CS40032: Principles of Programming Languages
Module 06: Type Systems

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

Source: *Concepts in Programming Languages*
by John C. Mitchell, Cambridge University Press, 2003

Feb 08, 10 & 15: 2021

Table of Contents

Remaining slides commented here - are out of syllabus

Type Systems

What is a Type System?

- A **type system** is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute

What is a Type?

- In general, a **type** is a collection of computational entities that share some common property
  - *Type Examples*: *int*, *bool*, $int \rightarrow bool$, etc.
  - *Type Non-Examples*: 3, *true*, *Even Integeres*, etc.
  - Distinction between sets of values that are types and sets that are not types is language dependent
- There are three main uses of types in programming languages:
  - Naming and organizing concepts
  - Making sure that bit sequences in computer memory are interpreted consistently
  - Providing information to the compiler about data manipulated by the program

Advantages of Types

- Program organization and documentation
  - Separate types for separate concepts
    - Represent concepts from problem domain
  - Document intended use of declared identifiers
    - Types can be checked, unlike program comments
- Identify and prevent errors
  - Compile-time or run-time checking can prevent meaningless computations such as `3 + true - "Bill"`
- Support optimization
  - Short integers require fewer bits
  - Access components of structures by known offset

What is a Type Error?

- 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
- Whatever the compiler/interpreter says it is?
- Something to do with bad bit sequences?
  - Floating point representation has specific form
  - An integer may not be a valid float
  - *Hardware Error*
    - `int x; x();`
    - `float_add(3, 4.5)`
- Something about programmer intent and use?
  - A type error occurs when a value is used in a way that is inconsistent with its definition
  - *Unintended Semantics*
    - `int_add(3, 4.5)`
    - declare as character, use as integer

Type errors are language dependent

- Array out of bounds access
  - C/C++: runtime errors
  - Haskell/Java: dynamic type errors
- Null pointer dereference
  - C/C++: run-time errors
  - Haskell/ML: pointers are hidden inside datatypes
    - Null pointer dereferences would be incorrect use of these datatypes, therefore static type errors

Type Safety

- A programming language is **type safe** if no program is allowed to violate its type distinctions
- A safe language protects its own abstractions:

Safety	*Statically checked*	*Dynamically checked*	*Remarks*
Unsafe	BCPL (*Basic Combined Programming Language*[1]) family including C, C++		• Type casts • Unions • Pointer arithmetic
Almost Safe	Algol family, Pascal, Ada		• Dangling pointers • Explicit Deallocation • Hard to make languages with explicit deallocation of memory fully type-safe
Safe	ML, Haskell, Java	Lisp, SmallTalk, Javascript, Scheme, Perl, Postscript, Python	• Complete type checking

[1] Procedural, Imperative, and Structured programming language

What are Type System good for?

- **Detecting Errors**: Static type-checking allows early detection of some programming errors
- **Abstraction**: Enforces disciplined programming
- **Documentation**: Types are useful when reading programs
- **Language Safety**: A safe language protects its own abstractions; Portability
- **Efficiency**: Distinguish between integer-valued arithmetic expressions and real-valued ones; Eliminate many of the dynamic checks; etc.
- **Other Applications**
    - *Computer and network security*
    - *Program analysis tools*
    - *Automated theorem proving*
    - *Database* – type analysis of *Document Type Definitions* and other kinds of schemas (such as XML-Schema standard [XS 2000]) for describing structured data in XML
    - *Computational linguistics* – typed λ–calculi form the basis for formalisms such as *categorical grammar*

Compile-time vs Run-time Checking

- JavaScript and Lisp use run-time type checking
  - `f(x)`: Make sure `f` is a function before calling `f`
  - `$ var f = 3`
    `$ f(2);`
    `$ typein:3: TypeError: f is not a function`
  - In JavaScript, we can write a function like
    `function f(x) { return x < 10 ? x : x(); }`
    Some uses will produce type error, some will not
- Haskell and Java use compile-time type checking
  - `f(x)`: Must have `f::A` $\rightarrow$ `B` and `x::A` inside datatypes
- Basic tradeoff
  - Both kinds of checking prevent type errors
  - Run-time checking slows down execution
  - Compile-time checking restricts program flexibility
    - JavaScript array: elements can have different types
    - Haskell list: all elements must have same type
  - Which gives better programmer diagnostics?

Compile-time vs Run-time Checking: Conservativity of Compile-Time Checking

- A property of compile-time type checking is that the compiler must be conservative
- The Compile-time type checking will find all statements and expressions that produce run-time type errors, but also may flag statements or expressions as errors even if they do not produce run-time errors
- More specifically, most checkers are both *sound* and *conservative*
  - A type checker is *sound* if no programs with errors are considered correct
  - A type checker is *conservative* if some programs without errors are still considered to have errors
- For any Turing-complete programming language, the set of programs that may produce a run-time type error is undecidable

```
if (complicated-expression-that-could-run-forever)
    then (expression-with-type-error)
    else (expression-with-type-error)
```

- Static typing is always conservative

```
function f(x) { return x < 10 ? x : x(); }
if (complicated-boolean-expression)
    then f(5);
    else f(15);
```

Compile-time vs Run-time Checking: Comparative

PoPL-06

Partha Pratim Das

Type Systems
Type & Type Error
Type Safety
Type Checking
Type Inference

Type Inference
add x = 2 + x
apply (f, x)
Inference Algorithm
Unification

Examples
sum
length
append
Homework

Type Deduction
Polymorphism
Ad-hoc
Parametric
Subtype
C++11,...

- Trade-offs between compile-time and run-time checking

Form of Type Checking	Advantages	Disadvantages
Run-time	• Prevents type errors • Need not be conservative	• Slows program execution
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*

- Combining Compile-Time and Run-Time Checking
  - Most programming languages actually 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 (which are a form of type error) are checked at run-time

Type Inference

- **Type inference** is the process of determining the types of expressions based on the known types of some symbols that appear in them
- The difference between type inference and compile-time type checking is really a matter of degree
- A **type-checking** algorithm goes through the program to check that the types declared by the programmer agree with the language requirements
- In **type inference**, the idea is that some information is not specified, and some form of logical inference is required for determining the types of identifiers from the way they are used

Type Inference: Checking vs Inference

- Standard Type Checking

```
int f(int x) { return x+1; };
int g(int y) { return f(y+1)*2; };
```

  - Examine body of each function
  - Use declared types to check agreement

- Type Inference

```
T f(T x) { return x+1; };
T g(T y) { return f(y+1)*2; };
```

  - Examine code without type information
  - Infer the most general types that could have been declared

Type Inference

- In addition to providing a flexible form of compile-time type checking, type inference supports **polymorphism**
- Type-inference algorithm uses **type variables** as placeholders for types that are not known
- In some cases, the type-inference algorithm resolves all type variables and determines that they must be equal to specific types such as Int, Bool, or String
- In other cases, the type of a function may contain type variables that are not constrained by the way the function is defined. In these cases, the function may be applied to any arguments whose types match the form given by a type expression containing type variables

Type Inference

Source: *Lecture 26: Type Inference and Unification*
Cornell University, 2005

https://www.cs.cornell.edu/courses/cs3110/2011sp/Lectures/lec26-type-inference/type-inference.htm

Type Inference by Hand-weaving: Example 1

- Consider

```
f x = 2 + x
> f :: Int -> Int
```

- What is the type of f?
  - $+$ has type: Int $\rightarrow$ Int $\rightarrow$ Int
  - 2 has type: Int
  - Since we are applying $+$ to x we need x :: Int
  - Therefore f x $=$ 2 $+$ x has type Int $\rightarrow$ Int

Type Inference by Hand-weaving: Example 2

- Consider

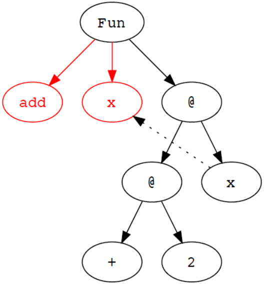
```
f (g, h) = g (h(0))
> f :: (a -> b, Int -> a) -> b
```

- What is the type of f?
  - h is applied to an integer argument, and so h must be a function from Int to something
  - Represent "something" by introducing a type variable a.
  - g must be a function that takes whatever h returns (of type a) and then returns something else
  - g is not constrained to return the same type of value as h, so we represent this second one by a new type variable, b
  - Putting the types of h and g together, we see that the first argument to f has type (a $\rightarrow$ b) and the second has type (Int $\rightarrow$ a)
  - Function f takes the pair of these two functions as an argument and returns the same type of value as g returns
  - Therefore, the type of f is (a $\rightarrow$ b, Int $\rightarrow$ a) $\rightarrow$ b.

