

# Lecture 13

## Type system

Iztok Sarnik, FAMNIT

May, 2024.

# Literature

- John Mitchell, Concepts in Programming Languages, Cambridge Univ Press, 2003 (Chapter 6)
- Michael L. Scott, Programming Language Pragmatics (3rd ed.), Elsevier, 2009 (Chapter 7)
- Emmanuel Chailloux, Pascal Manoury, Bruno Pagano, Developing Applications With Objective Caml, O'REILLY & Associates, 2000

# Outline

1. Introduction
2. Type equivalence and compatibility
3. Type inference

# Introduction

- At least three ways to think about types
- Denotational
  - Type is simply a set of values
  - Value has a given type if it belongs to the set
- Constructive
  - Type is either one of a small collection of built-in types
  - Created by applying a type constructor ( record , array , set, etc.) to one or more simpler types
- Abstraction-based
  - Type is a data structure
  - Type is an interface consisting of a set of operations with well-defined semantics

# Why types?

- Naming and organizing concepts
  - Structuring data
  - Documenting data organization
- Consistent interpretation of data (bit sequences) in computer memory
  - Memory layout for accessing data
- Providing information to the compiler about data manipulated by the program
  - Ensuring the compatibility of operation operands
    - Type inference, type-checking, type errors
  - Dynamic binding and dynamic type checking
  - Types and optimization
  - Locating references for garbage collection

# Why types?

There are two basic functions of types:

1. Types provide **implicit context** for many operations, so that the programmer does not have to specify that context explicitly
  - $a + b$ , `new p`, ...
  - Choice of operation, sizes of structures, etc.
2. Types **limit the set of operations** that may be performed in a semantically valid program
  - Expressions and values with attached “meaning”
  - Typing and then catching type errors significantly improves code
  - Catch nonsensical operation (`1+”banana”`)

# Type systems

- A **type system** consists of:
  1. A mechanism to define types and associate them with certain language constructs
  2. A set of rules for type equivalence, type compatibility, and type inference
  3. A type-checking algorithm
- Which constructs have types?
  - Imperative language: those that have values
    - named constants, variables, record fields, parameters, subroutines, expressions
  - Functional languages: any expression has a type
    - Functions, values, expressions, statements, classes, modules

# Two kinds of type systems

- Two lambda calculus with types:
  - Implicit types, Curry Haskell
  - Explicit types, Alonzo Church
- **Implicit types**
  - Or, Curry type annotations
  - Optional type annotations
  - Type annotations are added where needed
  - Types are derived from expressions
  - Sophisticated type inference algorithms
  - ML, Haskell, Ocaml
    - Functional languages (not Lisp)



# Two kinds of type systems

- **Explicit types**
  - Or, Church type annotations
  - Strict type annotations
  - Language implementations include verification of types of variables, expressions, etc.
  - Types derived from expressions must be equivalent or compatible to annotations
  - Imperative languages usually use explicit type annotations; also most of object-oriented languages
  - *Examples:* Pascal, C, C++, Java, Scala

# Type checking

- When an object of a certain type can be used in a certain context?
- At this point the following three procedures are needed to judge the position
  - Type **equivalence** and/or **compatibility**
  - Type **inference**
- At a minimum, the object can be used if its type and the type expected by the context are **equivalent**
  - Compatibility is a looser relationship than equivalence
  - Objects and contexts are often compatible even when their types are different

# Type checking

- Type **compatibility** is the one of most concern to programmers
  - Type compatibility can involve: **type conversion (cast) and coercion**
- Type **inference** procedure computes type of an expression constructed from simpler subexprs
  - Given the types of the sub-expressions (and possibly the type expected by the surrounding context), what is the type of the expression as a whole?
- **Type-checking** procedure
  - Given a program, checks all expressions that have types by using type equivalence, compatibility and type inference

# Type checking

- Type checking is the process of ensuring that a program obeys the language's type compatibility rules
- A language is **strongly typed**
  - Prohibits the application of any operation to any object that is not intended to support that operation
- A language is said to be **statically typed**
  - Type checking is performed at compile time
  - Compile-time type checking catches errors earlier than run-time type checking
  - In statically typed language usually some type checking is done in run-time
  - Pascal, C, C++, Java, Scala, C++

# Type checking

- **Dynamic** (run-time) **type checking**
  - Types are checked at run-time
  - A form of late binding
  - Tends to be found in languages that delay other issues until run time as well
  - Lisp, Smalltalk are dynamically typed
  - Most scripting languages are dynamically typed
  - Python, Ruby are also strongly typed

