Note 3

Types

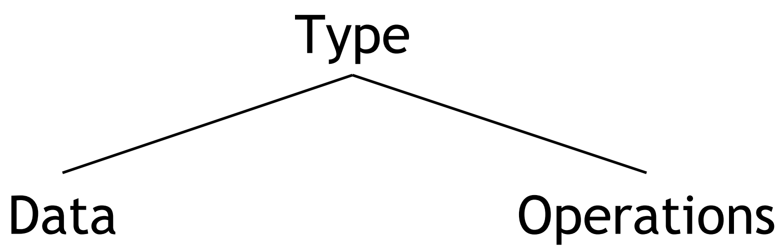
Yunheung Paek
Associate Professor
Software Optimizations and Restructuring Lab.
Seoul National University

so&r

Topics

- Definition of a type
- Kinds of types
- Issues on types
- Type checking
- Type conversion

Components of a data type



- a set of data objects that model a collection of abstract objects in real world

  ex: in C language

  - **`int`** ↔ integers, student id, exam scores, ...
  - **`char[]`** ↔ letters, names, ...

- a set of operations that can be applied to the objects

  ex: **`+ - * /`** ↔ add, subtract, multiply, divide for integers

Using types ...

- improves readability and writability.
  - ex:

```
char* student_name;
struct employee_records {
    char* name;
    int salary;
    . . .
}
```

- reduces programming errors.
  - ex:

```
student_name / 5
```

- makes memory allocation and data access efficient.
  - ex:

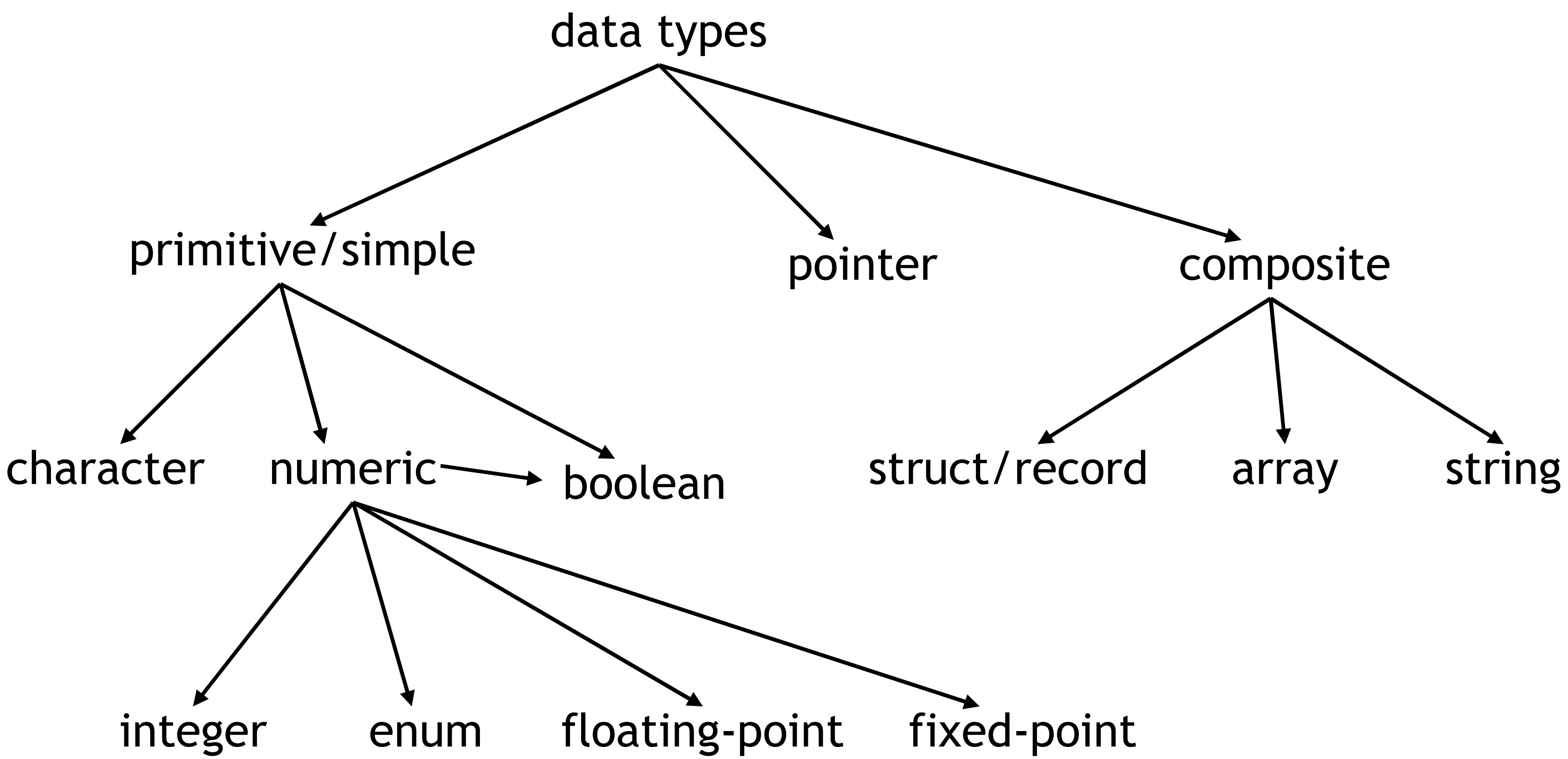
```
struct {
    int i;
    char* c;
}
```