Type Inference Algorithm: Example 1: Simple Function: `add x = 2 + x`

- `add x = 2 + x`

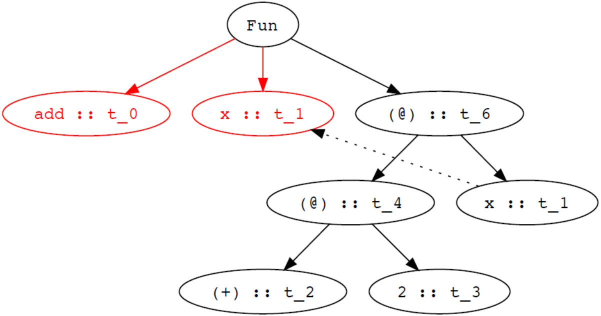
  `add :: ?`
- **Parse Program** text to construct parse tree
- Infix operators are converted to Curried function application during parsing: $2 + x \Rightarrow (+)\ 2\ x$
- Dotted link shows where the variable is bound



Parse Tree for Add Function

Type Inference Algorithm: Example 1: Simple Function: `add x = 2 + x`

- `add x = 2 + x`
- **Assign type variables to nodes**
- Variables are given same type as binding occurrence

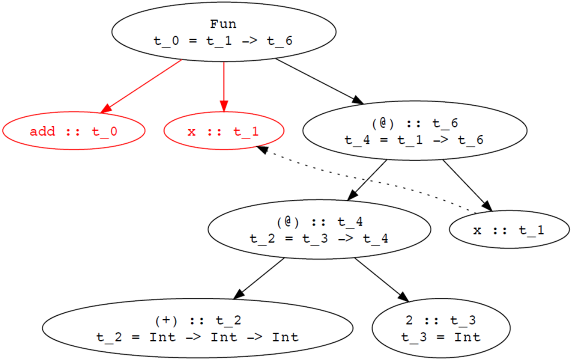


Parse Tree Labeled with Type Variables

Type Inference Algorithm: Example 1: Simple Function: add x = 2 + x

- **Add Constraints**

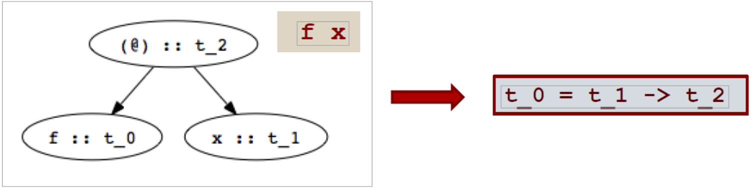
```
t_0 = t_1 -> t_6
t_4 = t_1 -> t_6
t_2 = t_3 -> t_4
t_2 = Int -> Int -> Int
t_3 = Int
```



Parse Tree Labeled with Type Constraints

Type Inference Algorithm: Constraints from Application Nodes

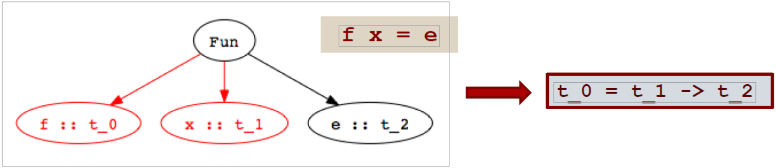
- Function application (apply f to x)
    - Type of f (t_0 in figure) must be domain $\rightarrow$ range
    - Domain of f must be type of argument x (t_1 in fig)
    - Range of f must be result of application (t_2 in fig)
    - Constraint: `t_0 = t_1 -> t_2`



- We shall see this formally in Slides 46

Type Inference Algorithm: Constraints from Abstractions

- Function declaration / abstraction
  - Type of f (t_0 in figure) must be domain → range
  - Domain is type of abstracted variable x (t_1 in fig)
  - Range is type of function body e (t_2 in fig)
  - Constraint: `t_0 = t_1 -> t_2`



- We shall see this formally in Slides 46

Type Inference Algorithm: Example 1: Simple Function: add x = 2 + x

- **Solve Constraints**

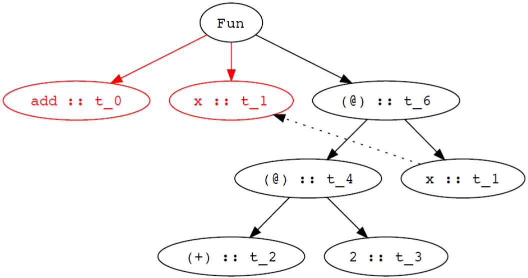
```
t_0 = t_1 -> t_6
t_4 = t_1 -> t_6
t_2 = t_3 -> t_4           ==>    t_3 -> t_4 = Int -> (Int -> Int)
t_2 = Int -> Int -> Int           t_3 = Int
t_3 = Int                         t_4 = Int -> Int

t_0 = t_1 -> t_6
t_4 = t_1 -> t_6           ==>    t_1 -> t_6 = Int -> Int
t_4 = Int -> Int                  t_1 = Int
t_2 = Int -> Int -> Int           t_6 = Int
t_3 = Int

t_0 = Int -> Int
t_1 = Int
t_6 = Int
t_4 = Int -> Int
t_2 = Int -> Int -> Int
t_3 = Int
```

Type Inference Algorithm: Example 1: Simple Function: add x = 2 + x

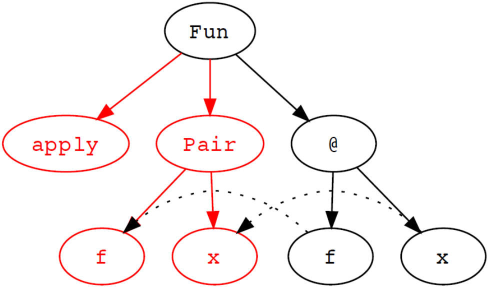
- **Determine type of declaration**
- ```
  t_0 = Int -> Int
  t_1 = Int
  t_6 = Int
  t_4 = Int -> Int
  t_2 = Int -> Int -> Int
  t_3 = Int
  ```
- ```
  add x = 2 + x
  > add :: Int -> Int
  ```



Parse Tree Labeled with Type Variables

Type Inference Algorithm: Example 2: Polymorphic Function: `apply (f, x) = f x`

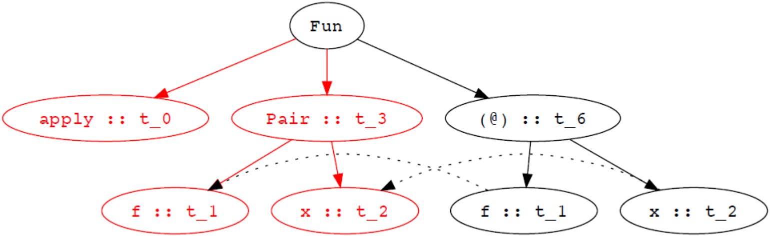
- `apply (f, x) = f x`
  `apply :: (t -> t1, t) -> t1`
- The `apply` function has a type involving type variables, making the function polymorphic.
- **Parse Program** text to construct parse tree



Parse Tree for Apply Function

Type Inference Algorithm: Example 2: Polymorphic Function: `apply (f, x) = f x`

- `apply (f, x) = f x`
- **Assign type variables to nodes**

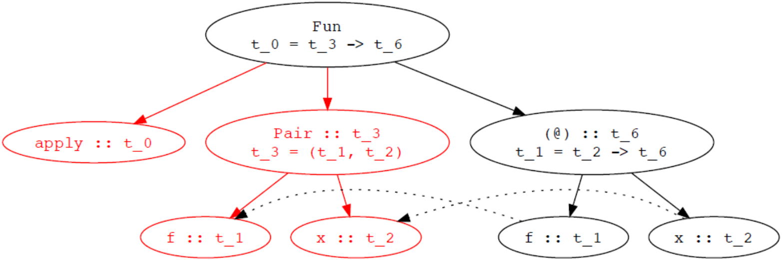


Parse Tree for Apply Function Labeled with Type Constraints

Type Inference Algorithm: Example 2: Polymorphic Function: `apply (f, x) = f x`

- **Add Constraints**

```
t_1 = t_2 -> t_6
t_0 = t_3 -> t_6
t_3 = (t_1, t_2)
```