# Type checking and polymorphism

- Polymorphism
  - Single body of code works with objects of multiple types
    - It may or may not imply the need for run-time type checking
- Dynamic typing
  - Supports implicit parametric polymorphism
    - Types can be thought of as implied (unspecified) parameters
    - Types of arguments are checked in run-time
  - Powerful and straightforward
    - Operation implementation is selected at run-time
  - Languages Lisp, Smalltalk, Script languages, etc.
  - Significant run-time cost for type checking

# Type checking and polymorphism

- Subtype polymorphism in OO languages
  - Given a straightforward model of inheritance
  - Type checking for subtype polymorphism can be implemented entirely at compile time.
- Explicit parametric polymorphism
  - A class is specified by using type parameters
  - Generics in C++, Eiffel, Java, and C#
  - Useful as a base class for the containers
  - Compile-time static type checking suffices
    - Similarly to subtype polymorphism

# Type checking and polymorphism

- ML family
  - Sophisticated system of type inference
  - ML compiler infers for every expression a type
    - With rare exceptions, the programmer need not specify the types of objects explicitly
  - Task of the compiler is to determine whether there exists a consistent assignment of types to expressions
    - This guarantees, statically, that no operation will be applied to a value of an inappropriate type at run time
    - Formalized as the problem of unification
  - Implicit parametric polymorphism with static typing
    - Computes the most general types
    - Derives type variables if there is no other constraints



# Outline

1. Introduction
2. Type equivalence and compatibility
3. Type inference

# Type equivalence

- Two principal ways of defining type equivalence
- **Structural equivalence** is based on content of type definitions
  - Two types are the same if they consist of the same components
  - Algol-68, Modula-3, C and ML
- **Name equivalence** is based on the lexical occurrence of type definitions
  - More popular approach in recent languages
  - Java, C#, standard Pascal, and most Pascal descendants, including Ada

# Structural equivalence

- Exact definition varies from one language to another
  - ML says Ok; most languages say error
- Two types are **structurally equivalent**
  - Replace any embedded type names with their definitions, recursively
  - Until nothing is left but type constructors, field names, and built-in types
  - Then, compare structures
  - Problem is an inability to distinguish between types that the programmer may think of as distinct

```
/* Pascal */  
type R2 = record  
  a, b : integer  
end;  
/* same as? */  
type R3 = record  
  a : integer;  
  b : integer  
end;  
/* what about this? */  
type R4 = record  
  b : integer;  
  a : integer  
end;
```

```
type student = record  
  name, address : string  
  age : integer  
type school = record  
  name, address : string  
  age : integer  
x : student;  
y : school;  
...  
x := y;  /* ? */
```

# Name equivalence

- **Assumption:**
  - If programmer writes two definitions (for the same type) then they are meant to represent different types
- **Example:**
  - Variables  $x$  and  $y$  are of different type and (under name equivalence) therefore we have type-checking error
- Name equality means that two type names are considered equal in type checking only if they are the same

```
type student = record
  name, address : string
  age : integer
type school = record
  name, address : string
  age : integer
x : student;
y : school;
...
x := y; /* ? */
```

# Variants of name equivalence

- There are two variants of name equivalence
- The simplest of type definitions

```
TYPE new_type = old_type;  /* Modula-2*/
```

- Here new\_type is said to be an alias for old\_type
  - Should we treat them as two different names or the names of the same type?

- **Strong name equivalence**
  - Treat them strictly as different types

- **Loose name equivalence**
  - Treat them as two names of the one type

```
TYPE celsius_temp = REAL;  
      fahrenheit_temp = REAL;  
VAR c : celsius_temp;  
      f : fahrenheit_temp;  
  
...  
f := c; /* error? */
```

```
TYPE stack_element = INTEGER; (* alias *)
```

# Type conversion (type cast)

- **Explicit type conversion!**
- There are many contexts in which values of a specific type are expected
  - We expect right-hand side to have the same type as a

`a := expression`
  - The overloaded + symbol designates either integer or floating-point addition

`a + b`

    - both integers or both reals

`foo(arg1, arg2, . . . , argN)`
  - We expect the types of the arguments to match those of the formal parameters
- Suppose in each of these cases that the types (expected and provided) are exactly the same