// 8 bytes → Sizes are statically known.

→ useful for the compiler to optimize memory allocation (e.g., use stack in stead of heap)

Hierarchies of types

- Most imperative languages (C/C++, Ada, …)

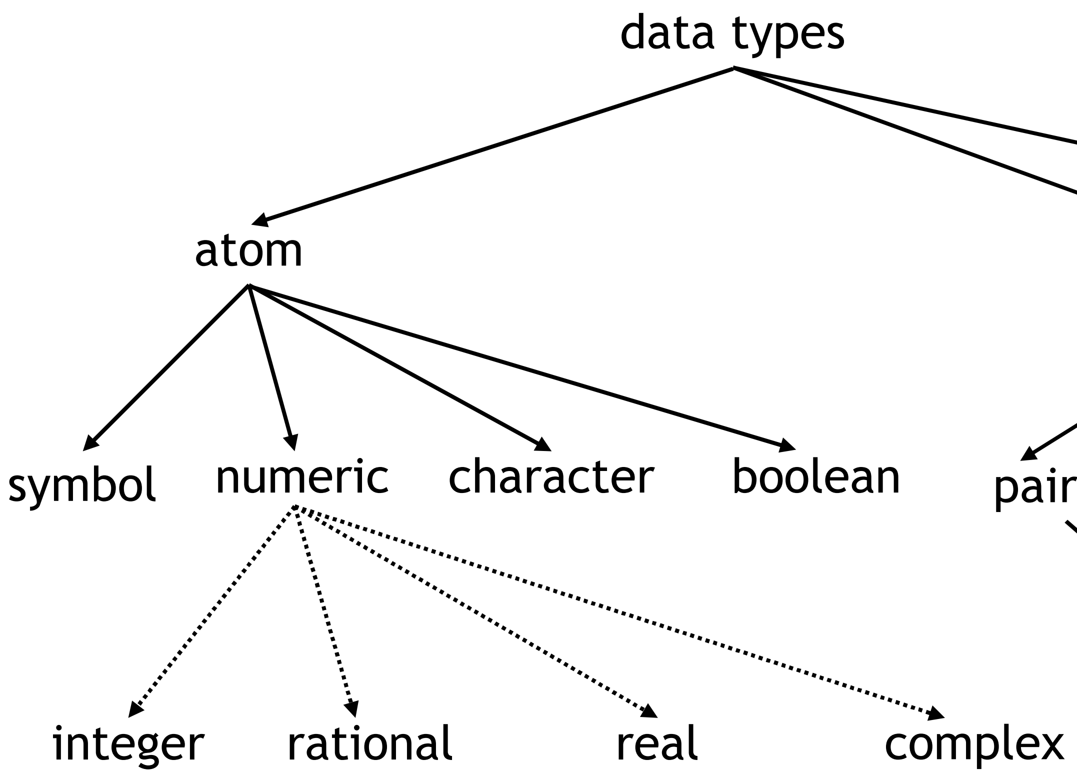


C/C++ → char, int/long, float/double, struct/class, array, string(?)
Java → … boolean

no clear distinction between char and int, no special op for char/string

Hierarchies of types

- Functional languages (Scheme, LISP, ML, …)



More atomic data types to support symbolic computations (e.g., rational, complex)

Selection of data types

- "What kinds of types should be included in a language" is very important for programming language design.
- Primitive/simple types are supported in almost all existing programming languages
- Composite types being supported differ from language to language based on what is the purpose of the language.
- Several issues related to the selection of types
  - fixed-point vs. floating-point real numbers
  - array bounds
  - structure of composite types
  - pointer types
  - subtypes

Cobol for string type ?
class type for OO languages

Fixed-point vs. Floating-point

- fixed-point
  - Precision and scale are fixed.
    → a fixed radix point for all real numbers of the same type
  - ex: salary amount of graduate assistants
    → 6 digits for precision and 2 digits for scale
      → 1234.56, 2000.00
- floating-point
  - radix points are floating
  - ex: 21.32, 9213.1, 4.203e+9
- COBOL, PL/1 and Ada support the fixed-point real type, but most of other languages (Fortran, C, ...) don't.
  - Ada: `type salary is delta 0.01 range 0.0..3000.0`
  - C++: `float salary;`

Fixed-point vs. Floating-point

- Problem with fixed-point
  - possible loss of information after some operations at run-time
    - ex: *double the salary of EE students!*