Type Constraints for Apply Function

Type Inference Algorithm: Example 2: Polymorphic Function: `apply (f, x) = f x`

- **Solve Constraints**

```
t_0 = t_3 -> t_6
t_1 = t_2 -> t_6
t_3 = (t_1, t_2)

Replace t_3:
t_0 = (t_1, t_2) -> t_6
t_1 = t_2 -> t_6
t_3 = (t_1, t_2)

Replace t_1:
t_0 = (t_2 -> t_6, t_2) -> t_6
t_1 = t_2 -> t_6
t_3 = (t_2 -> t_6, t_2)

Replace t_2 w/ t and t_6 w/ t1:
t_0 = (t -> t1, t) -> t1
t_1 = t -> t1
t_3 = (t -> t1, t)
```

- **Determine Type**

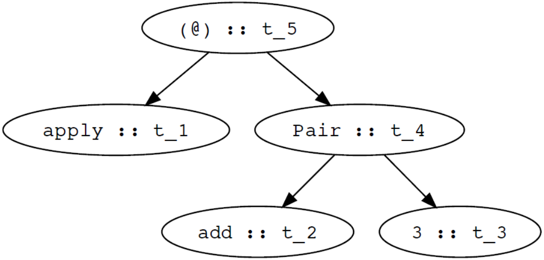
```
apply (f, x) = f x
> apply :: (t -> t1, t) -> t1
```

Type Inference Algorithm: Example 3: Function Application: apply (add, 3)


- ```
  apply (f, x) = f x
  > apply :: (a_1 -> a_2, a_1) -> a_2

  add x = 2 + x
  > add :: Int -> Int

  apply (add, 3) :: ?
  ```
- **Parse Tree and Assignment of Type Variables**

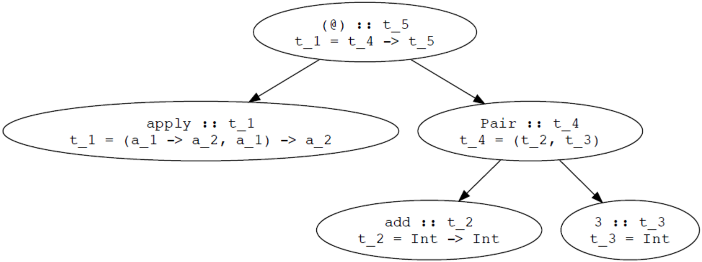


Type Variable Assignment for Apply Function Application Parse Tree
  ```

Type Inference Algorithm: Example 3: Function Application: apply (add, 3)

- **Add Constraints**

```
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int
t_3 = Int
t_4 = (t_2, t_3)
```



Constraints for Apply Function Application Parse Tree

Type Inference Algorithm: Example 3: Function Application: apply (add, 3)

- **Solve Constraints**

```
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int
t_3 = Int
t_4 = (t_2, t_3)
```

```
Equate t_1:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int
t_3 = Int
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
```

```
Equate t_4:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int
t_2 = a_1 -> a_2
t_3 = Int
t_3 = a_1
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
```

```
Equate t_2:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int
t_2 = a_1 -> a_2
t_3 = Int
t_3 = a_1
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
a_1 = Int
a_2 = Int
```

```
Solution:
t_1 = (Int -> Int, Int) -> Int
t_2 = Int -> Int
t_3 = Int
t_4 = (Int -> Int, Int)
t_5 = Int
a_1 = Int
a_2 = Int
```

Type Inference Algorithm: Example 3: Function Application: apply (add, 3)

- **Determine Type**
-
```
apply (f, x) = f x
> apply :: (a_1 -> a_2, a_1) -> a_2

add x = 2 + x
> add :: Int -> Int

apply (add, 3) :: t_5 = Int
```

Type Inference Algorithm: Example 4: Function Application: `apply (not, False)`


- ```
  apply (f, x) = f x
  > apply :: (a_1 -> a_2, a_1) -> a_2

  > not :: Bool -> Bool

  apply (not, False) :: ?
  ```
- Proceeding similarly as Example 4 gives:
  ```
  Solution:
  t_1 = (Bool -> Bool, Bool) -> Bool
  t_2 = Bool -> Bool
  t_3 = Bool
  t_4 = (Bool -> Bool, Bool)
  t_5 = Bool
  a_1 = Bool
  a_2 = Bool
  ```
- This fact illustrates the polymorphism of `apply`: Because the type `(a_1 -> a_2, a_1) -> a_2` of `apply` contains type variables, the function may be applied to any type of arguments that can be obtained if the type variables in `(a_1 -> a_2, a_1) -> a_2` are replaced with type names or type expressions.
  ```

Type Inference Algorithm: Example 4A: Function Application: `apply (add, False)`


- ```
  apply (f, x) = f x
  > apply :: (a_1 -> a_2, a_1) -> a_2

  add x = 2 + x
  > add :: Int -> Int

  apply (add, False) :: ?
  ```
  ```

Type Inference Algorithm: Example 4A: Function Application: apply (add, False)

- **Solve Constraints**

```
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int
t_3 = Bool
t_4 = (t_2, t_3)
```

```
Equate t_1:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int
t_3 = Bool
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
```

```
Equate t_4:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int
t_2 = a_1 -> a_2
t_3 = Bool
t_3 = a_1
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
```

```
Equate t_2:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Int -> Int
t_2 = a_1 -> a_2
t_3 = Bool
t_3 = a_1
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
a_1 = Int
a_2 = Int
```

```
Type Inconsistency:
t_1 = (Int -> Int, Int) -> Int
t_2 = Int -> Int
t_3 = Bool
t_3 = Int -- t_3 = a_1 = Int // Type Error
t_4 = (Int -> Int, ?)
t_5 = Int
a_1 = Int
a_2 = Int
```

Type Inference Algorithm: Example 4B: Function Application: `apply (not, 3)`


- ```
  apply (f, x) = f x
  > apply :: (a_1 -> a_2, a_1) -> a_2

  > not :: Bool -> Bool

  apply (not, 3) :: ?
  ```
  ```

Type Inference Algorithm: Example 4B: Function Application: `apply (not, 3)`

- **Solve Constraints**

```
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Bool -> Bool
t_3 = Int
t_4 = (t_2, t_3)
```

```
Equate t_1:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Bool -> Bool
t_3 = Int
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
```

```
Equate t_4:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Bool -> Bool
t_2 = a_1 -> a_2
t_3 = Int
t_3 = a_1
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
```

```
Equate t_2:
t_1 = (a_1 -> a_2, a_1) -> a_2
t_1 = t_4 -> t_5
t_2 = Bool -> Bool
t_2 = a_1 -> a_2
t_3 = Int
t_3 = a_1
t_4 = (t_2, t_3)
t_4 = (a_1 -> a_2, a_1)
t_5 = a_2
a_1 = Bool
a_2 = Bool
```

```
Type Inconsistency:
t_1 = (Int -> Int, Int) -> Int
t_2 = Bool -> Bool
t_3 = Int
t_3 = Bool -- t_3 = a_1 = Bool // Type Error
t_4 = (Bool -> Bool, ?)
t_5 = Bool
a_1 = Bool
a_2 = Bool
```

Type Inference Algorithm

1. Assign a type to the expression and each subexpression. For any compound expression or variable, use a type variable. For known operations or constants, such as $+$ or 3, use the type that is known for this symbol.
2. Generate a set of constraints on types, using the parse tree of the expression. These constraints reflect the fact that if a function is applied to an argument, for example, then the type of the argument must equal the type of the domain of the function.
3. Solve these constraints by means of unification, which is a substitution-based algorithm for solving systems of equations.