# Type conversion (type cast)

- To use a value of one type in a context that expects another we can use explicit type conversion (or, **type cast** )
  1. Types employ the same low-level representation, and have the same set of values
    - No code will need to be executed at run time
  2. Types have different sets of values, but the intersecting values are represented in the same way
    - One type may be a subrange of the other
    - Run-time check of exact types; can generate run-time error
  3. Types have different low-level representations but there is correspondence among the values
    - integer  $\leftrightarrow$  floating-point

# Example

```
type test_score = 0..100;  
    workday = mon..fri;  
type celsius_temp is new integer;  
type fahrenheit_temp is new integer;
```

```
/* Ada */  
n : integer;           -- assume 32 bits  
r : real;              -- assume IEEE double-precision  
t : test_score;        -- as in Example 7.9  
c : celsius_temp;      -- as in Example 7.20  
...  
t := test_score(n);    -- run-time semantic check required  
n := integer(t);       -- no check req.; every test_score is an int  
r := real(n);          -- requires run-time conversion  
n := integer(r);       -- requires run-time conversion and check  
n := integer(c);       -- no run-time code required  
c := celsius_temp(n);  -- no run-time code required
```



# Type Compatibility

- Most languages do not require equivalence of types in every context
- Value's type must be compatible with that of the context in which it appears
  - Left and right side of assignment statement
  - Values used in arithmetic operations
  - Actual parameter in function call
- The definition of type compatibility varies greatly from language to language

# Coercion

- Language allows a value of one type to be used in a context that expects another
  - Language implements automatic, **implicit conversion** to the expected type!
  - Run-time code must perform a dynamic semantic check, or convert between low-level representations
- OCaml provides **explicit coercion**
  - Coercion operator “:>”
  - Programmer has to take care of conversions
    - Avoiding errors that are hard to find
  - Base types and objects can be coerced
  - Separate operations for separate types (+, +., ...)
  - More in chapter on OO languages

(name : sub type :> super type ) (name :> super type )
---

# Coercion

- C++ provides an extremely rich, programmer-extensible set of coercion rules
  - Coercion code can be defined when new type is defined
  - This makes C++ flexible
  - One of the most difficult C++ features to understand and use correctly
  - Rules interact in complicated ways with the rules for resolving overloading

# Coercion in C

- C performs quite a bit of coercion

```
short int s;  
unsigned long int l;  
char c;    /* may be signed or unsigned -- implementation-dependent */  
float f;    /* usually IEEE single-precision */  
double d; /* usually IEEE double-precision */  
...  
s = l; /* l's low-order bits are interpreted as a signed number. */  
l = s; /* s is sign-extended to the longer length, then  
       its bits are interpreted as an unsigned number. */  
s = c; /* c is either sign-extended or zero-extended to s's length;  
       the result is then interpreted as a signed number. */  
f = l; /* l is converted to floating-point. Since f has fewer  
       significant bits, some precision may be lost. */  
d = f; /* f is converted to the longer format; no precision lost. */  
f = d; /* d is converted to the shorter format; precision may be lost.  
       If d's value cannot be represented in single-precision, the  
       result is undefined, but NOT a dynamic semantic error. */
```

# Coercion in Fortran

- Fortran allows arrays and records to be intermixed if their types have the same shape
- Two arrays are of the same shape
  - The same number of dimensions, elements and the same shape of element type
- Two records have the same shape
  - The same number of fields, and the fields are of the same shape
  - Field names do not matter, nor do the actual high and low bounds of array dimensions

# Trends in coercion use

- Modern compiled languages display a trend toward static typing and away from type coercion
- Some language designers argue that coercions are a natural way in which to support abstraction and extensibility
  - It is easier to use new types together with existing ones
  - This is especially true for scripting languages

# Outline

1. Introduction
2. Type equivalence and compatibility
3. Type inference

# Type inference

- Type inference is used for type-checking
  - The process of determining the types of expressions based on the known types
  - Inferred types are compared to types expected in a given context
- There are two general **approaches** to type inference:
  - 1) Type inference algorithms is based on typing rules that derive concrete (ground) types
    - Pascal, Java, C, C++,
  - 2) Type inference algorithms based on typing rules that derive parametrized types
    - ML, Ocaml, Haskell



# Type inference based on rules

- Type of an expression is inferred by means of [typing rules](#)
- During compilation expression is parsed into abstract syntax tree (AST)
  - AST is used to attach the type to each of sub-expressions
    - Check the lecture on Compilers and interpreters
- Types are computed bottom-up
  - A type of an expression is computed from types of its sub-expressions
    - Atomic types are either specified or can be determined from values
  - Typing rules act as patterns that match given syntactic constructions