- Problem with floating-point
  - Large numbers may be machine-dependent.
    - ex: *port a C-program to 32-bit and 64-bit machines!*
  - Less secure
    - ex: *double the salary illegally*

Determination of array bounds

- static arrays (C, Fortran, Pascal)
  - array bounds determined at compile time and static storage allocation. → efficient
  - ex: `int a[10], b[5];`
- stack-dynamic (Ada)
  - array bounds determined at run time but static storage allocation
  - ex:

```
read size;
call foo(size);
        . . .
subroutine foo(int size)
int a[size], b[size*2];
```

- dynamic (C, Fortran90)
  - array bounds determined at run time and dynamic storage allocation
  - ex:

```
int *a, *b;
a = b = new int[10]
        . . .
delete [] a;
b = new int[20]
```

Array bounds as a data type

- Pascal includes array bounds as a data type, which implies
  1. All arrays in Pascal are static.
  2. Errors in illegal array assignments and parameter passing can be detected at compile time.

  ex:

```
var x, y : array of [1..30] of real;
var z : array of [1..30] of real;
function foo(w : array of [1..20] of real): integer;
              . . .
begin
   i := foo(x) + 10;  // illegal
   x := z;            // illegal
   x := y;            // legal
```

```
type vector30 = array of [1..30] of real;
var x, y: vector30;
var z: vector30;
      . . .
x := z;               // legal
```

- Most other languages do not include array bounds as a type
  - needs more complex memory managements and error-prone

    ex:

```
real x(1:10)
x(i) = 5.1
```

    → *What if* `i > 10`*?*
  - but more flexible → *consider the function* `foo` *in the Pascal code*

Structure of composite types

- Is this assignment legal?

```
struct man { char* name; int age; }
struct woman { char* name; int age; }
man Tom;
woman Jane;
   . . .
Jane = Tom;
```