Framing & Solving Type Constraints: Matrix Example

- Let us write the type of an $m \times n$ matrix as $m \rightarrow n$. Now the rules of matrix algebra can be expressed as typing rules:
  - Multiplication
  $$\frac{\mathcal{E} \vdash A : s \rightarrow t, \mathcal{E} \vdash B : t \rightarrow u}{\mathcal{E} \vdash AB : s \rightarrow u}$$
  - Addition
  $$\frac{\mathcal{E} \vdash A : s \rightarrow t, \mathcal{E} \vdash B : s \rightarrow t}{\mathcal{E} \vdash A + B : s \rightarrow t}$$
  - Squaring
  $$\frac{\mathcal{E} \vdash A : s \rightarrow s}{\mathcal{E} \vdash A^2 : s \rightarrow s}$$
- What is the type of $(AB + CD)^2$, if the types of A, B, C, D are:

$$\begin{array}{lcl} A & : & s \rightarrow t \\ B & : & u \rightarrow v \\ C & : & w \rightarrow x \\ D & : & y \rightarrow z \end{array}$$

Framing & Solving Type Constraints: Matrix Example

- What is the type of $(AB + CD)^2$, if the types of A, B, C, D are: $A : s \to t, B : u \to v, C : w \to x, D : y \to z$
- We assign type variables for sub-expressions of $(AB + CD)^2$:

$$\begin{array}{lcl@{\qquad}rcl} \mathrm{A} & : & \mathrm{s} \to \mathrm{t} & \mathrm{AB} & : & \mathrm{a} \to \mathrm{b} \\ \mathrm{B} & : & \mathrm{u} \to \mathrm{v} & \mathrm{CD} & : & \mathrm{c} \to \mathrm{d} \\ \mathrm{C} & : & \mathrm{w} \to \mathrm{x} & \mathrm{AB} + \mathrm{CD} & : & \mathrm{e} \to \mathrm{f} \\ \mathrm{D} & : & \mathrm{y} \to \mathrm{z} & (\mathrm{AB} + \mathrm{CD})^2 & : & \mathrm{g} \to \mathrm{h} \end{array}$$

- Applying the typing rules we get:

$$\begin{array}{lll} \mathrm{t} = \mathrm{u},\ \mathrm{a} = \mathrm{s},\ \mathrm{b} = \mathrm{v} & \text{for} & AB \\ \mathrm{x} = \mathrm{y},\ \mathrm{c} = \mathrm{w},\ \mathrm{d} = \mathrm{z} & \text{for} & CD \\ \mathrm{a} = \mathrm{c} = \mathrm{e},\ \mathrm{b} = \mathrm{d} = \mathrm{f} & \text{for} & AB + CD \\ \mathrm{e} = \mathrm{f} = \mathrm{g} = \mathrm{h} & \text{for} & (AB + CD)^2 \end{array}$$

- Solving the constraints, we get three equivalence classes:

$$\mathrm{a} = \mathrm{b} = \mathrm{c} = \mathrm{d} = \mathrm{e} = \mathrm{f} = \mathrm{g} = \mathrm{h} = \mathrm{s} = \mathrm{v} = \mathrm{w} = \mathrm{z}$$
$$\mathrm{t} = \mathrm{u}$$
$$\mathrm{x} = \mathrm{y}$$

- Hence, $\begin{array}{lcl} \mathrm{A} & : & \mathrm{a} \to \mathrm{t} \\ \mathrm{B} & : & \mathrm{t} \to \mathrm{a} \\ \mathrm{C} & : & \mathrm{a} \to \mathrm{x} \\ \mathrm{D} & : & \mathrm{x} \to \mathrm{a} \end{array}$ is the most general typing. Any values for a, t, and x make the expression $(AB + CD)^2$ well-formed

Unification

- The task of *unification* is to find a *substitution* S that **unifies** two given terms (that is, makes them equal). Let us write $s\ S$ for the result of applying the substitution S to the term s.
- Given s and t, we want to find S such that $s\ S = t\ S$. Such a substitution S is called a **unifier** for s and t.
- Example, given the two terms

```
f x (g y)              f (g z) w
```

where x, y, z, and w are variables, the substitution

```
S  =  [x <- g z, w <- g y]
```

would be a unifier, since

```
f x (g y) [x <- g z, w <- g y] =
f (g z) (g y) =
f (g z) w [x <- g z, w <- g y]
```

- *Unification is a purely syntactic definition; the meaning of expressions is not considered when computing unifiers*

Unification

- **Unifiers do not necessarily exist**. For example, the terms x and $f\ x$ cannot be unified, since no substitution for x can make the two terms equal.
- **Even when unifiers exist, they are not necessarily unique**. For example, for the two terms

```
f x (g y)          f (g z) w
```

the substitution

```
T  =  [x <- g (f a b),
       y <- f b a,
       z <- f a b,
       w <- g (f b a)]
```

is also a unifier:

```
f x (g y) T  =
f (g (f a b)) (g (f b a)) =
f (g z) w) T
```

Unification: mgu

- When a unifier exists, there is a **most general unifier (mgu)** that is unique up to renaming. A unifier S for s and t is an mgu for s and t if
  - S is a unifier for s and t; and
  - any other unifier T for s and t is a refinement of S; that is, T can be obtained from S by doing further substitutions.

  For example, the substitution

  ```
  S  =  [x <- g z, w <- g y]
  ```

  in the example above is an mgu for `f x (g y)` and `f (g z) w`. The unifier

  ```
  T  =  [x <- g (f a b), y <- f b a, z <- f a b, w <- g (f b a)]
  ```

  is a refinement of S, since $T = S\ U$, where

  ```
  U  =  [z <- f a b, y <- f b a]
  ```

  Note that

  ```
  f x (g y) S U
  = f x (g y) [x <- g z, w <- g y] [z <- f a b, y <- f b a]
  = f (g z) (g y) [z <- f a b, y <- f b a]
  = f (g (f a b)) (g (f b a))
  = f x (g y) T
  ```

Type Inference

- We assume that all bound variables are distinct. If not, we can rename bound variables (by α-reduction) to make this true. As in λ- calculus, it is always fine to α-convert. For example, `f x = x + 3` and `f y = y + 3` are semantically equivalent.
- The typing rules are:
  - **Function Application** (See Slide 23): An expression $(e_1\ e_2)$ only makes sense if e_1 is a function having a type of the form $s \to t$, and the input type of e_1 is the same as the type of its argument e_2. When these premises are satisfied, then the result, represented by the expression $(e_1\ e_2)$, has the same type as the result type of e_1.

$$\frac{\mathcal{E} \vdash e_1 : s \to t, \mathcal{E} \vdash e_2 : s}{\mathcal{E} \vdash (e_1\ e_2) : t}$$

  - **Function Abstraction** (See Slide 24): An expression $f\ x = e$ represents a function taking elements of the same type as x to elements of the type of e.

$$\frac{\mathcal{E} \vdash x : s, \mathcal{E} \vdash e : t}{\mathcal{E} \vdash f\ x = e : s \to t}$$

  - Essentially, it is necessary to maintain a **type environment** $\mathcal{E}$, and type inferences are done with respect to that environment.

Type Inference

- These rules impose constraints as follows. Suppose we want to do type inference on a given expression e. We first assign unique type variables $'a$:
  - one to each variable x occurring in e, and
  - one to each occurrence of each subexpression of e.
- In the 1^{st} clause, the type variable is associated with the variable (x), and in the 2^{nd}, it is associated with the occurrence of the subexpression in e. Call the type variable assigned to x in the 1^{st} clause $u(x)$, and call the type variable assigned to occurrence of a subexpression e in the 2^{nd} clause $v(e)$
- Now we take the following constraints:
  - $u(x) = v(x)$ for each occurrence of a variable x
  - $v(e_1) = v(e_2) \rightarrow v((e_1\ e_2))$ for each occurrence of a subexpression $(e_1\ e_2)$
  - $v(f\ x = e) = v(x) \rightarrow v(e)$ for each occurrence of a subexpression $f\ x = e$

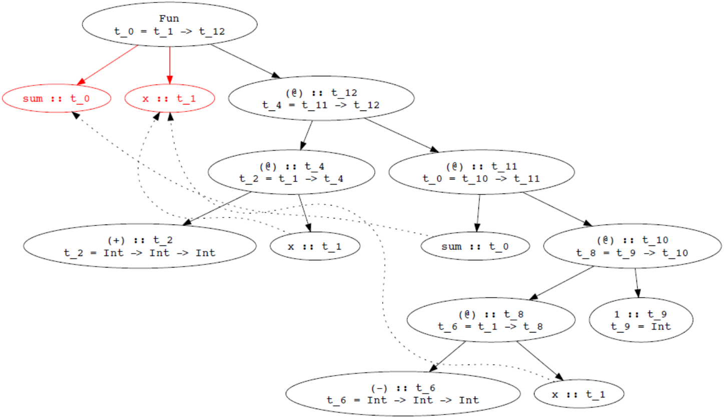
Type Inference Examples

Source: *Lecture 26: Type Inference and Unification*
Cornell University, 2005

`https://www.cs.cornell.edu/courses/cs3110/2011sp/Lectures/lec26-type-inference/type-inference.htm`

Type Inference Algorithm: Example 5: Recursive Function: sum

- `sum x = x + sum (x-1)`
  `> sum :: Int -> Int`



Parse Tree for sum Function Annotated with Type Variables and Associated Constraints