# Typing rules

- Typing rules concern **judgments** of the form

$$\Gamma \vdash e : T$$

- where  $\Gamma$  is a context, which contains e.g. typings of identifiers
- The judgment says: in the environment  $\Gamma$ , expression  $e$  has type  $T$
- Judgments are used in **typing rules** of the form

$$\frac{J_1 \ J_2 \ \dots \ J_n}{J} \quad C$$

- $J_i$  are called premises,  $J$  is called conclusion and  $C$  condition

# Typing rules

- Example rule:
  - If  $x$  and  $y$  have type  $\text{int}$  then  $x+y$  has type  $\text{int}$

$$\Gamma = x:\text{int}, y:\text{int}$$
$$\frac{\Gamma \vdash x:\text{int} \quad \Gamma \vdash y:\text{int}}{\Gamma \vdash x + y : \text{int}}$$

- Context  $\Gamma$  is written in the form

$$\Gamma = x_1:T_1, x_2:T_2, \dots, x_n:T_n$$

- Judgement form for typing is generalized to

$$\Gamma \vdash e:T$$

- To add a new variable to the context  $\Gamma$ , we write

$$\Gamma, x : T$$

# Example

- Type checking rules for arithmetic expressions

$$\frac{\Gamma \vdash e1 : \text{int} \quad \Gamma \vdash e2 : \text{int}}{\Gamma \vdash e1 + e2 : \text{int}} \qquad \frac{\Gamma \vdash e1 : \text{int} \quad \Gamma \vdash e2 : \text{int}}{\Gamma \vdash e1 * e2 : \text{int}}$$
$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{}{\Gamma \vdash i : \text{int}} \text{ } i \text{ is an integer literal}$$

- Derivation of judgment:  $\Gamma = x : \text{int}, y : \text{int} \Rightarrow x + 12 * y : \text{int}$

$$\frac{\Gamma \vdash x : \text{int} \quad \frac{\Gamma \vdash 12 : \text{int} \quad \Gamma \vdash y : \text{int}}{\Gamma \vdash 12 * y : \text{int}}}{\Gamma \vdash x + 12 * y : \text{int}}$$

# Typing functions

- Type of function with one parameter is written

$$f : T_1 \rightarrow T_2$$

- The typing rule for function says: if  $x$  has type  $T_1$  and  $f$  has type  $T_1 \rightarrow T_2$ , then  $f(x)$  has type  $T_2$

$\frac{\Gamma \vdash x:T_1 \quad \Gamma \vdash f : T_1 \rightarrow T_2}{\Gamma \vdash f(x) : T_2}$
--

- Typing rule for functions with more than one parameters (one parameter seen as tuple)

$$f : T_1 * \dots * T_n \rightarrow T$$

# Typed $\lambda$ -calculus

$$\frac{x:T \in \Gamma}{\Gamma \vdash x:T} \text{ (axiom)}$$

$$\frac{\Gamma \vdash M : (\sigma \rightarrow \tau) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (M N) : \tau} \text{ (}\rightarrow\text{-elimination)}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : (\sigma \rightarrow \tau)} \text{ (}\rightarrow\text{-introduction)}$$

# Type inference in ML

- ML type inference algorithm derives **most general** parametrized type of expression
  - H. Curry, R. Feys, R. Hindley, R. Milner
    - Hindley-Milner type system
  - Type inference can be applied to a variety of programming languages
- ML type inference supports **polymorphism**
  - Type variables are used as place-holders for types that are not known
- Algorithm will be presented by examples

# Example 1

- Type of 2 is int
- Operator + is

```
- fun f1(x) = x + 2;  
val f1 = fn : int → int
```

overloaded but since we have one integer, then it must have type  $\text{int} \rightarrow (\text{int} \rightarrow \text{int})$

- Therefore, x must be of type int
- Putting this together we get that f1 is of type  $\text{int} \rightarrow \text{int}$



# Example 2

```
- fun f2(g,h) = g(h(0));  
val f2 = fn : ('a → 'b) * (int → 'a) → 'b
```

- Type of 0 is int
- The type of function h result is not known, therefore we write 'a
- Since the result of h is an argument of g then the domain of g is 'a
- Also the type of g is not known so we take 'b
- Since type of g result is 'b then also result of f2 is of type 'b
- We get the type  $('a \rightarrow 'b) * (int \rightarrow 'a) \rightarrow 'b$

# Type-Inference Algorithm

1. Assign a type to the expression and each subexpression
  - For any compound expression or variable, use a [type variable](#)
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-based algorithm for solving systems of equations

# Example 3

```
- fun g(x) = 5 + x;
val g = fn : int → int
```

Subexpression	Type
$\lambda x. ((+ 5) x)$	$r$
$((+ 5) x)$	$s$
$(+ 5)$	$t$
$+$	$\text{int} \rightarrow (\text{int} \rightarrow \text{int})$
$5$	$\text{int}$
$x$	$u$

**Function Application:** If the type of  $f$  is  $a$ , the type of  $e$  is  $b$ , and the type of  $f(e)$  is  $c$ , then we must have  $a = b \rightarrow c$ .