- In Ada and the early Pascal, the answer is "no".
  → *name equivalence/compatibility*
- In most others like C, the answer is "yes".
  → *structure equivalence*
- The name equivalence provides
  - easy type checking by string comparison → fast compilation
  - more secure and less error-prone compilation → `Jane = Tom;` *(unsafe!)*
  - less flexible programming
    - No anonymous type is allowed. → cf: C/C++
    - Type must be globally defined. → Why?

```
struct {
   char* name;
   int age;
} Tom;
```

The pointer type

- In PL/1, the pointer has no data type.

```
declare p pointer;
declare i integer;
declare c, d char;
    . . .
p = address of(c);
    . . .
p = address of(i);
d = dereference(p);
```

→ *Error won't be detected until run time*

  - *This is more flexible and may save memory, but is error-prone!*

- In C, the pointer type is a part of data types

```
int* p;
int i;
char c, d;
    . . .
p = &i;
d = *p;
```

→ *Error can be detected by the compiler*

  - *Most languages include the pointer type.*

Subtypes

- Primitive types provided by languages are not enough. Why?

```
int day, month;
month = 9;          // It's OK...but need more...
day = -11;          // Non-sense! Semantic error! may not be caught even at run time
```

  - *How can we capture this semantic error with data types?*

- Users need to restrict the primitive types.
  - enumerated types (C++, Pascal)
    - *C++*

```
enum day_type {first, second, . . . , thirty_first};
enum month_type {Jan, Feb, . . . , Dec};
day_type day;
month_type month;
month = Sep;          // That's better. More readable.
day = -11;            // Error detected at compile time!
```

    - *Pascal*

```
type month_type = (Jan, Feb, . . . , Dec);
```

  - subrange type (Pascal) - in some case, more compact and flexible

enum type for 1..99999?

```
subtype day_type is integer range 1..31;
        . . .
day := -11;           // Error still can be detected.
day := day + 20;      // Also it can be used in integer operation.
```

➔ *The problem is this may be error. But error can be easily detected at run-time.*

Monomorphic/polymorphic objects

- A *monomorphic* object (function, variable, constant, …) has a single type.
  - constants of simple types (character/integer/real): ‘a’, 1, 2.34, …
  - variables of simple types: **`int i;`** *(C)*, **`x :real;`** *(Pascal)*
  - various user-defined functions: **`int foo(char* c);`**
- A *polymorphic (generic)* object has more than one types.
  - the constant **`nil`** in Pascal and **`0`**(*integer,virtual function, pointer*) in C
  - the functions for lists in Scheme and Lisp: **`cons, car, cdr`**, …
  - the basic operators **`+, –, :=, ==, ^, *`**(*multiply, dereference*), …
  - subtype objects
    - subrange types
    - derived class objects in object-oriented languages

Type expressions

- A *type expression* describes how the representation for a monomorphic or polymorphic object is built.
- Examples of type expressions in real languages
  - simple types

    ```
    int, boolean, char*, ^real, ...
    ```
  - composite types

    ```
    array [...] of real
    char <name>[...]
    struct {...}
    record <name> is ... end <name> end record;
    ```
  - functions

    ```
    float <function name>(...) { ... }
    ```
- Type expressions are useful to formally represent monomorphic and polymorphic objects.

Type expressions

- The syntax of type expressions for monomorphic and polymorphic objects
  - *int*, *real*, *list* . . . denote basic types.
  - α, β, . . . denote *type variables*.
  - the type constructors →, X are used for functions
- *Ex:*

  726 : *int*

  "string" : *char∗*

  a : *list of real*

  foo : *char∗ → int*

  + : $\begin{cases} \text{real x real} \to \text{real} \\ \text{int x int} \to \text{int} \end{cases}$

  ***** : $\begin{cases} \text{real x real} \to \text{real} \\ \text{int x int} \to \text{int} \\ \alpha* \to \alpha \end{cases}$

  nil : $\alpha*$

  cons : ?