Type Inference Algorithm: Example 5: Recursive Function: sum

- **Add Constraints**

```
t_0  = t_1 -> t_12
t_0  = t_10 -> t_11
t_2  = t_1 -> t_4
t_2  = Int -> Int -> Int
t_4  = t_11 -> t_12
t_6  = t_1 -> t_8
t_6  = Int -> Int -> Int
t_8  = t_9 -> t_10
t_9  = Int
```

- **Solve Constraints**

```
t_0  = Int -> Int
t_1  = t_10 = Int
t_2  = Int -> (Int -> Int)
t_4  = Int -> Int
t_6  = Int -> (Int -> Int)
t_8  = Int -> Int
t_9  = Int
t_10 = t_1 = Int
t_11 = t_12 = Int
t_12 = t_11 = Int
```

Type Inference Algorithm: Example 5: Recursive Function: `sum`

- `sum x = x + sum (x-1)`
  `> sum :: Int -> Int`
- As the constraints can be solved, the function is *typeable*. By the solution of the constraints, the type of `sum` (the type `t_0`) is `Int -> Int`

PoPL-06

Partha Pratim Das

Type Inference Algorithm: Example 6: Polymorphic Datatypes: `length`

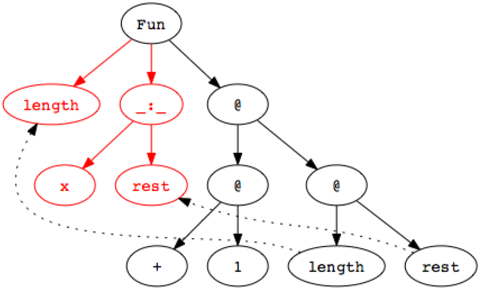
- Functions may have multiple clauses

```
length [] = 0
length (x:rest) = 1 + (length rest)
> length :: [t] -> Int
```

- Type inference
  - Infer separate type for each clause
  - Combine by adding constraint that all clauses must have the same type
  - Recursive calls: function has same type as its definition

Type Inference Algorithm: Example 6: Polymorphic Datatypes: `length`

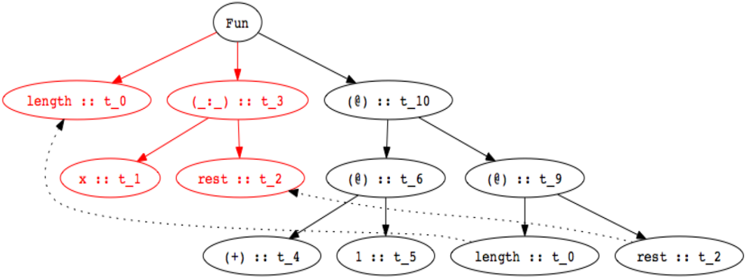
- `length (x:rest) = 1 + (length rest)`
- The `length` function has a type involving type variables, making the function polymorphic.
- **Parse Program** text to construct parse tree



Parse Tree for `length` Function

Type Inference Algorithm: Example 6: Polymorphic Datatypes: `length`

- `length (x:rest) = 1 + (length rest)`
- **Assign type variables to nodes**



Parse Tree Labeled with Type Variables

Type Inference Algorithm: Example 6: Polymorphic Datatypes: `length`

- `length (x:rest) = 1 + (length rest)`
- **Add Constraints**

```
t_0 = t_3 -> t_10
t_3 = t_2
t_3 = [t_1]
t_6 = t_9 -> t_10
t_4 = t_5 -> t_6
t_4 = Int -> Int -> Int
t_5 = Int
t_0 = t_2 -> t_9
```

- **Solve Constraints**

```
t_6 = Int -> Int
t_4 = Int -> Int -> Int
t_5 = Int
t_9 = Int
t_10 = Int
t_0 = [t_1] -> Int
```

- Conforms for:

```
length [] = 0
> length :: [t_1] -> Int
```

Type Inference Algorithm: Example 6: Multi-Clause Function: append

- Type inference for functions with several clauses may be done by a type check of each clause separately. Then, because all clauses define the same function, we impose the constraint that the types of all clauses must be equal.
- ```
  append ([], r) = r
  append (x:xs, r) = x : append(xs, r)
  > append :: ([t], [t]) -> [t]
  ```
- As the type `([t], [t]) -> [t]` indicates, append can be applied to any pair of lists, as long as both lists contain the same type of list elements. Thus, append is a polymorphic function on lists.
  ```

Type Inference Algorithm: Example 6: Multi-Clause Function: append


- ```
  append ([], r) = r
  append (x:xs, r) = x : append(xs, r)
  > append :: ([t], [t]) -> [t]
  ```
- Intuitively, the first clause has type

  `([t], t_1) -> t_1`

  because the first argument must match the empty list `[]`, but the second argument may be anything
- The second clause has type

  `([t], t_1) -> [t]`

  because the return result is a list containing one element from the list passed as the first argument.
- If we require that the two clauses have the same type by imposing the constraint

  `([t], t_1) -> t_1 = ([t], t_1) -> [t]`

  we must have

  `t_1 = [t]`
- This equality gives us the final type for append:

  `append :: ([t], [t]) -> [t]`
  ```

Homework 1: reverse

Infer the types of the following:

1.
```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse xs ++ [x]
```

   where x is the head of the list, xs is the tail of the list, [x] is the list builder, ++ is the concatenation operator.

2.
```
appendreverse xs =
    let rev ( [], elem ) = elem
        rev ( y:ys, elem ) = rev( ys, y:elem)
    in rev( xs, [] )
```

3.
```
reverse2 xs  = app ([], xs)
       where
       app (ys, [])      = ys
       app (ys, (x:xs)) = app ((x:ys), xs)
```

4.
```
reverseW [] = []
reverseW (x:xs) = reverseW xs
reverseW :: [t] -> [t]
```

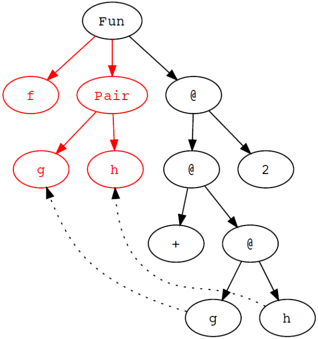
   Comment on the type correctness and logic correctness of the function `reverseW`

Homework 2: apply: modified

Use the parse graph in the figure below to calculate the μHaskell type for the function

`f(g, h) = g(h) + 2`

Assume that 2 has type `Integer` and + has type `Integer -> Integer -> Integer`

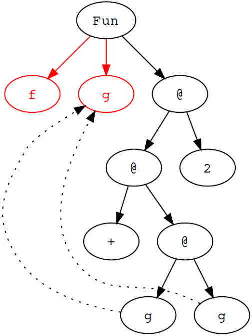


Homework 3: `apply`: self-apply

Use the following parse graph to follow the steps of the Haskell type-inference algorithm on the function declaration

```
f g = (g g) + 2
```

Assume that 2 has type `Integer` and + has type `Integer -> Integer -> Integer`
What is the output of the type checker?



Type Deduction in C++

Polymorphism

- Polymorphism, which literally means *having multiple forms*, refers to constructs that can take on different types as needed
- There are three forms of polymorphism in contemporary programming languages:
  - *Ad hoc polymorphism*, another term for overloading, in which two or more implementations with different types are referred to by the same name
  - *Parametric polymorphism*, in which a function may be applied to any arguments whose types match a type expression involving type variables
  - *Subtype polymorphism*, in which the subtype relation between types allows an expression to have many possible types.
- **The main characteristic of parametric polymorphism is that the set of types associated with a function or other value is given by a type expression that contains type variables**. For example, a Haskell function that sorts lists might have the Haskell type

```
sort :: ((t, t) -> Bool, [t]) -> [t]
```

Ad-hoc Polymorphism: Overloading: Overload Resolution

- To resolve overloaded functions with one parameter
  1. Identify the set of *Candidate Functions*
  2. From the set of candidate functions identify the set of *Viable Functions*
  3. Select the *Best viable function* through (*Order is important*)
     - Exact Match
     - Promotion
     - Standard type conversion
     - User defined type conversion

Ad-hoc Polymorphism: Overloading: Resolution: Candidate Function

Step 1: Find candidate functions via name lookup. Unqualified calls will perform both regular unqualified lookup as well as argument-dependent lookup (if applicable).

