -- = *foldr* (*uncurry* *Map.insert*)

## Jay: Checking Validity of Declarations

- **Overloaded validity function**  $V$
- **Validity of declaration block:** Each variable name declared at most once:

$$V : \text{Declarations} \rightarrow \mathbf{B}$$

$$V (\text{Declarations } d) = \forall i, j : \{1, \dots, n\} \bullet (i \neq j \Rightarrow d_i.v \neq d_j.v)$$

- Implementation:

```
public boolean V (Declarations d) {
  for (int i=0; i<d.size() - 1; i++)
    for (int j=i+1; j<d.size(); j++)
      if (((Declaration)(d.elementAt(i))).v.equals
          ((Declaration)(d.elementAt(j))).v)
        return false;
  return true;
}
```

## Jay: Checking Validity of Declarations — Haskell

- **Overloaded validity function**  $V$
- **Validity of declaration block:** Each variable name declared at most once:

$$V : \text{Declarations} \rightarrow \mathbf{B}$$

$$V (\text{Declarations } d) = \forall i, j : \{1, \dots, n\} \bullet (i \neq j \Rightarrow d_i.v \neq d_j.v)$$

- Haskell Implementation: The list of declared variables contains no duplicates:

```
declsValid :: [Declaration] -> Bool
declsValid = noDups . map (\ (MkDecl v ty) -> v)
```

```
noDups :: Eq a => [a] -> Bool
```

```
noDups [] = True
```

```
noDups (x:xs) = (x 'notElem' xs) && noDups xs
```

## Jay Expression Typing Rules — Haskell

```

data Expression = Var Variable
              | Value Literal
              | Binary BinOp Expression Expression
              | Unary UnaryOp Expression
type Variable = String
data Literal = LitInt Int | LitBool Bool
data BinOp = MkBoolOp BoolOp | MkRelOp RelOp | MkArithOp ArithOp

```

- Variables must have been declared with one of the two types `int` and `boolean`
- Arithmetic operators `+`, `-`, `*`, `/` demand two `int` arguments and produce an `int` expression
- Relational operators `==`, `!=`, `<`, `<=`, `>`, `>=` demand two `int` arguments and produce a `boolean` expression
- Boolean operators `&&`, `||` demand two `boolean` arguments and produce a `boolean` expression

## Jay Expression Superficial Type Inference — Haskell

```

typeOfExpr :: TypeMap → Expr → Maybe Type
typeOfExpr tm (Var v) = Map.lookup v tm
typeOfExpr tm (Value lit) = typeOfLit lit
typeOfExpr tm (Binary op e1 e2) = Just $ case op of
  MkBoolOp bOp → BoolType
  MkRelOp rOp → BoolType
  MkArithOp aOp → IntType
typeOfExpr tm (Unary Not e) = Just BoolType

```

**Only inspects top-level construction!**

## Jay Expression Typing Rules — Java

```

class Expression {} // Expression = Variable | Value | Binary | Unary
class Variable extends Expression { String id; }
class Value extends Expression { // Value = int intValue | boolean boolValue
  Type type; int intValue; boolean boolValue; }
class Binary extends Expression {
  Operator op; Expression term1, term2; }
class Operator { String val; }

```

- Variables must have been declared with one of the two types `int` and `boolean`
- Arithmetic operators `+`, `-`, `*`, `/` demand two `int` arguments and produce an `int` expression
- Relational operators `==`, `!=`, `<`, `<=`, `>`, `>=` demand two `int` arguments and produce a `boolean` expression
- Boolean operators `&&`, `||` demand two `boolean` arguments and produce a `boolean` expression

## Jay Expression Superficial Type Inference — Java

```

public Type typeOf (Expression e, TypeMap tm) {
  if (e instanceof Value) return ((Value)e).type;
  if (e instanceof Variable) {
    Variable v = (Variable)e;
    if (!tm.containsKey(v)) return new Type(Type.UNDEFINED);
    else return (Type) tm.get(v);
  }
  if (e instanceof Binary) {
    Binary b = (Binary)e;
    if (b.op.ArithmeticOp()) return new Type(Type.INTEGER);
    if (b.op.RelationalOp() || b.op.BooleanOp())
      return new Type(Type.BOOLEAN);
  }
  if (e instanceof Unary) {
    Unary u = (Unary)e;
    if (u.op.UnaryOp()) return new Type(Type.BOOLEAN);
  }
  return null;
}

