INF 212/CS 253
Type Systems

Instructors: Harry Xu   Crista Lopes

What is a Data Type?

- A type is a collection of computational entities that share some common property

- Programming languages are designed to help programmers organize computational constructs and use them correctly. Many programming languages organize data and computations into collections called types.

- Some examples of types are:
  - the type Int of integers
  - the type (Int→Int) of functions from integers to integers

Why do we need them?

- Consider "untyped" universes:
  - Bit string in computer memory
  - λ-expressions in λ calculus
  - Sets in set theory

- "untyped" = there's only 1 type

- Types arise naturally to categorize objects according to patterns of use
  - E.g. all integer numbers have same set of applicable operations

Use of Types

- Identifying and preventing meaningless errors in the program
  - Compile-time checking
  - Run-time checking

- Program Organization and documentation
  - Separate types for separate concepts
  - Indicates intended use declared identifiers

- Supports Optimization
  - Short integers require fewer bits
  - Access record component by known offset

Type Errors

- A type error occurs when a computational entity, such as a function or a data value, is used in a manner that is inconsistent with the concept it represents
- Languages represent values as sequences of bits. A "type error" occurs when a bit sequence written for one type is used as a bit sequence for another type
- A simple example can be assigning a string to an integer or using addition to add an integer or a string

Type Systems

- A tractable syntactic framework for classifying phrases according to the kinds of values they compute
- By examining the flow of these values, a type system attempts to prove that no *type errors* can occur
- Seeks to guarantee that operations expecting a certain kind of value are not used with values for which that operation does not make sense

Type Safety

A programming language is type safe if no program is allowed to violate its type distinctions

Example of current languages:
Not Safe : C and C++
Type casts, pointer arithmetic

Almost Safe : Pascal
Explicit deallocation; dangling pointers

Safe : Lisp, Smalltalk, ML, Haskell, Java, Scala
Complete type checking

Type Checking - Compile Time

- Check types at compile time, before a program is started
- In these languages, a program that violates a type constraint is not compiled and cannot be executed

Expressiveness of the Compiler:

a) *sound*
If no programs with errors are considered correct

b) *conservative*
if some programs without errors are still considered
to have errors (especially in the case of type-safe languages)

Type Checking - Run Time

- The compiler generates the code
- When an operation is performed, the code checks to make sure that the operands have the correct type

Combining the Compile and Run time

- Most programming languages use some combination of compile-time and run-time type checking
- In Java, for example, static type checking is used to distinguish arrays from integers, but array bounds errors are checked at run time.

A Comparison – Compile vs. Run Time

Form of Type Checking	**Advantages**	**Disadvantages**
Compile - Time	• Prevents type errors • Eliminates run-time tests • Finds type errors before execution and run-time tests	• May restrict programming because tests are conservative
Run - Time	• Prevents type errors • Need not be conservative	• Slows Program Execution

Type Declarations

Two basic kinds of type declaration:

1. *transparent*
    - meaning an alternative name is given to a type that can also be expressed without this name

For example, in C, the statements,

typedef char byte;
typedef byte ten bytes[10];

the first, declaring a type byte that is equal to char and the second an array type ten bytes that is equal to arrays of 10 bytes

Type Declarations

2. Opaque

Opaque, meaning a new type is introduced into the program that is not equal to any other type

Example in C,

```
typedef struct Node{
    int val;
    struct Node *left;
    struct Node* right;
}N;
```

Type Inference

- Process of identifying the type of the expressions based on the type of the symbols that appear in them

- Similar to the concept of compile type checking
  - All information is not specified
  - Some degree of logical inference required

- Some languages that include Type Inference are Visual Basic (starting with version 9.0), C# (starting with version 3.0), Clean, Haskell, ML, OCaml, Scala

- This feature is also being planned and introduced for C++0x and Perl6

Type Inference

Example: Compile Time checking:

For language, C:

```
int addone(int x) {
     int result;     /*declare integer result (C language)*/
     result = x+1;
     return result;
}
```