Source: Steps of Overload Resolution

Source: Overloaded Method Resolution

Source: Function overload resolution

Ad-hoc Polymorphism: Overloading: Resolution: Viable Function

Step 2: Filter the set of candidate functions to a set of *viable functions*. A viable function for which there exists an implicit conversion sequence between the arguments the function is called with and the parameters the function takes.

```
void f(char);          // (1)
void f(int ) = delete; // (2)
void f();              // (3)
void f(int& );         // (4)

f(4); // 1,2 are viable (even though 2 is deleted!)
      // 3 is not viable because the argument lists don't match
      // 4 is not viable because we cannot bind a temporary to
      //     a non-const lvalue reference
```

Ad-hoc Polymorphism: Overloading: Resolution: Best Match Function

Step 3: Pick the best viable candidate. A viable function F1 is a better function than another viable function F2 if the implicit conversion sequence for each argument in F1 is not worse than the corresponding implicit conversion sequence in F2, and...:

Step 3.1: For some argument, the implicit conversion sequence for that argument in F1 is a better conversion sequence than for that argument in F2, or

```
void f(int );  // (1)
void f(char ); // (2)

f(4);  // call (1), better conversion sequence
```

Ad-hoc Polymorphism: Overloading: Resolution: Best Match Function

Step 3.2: In a user-defined conversion, the standard conversion sequence from the return of F1 to the destination type is a better conversion sequence than that of the return type of F2, or

```
struct A {
    operator int();
    operator double();
} a;

int i = a; // a.operator int() is better than a.operator double() and a
float f = a; // ambiguous
```

Step 3.3: In a direct reference binding, F1 has the same kind of reference by F2 is not, or

```
struct A {
    operator X&();   // #1
    operator X&&(); // #2
};
A a;
X& lx = a;  // calls #1
X&& rx = a; // calls #2
```

Ad-hoc Polymorphism: Overloading: Resolution: Best Match Function

Step 3.4: F1 is not a function template specialization, but F2 is, or

```
template <class T> void f(T ); // #1
void f(int );                  // #2

f(42); // calls #2, the non-template
```

Step 3.5: F1 and F2 are both function template specializations, but F1 is more specialized than F2.

```
template <class T> void f(T );  // #1
template <class T> void f(T* ); // #2

int* p;
f(p); // calls #2, more specialized
```

Ambiguity: If there's no single best viable candidate at the end, the call is ambiguous:

```
void f(double ) { }
void f(float ) { }

f(42); // error: ambiguous
```

Overload Resolution: Exact Match

- lvalue-to-rvalue conversion
  - Most common
- Array-to-pointer conversion

  Definitions: `int ar[10];`
  `void f(int *a);`

  Call: `f(ar)`
- Function-to-pointer conversion

  Definitions: `typedef int (*fp) (int);`
  `void f(int, fp);`
  `int g(int);`

  Call: `f(5, g)`
- Qualification conversion
  - Converting pointer (only) to `const` pointer

Overload Resolution: Promotion & Conversion

- Examples of Promotion
  - `char` to `int`; `float` to `double`
  - `enum` to `int` / `short` / `unsigned int` / ...
  - `bool` to `int`
- Examples of Standard Conversion
  - integral conversion
  - floating point conversion
  - floating point to integral conversion

    The above 3 may be dangerous!
  - pointer conversion
  - bool conversion

Example: Overload Resolution with one parameter

- In the context of a list of function prototypes:

```
int g(double);                    // F1
void f();                         // F2
void f(int);                      // F3
double h(void);                   // F4
int g(char, int);                 // F5
void f(double, double = 3.4);     // F6
void h(int, double);              // F7
void f(char, char *);             // F8
```

The call site to resolve is:

```
f(5.6);
```

- Resolution:
  - Candidate functions (by name): F2, F3, F6, F8
  - Viable functions (by # of parameters): F3, F6
  - Best viable function (by type `double` – Exact Match): F6

Example: Overload Resolution fails

- Consider the overloaded function signatures:

```
int fun(float a) {...}            // Function 1
int fun(float a, int b) {...}     // Function 2
int fun(float x, int y = 5) {...} // Function 3

int main() {
    float p = 4.5, t = 10.5;
    int s = 30;

    fun(p, s); // CALL - 1
    fun(t);    // CALL - 2
    return 0;
}
```

- `CALL - 1`: Matches `Function 2` & `Function 3`
- `CALL - 2`: Matches `Function 1` & `Function 3`
- Results in ambiguity

Parametric Polymorphism

- Parametric polymorphism may be **implicit** or **explicit**
- In **explicit parametric polymorphism**, the program text contains type variables that determine the way that a function or other value may be treated polymorphically.
  - In addition, explicit polymorphism often involves **explicit instantiation** or type application to indicate how type variables are replaced with specific types in the use of a polymorphic value.
  - C++ templates are a well-known example of explicit polymorphism.
- Haskell polymorphism is called **implicit parametric polymorphism** because programs that declare and use polymorphic functions do not need to contain types – the type-inference algorithm computes when a function is polymorphic and computes the instantiation of type variables as needed.

Parametric Polymorphism: C++ Function Templates

- To swap values of variables of *any* type, we can define function template that uses a type variable T:

```
template <typename T>
void Swap(T& x, T& y) { T tmp = x; x = y; y = tmp; }
```

- `let Swap =` $\lambda(x : T).\lambda(y : T).\ E$, where T is the type variable
- Templates allow us to treat Swap as a function with a type argument.
- In C++, function templates are instantiated automatically as needed, with the types of the function arguments used to determine which instantiation is needed. For example:

```
int i,j; ... Swap(i,j); // replace T with int
float a,b; ... Swap(a,b); // replace T with float
string s,t; ... Swap(s,t); // replace T with String
```

- Applying type inference for `int i,j; ... Swap(i,j);`:
  - By Abstraction

    ```
    Swap (x, y) = ...
    Swap :: (T&, T&) -> void
    ```

  - By Application

    ```
    int i,j; ... Swap(i,j);
    Swap :: (int , int) -> void
    ```

  - `T = int`

Parametric Polymorphism: C++ Function Templates


- ```
  template <typename T>
  void Swap(T& x, T& y) { T tmp = x; x = y; y = tmp; }

  class IntWrap {
      int data;
  }
  ```
- Examples:

`int i,j;`	`Swap(i,j);`	`T = int`	
`double c,d;`	`Swap(c,d);`	`T = double`	
`string s,t;`	`Swap(s,t);`	`T = string`	
`IntWrap a,b;`	`Swap(a,b);`	`T = IntWrap`	
`int i,j;`	`Swap<int>(i,j);`	`T = int`	
`double c,d;`	`Swap<double>(c,d);`	`T = double`	
`string s,t;`	`Swap<string>(s,t);`	`T = string`	
`IntWrap a,b;`	`Swap<IntWrap>(a,b);`	`T = IntWrap`	
`const int ci, di;`	`Swap(ci,di);`	`T = const int`	const cannot be assigned
`int i, double d;`	`Swap(i,d);`	`T = ?`	template parameter 'T' is ambiguous
`int i, j;`	`Swap<double>(i,j);`	`T = double`	cannot convert argument 1 from 'int' to 'double
  ```

Parametric Polymorphism: C++ Function Templates


- ```
  template <typename T>
  void Swap(T& x, T& y) { T tmp = x; x = y; y = tmp; }

  class Uncopyable {
  protected:
      Uncopyable& operator=(const Uncopyable& u);
  };

  class IntWrap : public Uncopyable {
          int data;
  ```
- Examples:

`IntWrap a,b;`	`Swap(a,b);`	`T = IntWrap`	
	`Swap<IntWrap>(a,b);`		Link Error: unresolved external symbol "protected: class Uncopyable & __thiscall Uncopyable::operator=(class Uncopyable const &)"
  ```

Parametric Polymorphism: C++ Function Templates

- ```
  template<typename T> T Add(T& a, T&b) { return a + b; }

  class IntWrap { int i;
  public: IntWrap(int i_) : i(i_) {}
      friend IntWrap operator+(IntWrap& a, IntWrap& b) { return IntWrap(a.i + b.i); }
  };
  ```
- `let Add =` $\lambda(a : T).\lambda(b : T).\ (a + b)$, where T is the type variable
- Examples:

`int i,j;`	`Add(i,j);`	`T = int`
`double c,d;`	`Add(c,d);`	`T = double`
`string s,t;`	`Add(s,t);`	`T = string`
`IntWrap a,b;`	`Add(a,b);`	`T = IntWrap`
`int i,j;`	`Add<int>(i,j);`	`T = int`
`double c,d;`	`Add<double>(c,d);`	`T = double`