```

**Only inspects top-level construction!**

## Jay Expression Type Checking — Haskell

```

exprValid :: TypeMap → Expr → Bool
exprValid tm ( Value lit ) = True
exprValid tm ( Var v ) = v `Map.member` tm
exprValid tm ( Binary op e1 e2 ) = case typeOfExpr tm e1 of
  Nothing → False
  Just ty1 → case typeOfExpr tm e2 of
    Nothing → False
    Just ty2 → exprValid tm e1 && exprValid tm e2 && case op of
      MkBoolOp bOp → case ( ty1, ty2 ) of
        ( BoolType, BoolType ) → True
        _ → False
      _ → case ( ty1, ty2 ) of
        ( IntType, IntType ) → True
        _ → False
typeOfExpr tm ( Unary Not e ) = case typeOfExpr tm e of
  Just BoolType → exprValid tm e
  _ → False

```

## Jay Expression Type Checking — Java

```

public boolean V (Expression e, TypeMap tm) {
  if (e instanceof Value) { return true; }
  if (e instanceof Variable) { return tm.containsKey((Variable)e); }
  if (e instanceof Binary) {
    Type typ1 = typeOf(((Binary)e).term1, tm);
    Type typ2 = typeOf(((Binary)e).term2, tm);
    if (! V (((Binary)e).term1, tm)) return false;
    if (! V (((Binary)e).term2, tm)) return false;
    if (((Binary)e).op.ArithmeticOp( ) || ((Binary)e).op.RelationalOp( ))
      return typ1.isInteger() && typ2.isInteger();
    if (((Binary)e).op.BooleanOp( ))
      return typ1.isBoolean() && typ2.isBoolean();
  }
  if (e instanceof Unary) {
    Type typ1 = typeOf(((Unary)e).term, tm);
    return typ1.isBoolean() && V(((Unary)e).term, tm)
      && (((Unary)e).op.val.equals("!"));
  }
  return false;
}

```

## Jay Expression Type Inference — Haskell

```

typeOfExpr :: TypeMap → Expr → Maybe Type
typeOfExpr tm ( Var v ) = Map.lookup v tm
typeOfExpr tm ( Value lit ) = typeOfLit lit
typeOfExpr tm ( Binary op e1 e2 ) = case typeOfExpr tm e1 of
  Nothing → Nothing
  Just ty1 → case typeOfExpr tm e2 of
    Nothing → Nothing
    Just ty2 → case ( op, ty1, ty2 ) of
      ( MkBoolOp, BoolType, BoolType ) → Just BoolType
      ( MkRelOp, IntType, IntType ) → Just BoolType
      ( MkArithOp, IntType, IntType ) → Just IntType
      _ → Nothing
typeOfExpr tm ( Unary Not e ) = case typeOfExpr tm e of
  Just BoolType → Just BoolType
  _ → Nothing

```

Returns Just ty only if no type errors!

## Jay Expression Type Inference — Maybe Monad

Monads are **computation** concepts — *Maybe* is the concept of “*possibly failing computations*”: Here: `return = Just` `fail s = Nothing`

```

typeOfExpr :: TypeMap → Expr → Maybe Type
typeOfExpr tm ( Var v ) = Map.lookup v tm
typeOfExpr tm ( Value lit ) = typeOfLit lit
typeOfExpr tm ( Binary op e1 e2 ) = do
  ty1 ← typeOfExpr tm e1
  ty2 ← typeOfExpr tm e2 of
  case ( op, ty1, ty2 ) of
    ( MkBoolOp, BoolType, BoolType ) → return BoolType
    ( MkRelOp, IntType, IntType ) → return BoolType
    ( MkArithOp, IntType, IntType ) → return IntType
    _ → fail "type error"
typeOfExpr tm ( Unary Not e ) = do
  BoolType ← typeOfExpr tm e
  return BoolType

```