Lets look at the following example,

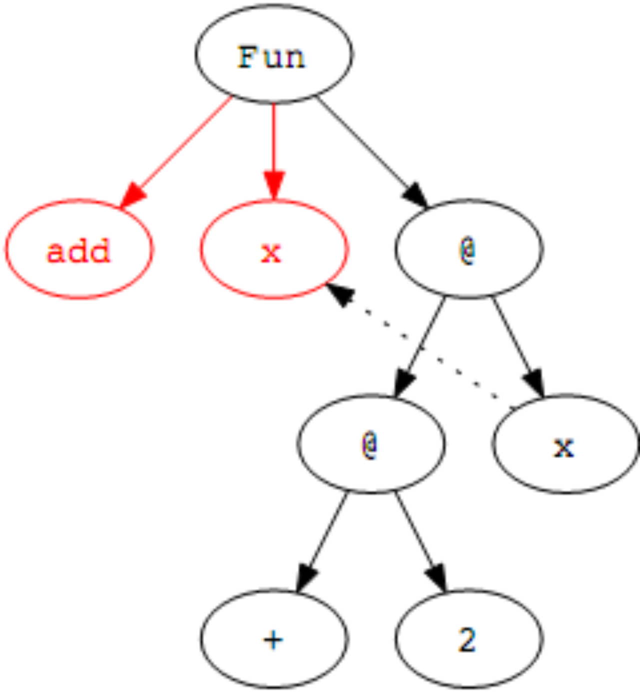
```
addone(x) {
    val result;      /*inferred-type result */
    result = x+1;
    return result;
}
```

Haskell Type Inference Algorithm

There are three steps:

1. Assign a type to the expression and each subexpression.
2. Generate a set of constraints on types, using the parse tree of the expression.
3. Solve these constraints by means of unification, which is a substitution-base algorithm for solving systems of equations.

Explanation of the Algorithm



Consider an example function:

add x = 2 + x
add :: Int → Int

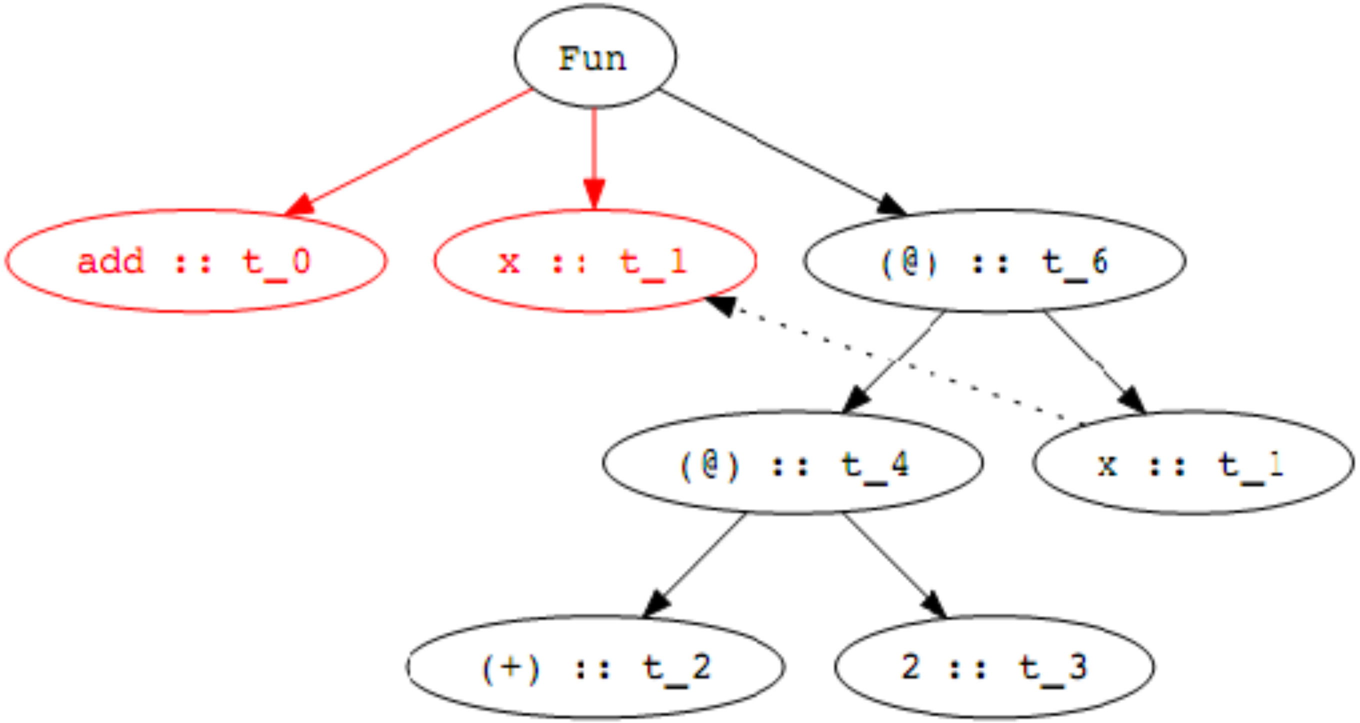
Step 0:

Construct a parse tree.