  length : ?

```
int foo(char* c) {
   float a;
   …
}
```

Type checking

- Recall → **data type** = set of **data objects** + set of **operators**

  Examples — *Type expressions* — *a function having a single, fixed type*

  - $* : int \times int \rightarrow int$ // type definition of a ***monomorphic function***
  - $\Sigma : list\ \alpha \rightarrow \alpha$ // type definition of a ***polymorphic function*** Σ — *a function having multiple types*

- A *data object* is **compatible** with an operator if the objects can be passed to the operator as the operands.

```
int i, j;
i * 3                  // legal
i * “string”           // illegal
Σ(1.3, 3.01, 2.0)      // legal
Σ(3.2, j, i)           // illegal
```

  - **Type error** occurs if an operator is applied to *incompatible* objects.
  - A program is **type safe** if it results in no *type error* while being executed.
  - **Type checking** is the activity of ensuring that a program is *type safe*.

Static vs. dynamic type binding

- static type binding
  - A variable …
    - is bound to a certain type by a declaration statement, and
    - should have only one type during its life time.

```
float x;                 // x is of a real type
char* x;                 // This is an error
```

  - most existing languages such as Fortran, PL/1, C/C++ and ML
- dynamic type binding
  - A variable …
    - is bound to a type when it is assigned a value during program execution, and
    - can be bound to as many types as possible.

```
> (define x 4.5)             // x is of a real type
> (define x '(a b c))        // now, x is of a list type
```

  - Scheme, LISP, APL, SNOBOL

Type inference

- implemented in functional programming languages with static type binding (e.g., ML and Miranda)
- Data types are inferred by the compiler without help from the user whenever possible.
  - *ML*

    ```
    5 - 3;
        2 : int
    ```

  - *Miranda*

    ```
    reverse list = rev list []
    rev [] n = n
    rev (l:m) n = rev m (l:n)
            rev : list × list → list
            reverse : list → list
    ```

- Type inference helps the user to enjoy advantages of dynamic type binding (→ *don't worry about assigning data types to variables*), while the compiler may better optimize code by statically inferring and binding data types for each variable.

Type inference

- Type parameters can be used for polymorphic functions,
  - *ML*

```
fun id(x) = x;
```

id : $\alpha \rightarrow \alpha$

- Error will occur if there is uncertainty that prevents type inference.
  - *ML*

```
fun add(x,y) = x + y;
```

error: variable '+' cannot be resolved

- To resolve uncertainty, more information needs to be provided by the user.

```
fun add(x,y) : int = x + y;
```

add : $int \times int \rightarrow int$

Static type checking

- type checking performed during compile time
  - Pascal, Fortran, C/C++, Ada, ML, …
  - The type of an expression is determined by static program analysis.
- To support static type checking in a language, a variable (or memory location) and a procedure must hold only one type of values, and this type must be statically bound or inferred.
  - *Pascal*

```
var x : x_type;
    w : array of [1..10] of real;
function foo(n : integer): real
        . . .
begin
   w[9] := foo(5) / w[1];          // OK
   w[10] := foo(x) + 3.4;          // error
   w[11] := w[foo(-2)] * 10.0;     // error
```

  - *Miranda*

```
reverse [a b c]        // return a list [c b a]
reverse 8              // error
```

  - *C++*

```
#include <stream.h>
main() {
        int i = bar();   // error: undefined function bar
        . . .
```

Dynamic type checking

- type checking performed during program execution
- required by languages that
  - perform dynamic type binding, or
    - *Scheme*

```
> (define a 10)
> (car a)
error: wrong arg to primitive car: 10
```

  - check the value of a program variable at run time.
    - *Pascal*

```
subtype day_type is integer range 1..31;
var day : day_type;
 i : integer;
. . .
day := i;                          // Is 1 ≤ i ≤ 31?
```

    - *Ada*

```
type x_type is (x1, x2);
type a_type (x : x_type) is           // discriminant
    record case x is
         when x1 => m : integer;
         when x2 => y : float;
     end case; end record;
a : a_type;
xn : x_type;
         . . .
a = (x => xn, m => 3);                // x = x1 ?
```