**Lambda Abstraction:** If the type of  $x$  is  $a$ , the type of  $e$  is  $b$ , and the type of  $\lambda x.e$  is  $c$ , then we have  $c = a \rightarrow b$ .

**(2) We get constraints:**

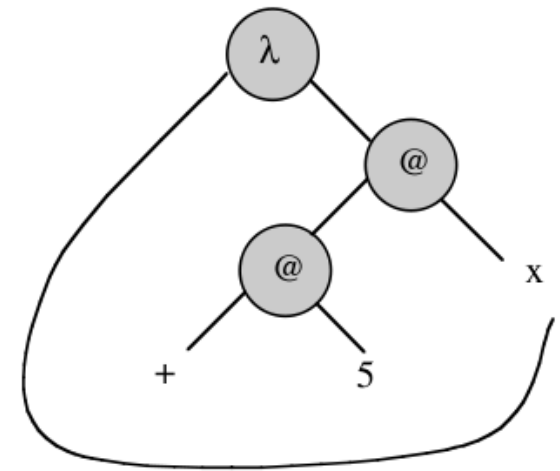
Subexpression  $(+5)$ :  $\text{int} \rightarrow (\text{int} \rightarrow \text{int}) = \text{int} \rightarrow t$

Subexpression  $(+5) x$ :  $t = u \rightarrow s$

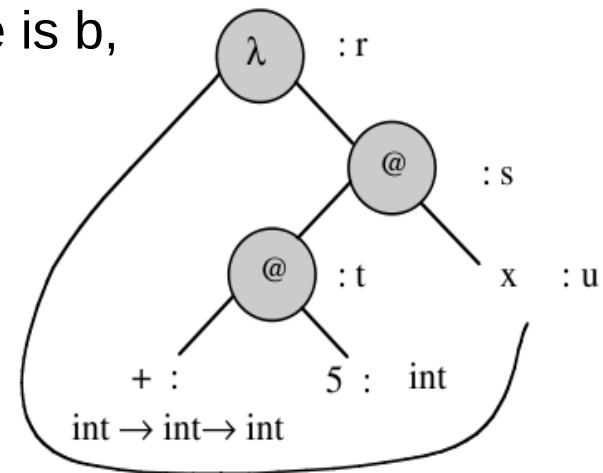
Subexpression  $\lambda x.((+5)x)$ :  $r = u \rightarrow s$

**(3) Solve equations**

$t = \text{int} \rightarrow \text{int}$ ,  $u = \text{int}$ ,  $s = \text{int}$ ,  $r = \text{int} \rightarrow \text{int}$



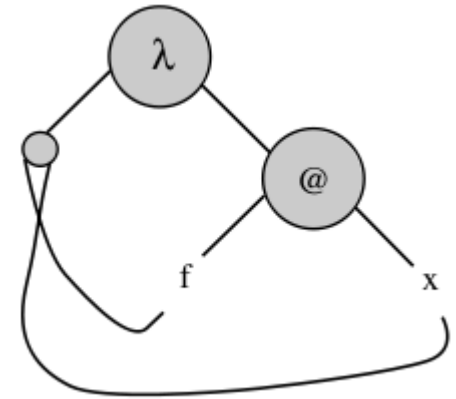
**(1)**



# Example 4

```
- fun apply(f,x) = f(x);
val apply = fn : ('a → 'b) * 'a → 'b
```

Subexpression	Type
$\lambda \langle f, x \rangle. fx$	$r$
$\langle f, x \rangle.$	$t \times u$
$fx$	$s$
$f$	$t$
$x$	$u$



(2) Generate constraints:

$t = u \rightarrow s$

$r = t * u \rightarrow s$

(3) Solve constraints:

$r = (u \rightarrow s) * u \rightarrow s$

$r = ('a \rightarrow 'b) * 'a \rightarrow 'b$