- Node 'Fun' represents function declaration.
- Children of Node 'Fun' are name of the function 'add', its argument and function body.
- The nodes labeled ‘@’ denote function applications, in which the left child is applied to the right child.
- Constant expressions like '+', '3' and variables also have their own node.

Explanation of the Algorithm

Step 1:
Assign a type variable to the expression and each subexpression.
Each of the types, written t_i for some integer i,is a type variable, representing the eventual type of the associated expression.



Explanation of the Algorithm

Step 2:
Generate a set of constraints on types, using the parse tree of the expression.

- Constant Expression: we add a constraint equating the type variable of the node with the known type of the constant

- Variable will not introduce any type constraints

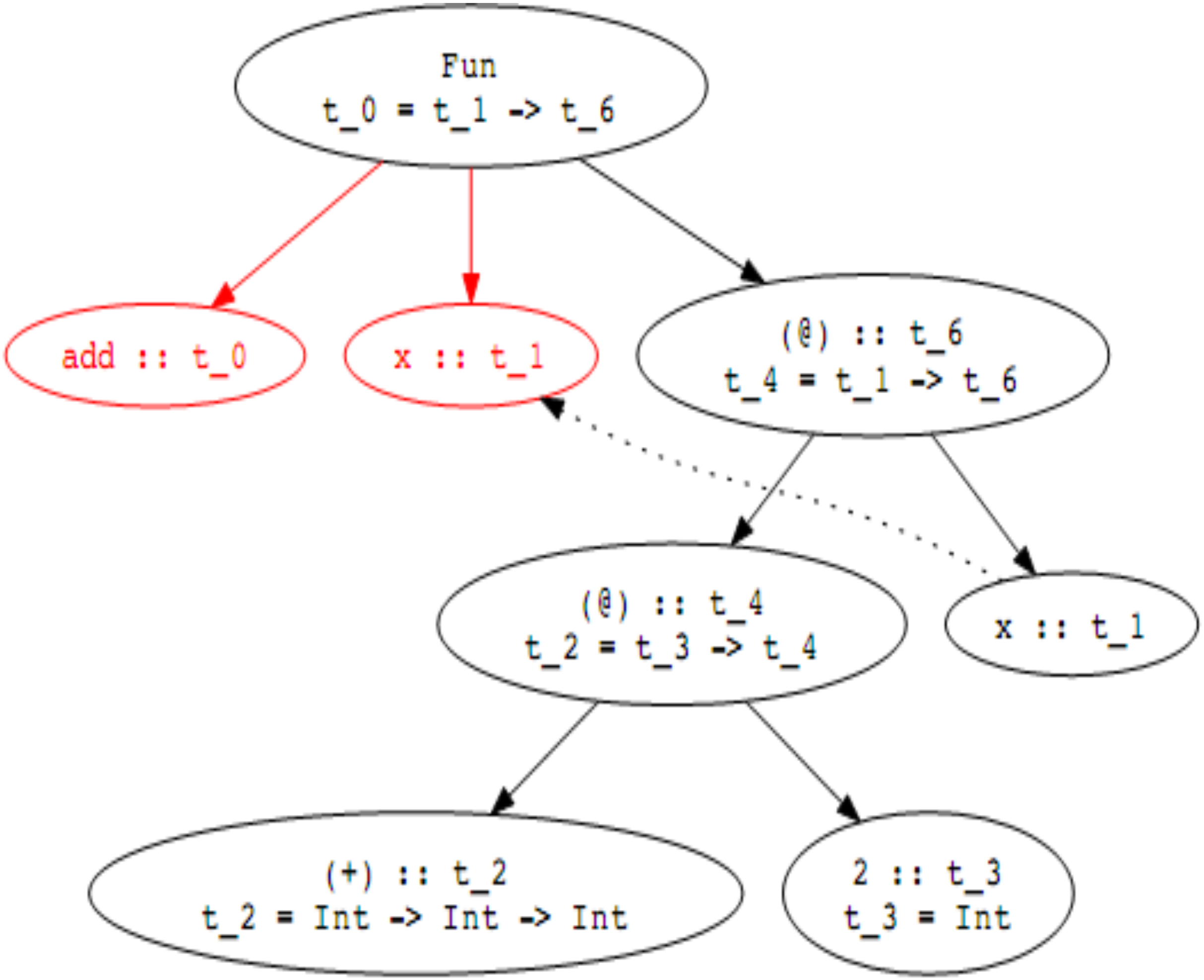
- Function Application (@ nodes): If the type of 'f' is t_f, the type of 'a' is t_a, and the type of 'f a' is t_r,then we must have t_f = t_a -> t_ r