Mixed Mode Addition

`int i, double d;`	`Add(i,d);`	`T = ?`	template parameter 'T' is ambiguous
`int i, double d;`	`Add(d,i);`	`T = ?`	template parameter 'T' is ambiguous
`int i, double d;`	`Add<int>(i,d);`	`T = int`	no instance of function template "Add" matches the arg. list arg. types are: (int, double)
`int i, double d;`	`Add<double>(i,d);`	`T = double`	no instance of function template "Add" matches the arg. list arg. types are: (double, int)
  ```

Parametric Polymorphism: C++ Function Templates

- ```
  template<typename T1, typename T2> T1 Add(T1& a, T2&b) { return a + b; }

  class IntWrap { int i;
  public: IntWrap(int i_) : i(i_) {}
      friend IntWrap operator+(IntWrap& a, IntWrap& b) { return IntWrap(a.i + b.i); }
  };
  ```
- `let Add` = $\lambda(a : T1).\lambda(b : T2).\ (a + b)$, where $T1$, and $T2$ are type variables
- Examples:

`int i,j;`	`Add(i,j);`	`T1 = T2 = int`	
`double c,d;`	`Add(c,d);`	`T1 = T2 = double`	
`string s,t;`	`Add(s,t);`	`T1 = T2 = string`	
`IntWrap a,b;`	`Add(a,b);`	`T1 = T2 = IntWrap`	
		Mixed Mode Addition	
`int i, double d;`	`Add(i,d);`	`T1 = int, T2 = double`	warning: 'return' : conversion from 'double' to 'int'
`int i, double d;`	`Add(d,i);`	`T1 = double, T2 = int`	
  ```

Parametric Polymorphism: C++ Function Templates

- ```
  template<typename T1, typename T2, typename R>
  R Add(T1& a, T2&b) { return a + b; }

  class IntWrap { int i;
  public: IntWrap(int i_) : i(i_) {}
      friend IntWrap operator+(IntWrap& a, IntWrap& b) { return IntWrap(a.i + b.i); }
  };
  ```
- `let Add =` $\lambda(a : T1).\lambda(b : T2).\lambda(r : R).\ (a + b)$, where $T1$, $T2$, and R are type variables
- Examples:

`int i,j;`	`Add(i,j);`	`T1 = T2 = int, R = ?`	could not deduce template argument for 'R'
`double c,d;`	`Add(c,d);`	`T1 = T2 = double, R = ?`	-do-
`string s,t;`	`Add(s,t);`	`T1 = T2 = string, R = ?`	-do-
`IntWrap a,b;`	`Add(a,b);`	`T1 = T2 = IntWrap, R = ?`	-do-
		Mixed Mode Addition	
`int i, double d;`	`Add(i,d);`	`T1 = int, T2 = double, R = ?`	-do-
`int i, double d;`	`Add(d,i);`	`T1 = double, T2 = int, R = ?`	-do-
  ```

Parametric Polymorphism: C++ Function Templates

- ```
  template<typename T1, typename T2, typename R>
  R Add(T1& a, T2&b) { return a + b; }

  class IntWrap { int i;
  public: IntWrap(int i_) : i(i_) {}
      friend IntWrap operator+(IntWrap& a, IntWrap& b) { return IntWrap(a.i + b.i); }
  };
  ```
- Examples:

`int i,j;`	`Add<int, int, int>(i,j);`	`T1 = T2 = int` `R = int`
`double c,d;`	`Add<double, double, double>(c,d);`	`T1 = T2 = double` `R = double`
`string s,t;`	`Add<string, string, string>(s,t);`	`T1 = T2 = string` `R = string`
`IntWrap a,b;`	`Add<IntWrap, IntWrap, IntWrap>(a,b);`	`T1 = T2 = IntWrap` `R = IntWrap`

Mixed Mode Addition

`int i, double d;`	`Add<int, double, double>(i,d);`	`T1 = int, T2 = double` `R = double`
`int i, double d;`	`Add<double, int, double>(d,i);`	`T1 = double, T2 = int` `R = double`
  ```

Parametric Polymorphism: C++ Function Templates

- Using Partial Template Specialization

```
template<typename T1, typename T2, typename R> // Fn-3
R Add(T1& a, T2& b) { return a + b; }

template<typename T1, typename T2> // Fn-2. Replace R by T1
T1 Add(T1& a, T2& b) { return Add<T1, T2, T1>(a, b); }

template<typename T> // Fn-1. Replace T1, T2, and R by T
T Add(T& a, T& b) { return Add<T, T, T>(a, b); }

template<> // Fn-0. Replace T1, T2, and R by int
int Add(int& a, int& b) { return Add<int, int, int>(a, b); }

class IntWrap { int i; public: IntWrap(int i_) : i(i_) {}
    friend IntWrap operator+(IntWrap& a, IntWrap& b) { return IntWrap(a.i + b.i); }
};
```

- Examples:

```
int i,j;            Add(i,j);                         Fn-0

double c,d;         Add(c,d);                         Fn-1
string s,t;         Add(s,t);                         Fn-1
IntWrap a,b;        Add(a,b);                         Fn-1

int i, double d;    Add(i,d);                         Fn-2
int i, double d;    Add(d,i);                         Fn-2

int i, double d;    Add<int, double, double>(i,d);    Fn-3
int i, double d;    Add<double, int, double>(d,i);    Fn-3
```

Important Features of C++: auto

C++03	C++11
`map<string,string> m;` `map<string,string>::iterator it = m.begin();`	`map<string,string> m;` `auto it = m.begin();`
`singleton& s = singleton::instance();`	`auto &s = singleton::instance();`
`T *p=new T();`	`auto p = new T();`

- auto is a mechanism to deduce the type from initializer:

```
int x1;       // potentially uninitialized
auto x2;      // error! initializer required
auto x3 = 0; // fine, x's value is well-defined
```

-
```
template<class T> void printall(const vector<T>& v) {
    for(auto p=v.begin(); p!=v.end(); ++p) cout << *p << "\n";
}
```
is better than
```
template<class T> void printall(const vector<T>& v) {
    for (typename vector<T>::const_iterator p=v.begin();
        p!=v.end(); ++p) cout << *p << "\n";
}
```

Important Features of C++: decltype

- decltype is an operator for querying the type of an expression.
- Similar to the sizeof operator, the operand of decltype is unevaluated.

C++03	C++11
`int i=4;`	`int i=4;`
`const int j=6;`	`const int j=6;`
`const int &k=i;`	`const int &k=i;`
`int a[5];`	`int a[5];`
`int* p;`	`int* p;`
`int var1;`	`decltype(i) var1;`
`int var2;`	`decltype(1) var2;`
`int var3;`	`decltype(2+3) var3;`
`int& var4=i;`	`decltype(i=1) var4=i;//no assignment`
`const int var5=1;`	`decltype(j) var5=1;`
`const int& var6=j;`	`decltype(k) var6=j;`
`int var7[5];`	`decltype(a) var7;`
`int& var8=i;`	`decltype(a[3]) var8=i;`
`int& var9=i;`	`decltype(*p) var9=i;`

Important Features of C++: decltype

- `decltype(E)` is the type ("declared type") of the name or expression `E` and can be used in declarations:

```
void f(const vector<int>& a, vector<float>& b) {
    typedef decltype(a[0]*b[0]) Tmp;
    for (int i=0; i<b.size(); ++i) {
        Tmp* p = new Tmp(a[i]*b[i]);
        // ...
    }
    // ...
}
```

- If you just need the type for a variable that you are about to initialize auto is often a simpler choice.
- You really need `decltype` if you need a type for something that is not a variable, such as a return type.

Important Features of C++: decltype

- Consider

```
template<class T, class U>
??? mul(T x, U y) { return x*y; }
```

- How to write the return type? It's "the type of x*y" – but how can we say that? First idea, use decltype:

```
template<class T, class U>
decltype(x*y) mul(T x, U y) // scope problem!
{ return x*y; }
```

- That won't work because x and y are not in scope. So:

```
template<class T, class U>
decltype(*(T*)(0)**(U*)(0)) mul(T x, U y)
    // ugly! & error prone
{ return x*y; }
```

- Put the return type where it belongs, after the arguments:

```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y)
{ return x*y; }
```

- We use the notation auto to mean *return type to be deduced or specified later* in **Suffix Return Type Syntax**

Important Features of C++: Move Semantics

- Suppose X is a class that holds a pointer or handle to some resource, say, m_pResource. By a resource, we mean anything that takes considerable effort to construct, clone, or destruct. A good example is std::vector, which holds a collection of objects that live in an array of allocated memory. Then, logically, the copy assignment operator for X looks like this:

```
X& X::operator=(X const & rhs) {
  // [...]
  // Make a clone of what rhs.m_pResource refers to.
  // Destruct the resource that m_pResource refers to.
  // Attach the clone to m_pResource.
  // [...]
}
```

  Similar reasoning applies to the copy constructor. Now suppose X is used as:

```
X foo();
X x;
// perhaps use x in various ways
x = foo();
```

  The last line above

  - clones the resource from the temporary returned by foo,
  - destructs the resource held by x and replaces it with the clone,
  - destructs the temporary and thereby releases its resource.