- Function Definition: If 'f' is a function with argument 'x' and body 'b',then if 'f' has type 't_f', 'x' has type t_x, and 'b' has type 't_b', then these types must satisfy the constraint t_f=t_x -> t_b

Explanation of the Algorithm

Set of Constraints generated:

1. t_0 = t_1 -> t_6
2. t_4 = t_1 -> t_6
3. t_2 = t_3 -> t_4
4. t_2 = Int -> (Int -> Int)
5. t_3 = Int



Explanation of the Algorithm

Step 3:
Solve the generated constraints using unification
For Equations (3) and (4) to be true, it must be the case that
t_3 -> t_4 = Int -> (Int -> Int), which implies that
6. t_3 = Int
7. t_4 = Int -> Int

Equations (2) and (7) imply that
8. t_1 = Int
9. t_6 = Int

Result of the Algorithm

Thus the system of equations that satisfy the assignment of all the variables:

t_0 = Int -> Int
t_1 = Int
t_2 = Int -> Int -> Int
t_3 = Int
t _4 = Int -> Int
t _6 = Int

Polymorphism

- Constructs that can take different forms
- poly = many
  morph = shape

Types of Polymorphism

- ***Ad-hoc polymorphism***
  similar function implementations for different types
  (method overloading, but not only)

- ***Subtype (inclusion) polymorphism***
  instances of different classes related by common super class

```
class A {...}
class B extends A {...}; class C extends A {...}
```

- ***Parametric polymorphism***
  functions that work for different types of data

```
def plus(x, y):
    return x + y
```

Ad-hoc Polymorphism

```
int plus(int x, int y) {
     return x + y;
}

string plus(string x, string y)
{
     return x + y;
}

float plusfloat(float x, float y)
{
     return x + y;
}
```

Subtype Polymorphism

- First introduced in the 60s with Simula

- Usually associated with OOP
  (in some circles, polymorphism = subtyping)

- Principle of safe substitution (Liskov substitution principle)

> “*if S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of the program.*”

Note that this is **behavioral** subtyping, stronger than simple functional subtyping.

Behavioral Subtyping Requirements

- Contravariance of method arguments in subtype (from narrower to wider, e.g. Triangle to Shape)
- Covariance of return types in subtype (from wider to narrower, e.g. Shape to Triangle)
- Preconditions cannot be strengthened in subtype
- Postcondition cannot be weakened in subtype
- History constraint: state changes in subtype not possible in supertype are not allowed (Liskov's constraint)

LSP Violations?

```
class Thing {...}

class Shape extends Thing {
  Shape m1(Shape a) {...}
}

class Triangle extends Shape {
  @Override
  Triangle m1(Shape a) {...}
}

class Square extends Shape {
  @Override
  Thing m1(Shape a) {...}
}
```

Java does not support contravariance of method arguments

LSP Violations?

```
class Thing {...}

class Shape extends Thing {
  Shape m1(Shape a) {
    assert(Shape.color == Color.Red);
    ...
  }
}

class Triangle extends Shape {
  @Override
  Triangle m1(Shape a) {
    assert(Shape.color == Color.Red);
    assert(Shape.nsizes == 3);
    ...
  }
}
```

Parametric Polymorphism

- ***Parametric polymorphism***
  functions that work for different types of data

```
def plus(x, y):
     return x + y
```

How to do this in statically-typed languages?

```
int plus(int x, int y):
     return x + y

???
```

Parametric Polymorphism

- Parametric polymorphism for statically-typed languages introduced in ML in the 70s
- aka “generic functions”
- C++: templates
- Java: generics
- C#, Haskell: parametric types

Parametric Polymorphism

Explicit Parametric Polymorphism

C++ implements explicit polymorphism, in which explicit instantiation or type application to indicate how type variables are replaced with specific types in the use of a polymorphic value.

Example:

```
  template <typename T>
        T lessthan(T& x, T& y){
          if( x < y) return x;
          else
                return y;
       }
```

We define a template function with T as a parameter which can take any type as a parameter to the function.

Parametric Polymorphism

Explicit Parametric Polymorphism

Java example:

```
 /**
  * Generic version of the Box class.
  * @param <T> the type of value being boxed
  */

public class Box<T> {

    // T stands for "Type"
    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

```
Box<Integer> integerBox;
...
void m(Box<Foo> fbox) {...}
```