Important Features of C++: Move Semantics

- Rather obviously, it would be ok, and much more efficient, to swap resource pointers (handles) between x and the temporary, and then let the temporary's destructor destruct x's original resource. In other words, in the special case where the right hand side of the assignment is an rvalue, we want the copy assignment operator to act like this:

```
// [...]
// swap m_pResource and rhs.m_pResource
// [...]
```

- This is called **move semantics**. With C++11, this conditional behavior can be achieved via an overload:

```
X& X::operator=(X&& rhs)
{
  // [...]
  // swap this->m_pResource and rhs.m_pResource
  // [...]
}
```

- Move Semantics is realized by *rvalue Reference*.

Important Features of C++: rvalue Reference

- **C Semantics**: An *lvalue* is an expression e that may appear on the left or on the right hand side of an assignment, whereas an *rvalue* is an expression that can only appear on the right hand side of an assignment.
  For example,

```
int a = 42;
int b = 43;

// a and b are both l-values:
a = b; // ok
b = a; // ok
a = a * b; // ok

// a * b is an rvalue:
int c = a * b; // ok, rvalue on right hand side of assignment
a * b = 42; // error, rvalue on left hand side of assignment
```

Important Features of C++: rvalue Reference

- **C++ Semantics**: An *lvalue* is an expression that refers to a memory location and allows us to take the address of that memory location via the & operator. An *rvalue* is an expression that is not an *lvalue*.
  For example,

```
// lvalues:
//
int i = 42;
i = 43; // ok, i is an lvalue
int* p = &i; // ok, i is an lvalue
int& foo();
foo() = 42; // ok, foo() is an lvalue
int* p1 = &foo(); // ok, foo() is an lvalue

// rvalues:
//
int foobar();
int j = 0;
j = foobar(); // ok, foobar() is an rvalue
int* p2 = &foobar(); // error, cannot take the address
                     // of an rvalue
j = 42; // ok, 42 is an rvalue
```

Important Features of C++: rvalue Reference

- If `X` is any type, then `X&&` is called an *rvalue reference* to `X`. For better distinction, the ordinary reference `X&` is now also called an *lvalue reference*.
- An *rvalue reference* is a type that behaves much like the ordinary reference `X&`, with several exceptions.
- The most important one is that when it comes to function overload resolution, *lvalue*s prefer old-style *lvalue references*, whereas *rvalues* prefer the new *rvalue references*:
  For example,

```
void foo(X& x);  // lvalue reference overload
void foo(X&& x); // rvalue reference overload

X x;
X foobar();

foo(x); // argument is lvalue: calls foo(X&)
foo(foobar()); // argument is rvalue: calls foo(X&&)
```

- *Rvalue references* allow a function to branch at compile time (via overload resolution) on the condition "Am I being called on an *lvalue* or an *rvalue*?"

Source: C++ Rvalue References Explained

Important Features of C++: rvalue Reference

- In the context of:

```
X bar();
X& fun();
```

- References behave as follows:

Abstraction	Application

`void foo(X&);`	`foo(bar()); // r-value: ERROR`
	`foo(fun()); // l-value: OKAY: void foo(X&);`

`void foo(const X&);`	`foo(bar()); // r-value: OKAY: void foo(const X&);`
	`foo(fun()); // l-value: OKAY: void foo(const X&);`

`void foo(X&);`
`void foo(const X&);`
	`foo(bar()); // r-value: OKAY: void foo(const X&);`
	`foo(fun()); // l-value: OKAY: void foo(X&);`

`void foo(X&&);`	`foo(bar()); // r-value: OKAY: void foo(X&&);`
	`foo(fun()); // l-value: ERROR`

`void foo(X&);`
`void foo(X&&);`
	`foo(bar()); // r-value: OKAY: void foo(X&&);`
	`foo(fun()); // l-value: OKAY: void foo(X&);`

`void foo(const X&);`
`void foo(X&&);`
	`foo(bar()); // r-value: OKAY: void foo(X&&);`
	`foo(fun()); // l-value: OKAY: void foo(const X&);`

`void foo(X&);`
`void foo(const X&);`
`void foo(X&&);`
	`foo(bar()); // r-value: OKAY: void foo(X&&);`
	`foo(fun()); // l-value: OKAY: void foo(X&);`

Important Features of C++: rvalue Reference

- While we can overload any function using *rvalue reference*, in the overwhelming majority of cases, this kind of overload should occur only for copy constructors and assignment operators, for the purpose of achieving move semantics:
  For example,

```
X& X::operator=(X const & rhs); // classical implementation
X& X::operator=(X&& rhs)
{
  // Move semantics: exchange content between this and rhs
  return *this;
}
```

  Implementing an *rvalue reference* overload for the copy constructor is similar.

Important Features of C++: Sample of Move Semantics

PoPL-06

Partha Pratim Das

Type Systems
Type & Type Error
Type Safety
Type Checking
Type Inference

Type Inference
add x = 2 + x
apply (f, x)
Inference Algorithm
Unification

Examples
sum
length
append
Homework

Type Deduction
Polymorphism
Ad-hoc
Parametric
Subtype
C++11,...

```
#include <iostream>
#include <cstring>
using namespace std;

class String { char *ptr; public:
    String(char *ptr_) : ptr(strcpy
        (new char[strlen(ptr_) + 1], ptr_)) {
        cout << "ctor: " << ptr << endl; }
    ~String() { cout << "dtor: ";
        if (ptr) cout << ptr; cout << endl;
        delete ptr; }
    String(const String& s) : ptr(strcpy
        (new char[strlen(s.ptr) + 1], s.ptr)) {
        cout << "c-ctor: " << ptr << endl; }
    String(String&& s) : ptr(s.ptr) {
        cout << "m-ctor: " << ptr << endl;
        s.ptr = nullptr; }
    String& operator=(const String& s) {
        cout << "c-assign: " << ptr << endl;
        if (&s == this) return *this;
        delete ptr; // Release resource
        ptr = new char[strlen(s.ptr) + 1];
        strcpy(ptr, s.ptr); // Copy
        return *this; }
    String& operator=(String&& s) {
        cout << "m-assign: " << ptr << endl;
        if (&s == this) return *this;
        char *tptr = ptr; // Exchange
        ptr = s.ptr; s.ptr = tptr;
        return *this; }
};
```

```
String GenStr(char *ptr) {
    cout << "G-str: " << ptr << endl;
    String s(ptr); return s;
}
int main() {
    String s("red"); cout << endl;
    String t = s; cout << endl;
    String u = GenStr("green"); cout << endl;
    s = GenStr("blue"); cout << endl;
    t = s; cout << endl;
    u = move(s); cout << endl;
    return 0;
}
```

Output Trace

```
ctor: red

c-ctor: red

G-str: green
ctor: green
m-ctor: green
dtor:

G-str: blue
ctor: blue
m-ctor: blue
dtor:
m-assign: red
dtor: red
```

```
c-assign: red

m-assign: green

dtor: blue
dtor: blue
dtor: green
```

Important Features of C++: Universal Reference

- Given that *rvalue references* are declared using "&&", it seems reasonable to assume that the presence of "&&" in a type declaration indicates an *rvalue reference*. That is not the case:

```
Widget&& var1 = someWidget;       // && means rvalue reference

auto&& var2 = var1;               // && does not mean
                                  // rvalue reference

template<typename T>
void f(std::vector<T>&& param);   // && means rvalue reference

template<typename T>
void f(T&& param);                // && does not mean
                                  // rvalue reference
```

- "&&" in a type declaration sometimes means rvalue reference, but sometimes it means either rvalue reference or lvalue reference.
- References where this is possible are more flexible than either lvalue references or rvalue references. Rvalue references may bind only to rvalues, for example, and lvalue references, in addition to being able to bind to lvalues, may bind to rvalues only under restricted circumstances. In contrast, references declared with "&&" that may be either lvalue references or rvalue references may bind to anything. We call them *universal references*.