Static vs. dynamic type checking

- STC supports early detection of type errors at compile time.
  - Thereby ... → shortening program development cycle, and causing no run time overhead for type checking.
- STC guarantees a program itself is type safe.
- DTC only guarantees a particular execution of a program is type safe. Therefore, DTC must be repeated although the same program is executed.
- DTC needs extra space for special bits for each variable indicating the type of the variable at present.
- In general, STC allows greater efficiency in memory space and time.
- DTC handles the cases with unknown values that STC cannot handle.

Strongly vs. weakly typed languages

- strongly typed (Ada, ML, Miranda, Pascal) if all (or almost all w/ few exceptions like Pascal) programs are guaranteed to be type safe by either static or dynamic type checking.
- weakly typed or untyped (Fortran, C/C++, Scheme, LISP)
  - *Fortran*

```
real r
character c
equivalence (r,c)
print*, r;
```

// maybe wrong, but maybe still OK

  - *C++*

```
float foo(char cc, float x) { cout << cc << x; }
main() {
   float y = foo(100.7,'c');
```

// it runs! → output: **d 99**

char(100.7) → char(100) → 'd'
float('c') → 99

  - *Pascal*

```
type x_type = (x1, x2);
     a_type = record
          case x : x_type of
          x1 : (m : integer);
          x2 : (y : float);
     end case; end record;
var a : a_type;
   xn : x_type;
   . . .
a.m = 3;
```

// maybe right, maybe wrong

Overloading

- Often it is more convenient to use the same symbol to denote several values or operations of different types.
  - *Pascal*

    ```
    subtype day_type = 1..31
    ```

    → The numbers 1 ~ 31 are *overloaded* because the numeric symbols of type **day** are also of type **integer** in Pascal.
  - *C++*

    ```
    int ::operator+(int, int) {  . . . }
    float ::operator+(float, float) {  . . . }
    ```

    → This built-in symbol + is overload because it is used for the addition for integer and real types.
- In C++, the users can overload operators with the class construct.

```
class complex {
      . . .
      complex operator+(complex, complex);
}
```

Overloading

- Type checking tries to resolve ambiguities in an overloaded symbol from the context of its appearance.

```
day_type day;
day = 10;               // 10 is of type day
. . . 3 + 4             // integer addition
. . . 4.3 + 2.1         // real addition
```

→ If the ambiguity cannot be resolved, type error occurs.

Type conversion

- In order to allow **`3.46 + 2`** instead of **`3.46 + 2.0`**, one solution is to create extra two overloaded functions

```
float ::operator+(float, int) {  . . . }
float ::operator+(int, float) {  . . . }
```

→ But, this solution is tedious and may cause exponential explosion of the overloaded functions for each possible combination of types such as **`short, int, long, float, double, unsigned`**, … .

- A better solution: type conversion

→ *convert the types of operands.*

- Two alternatives for type conversion
  - **explicit**: type cast
  - **implicit**: coercion

Type cast

- Explicit type conversion
  - *C++*

```
float x = 3.46 + (float) 2;
int *ptr = (int *) 0xffffff;
x = x + (float) *ptr;
```

  - *Ada*

```
type day_type is new integer range 1..31
day_type day1, day2, day3;
integer d;
        . . .
day2 := day3 + 10;                  // 10 is overloaded
day1 := day2 + day_type(d);         // d is type casted
```

- Drawback of type cast
  - Heedless explicit conversion may invoke *information loss*. (e.g. truncation)
  - A solution? → implicit type conversion!
    - Languages provide *implicit type conversion (coercion)* to coerce the type of some of the arguments in a direction that has preferably no information loss.

Coercion

- In many languages (PL/1, COBOL, Algol68, C), coercions are the rule. They provide a predefined set of implicit coercion precedences. → *Generally, a type is widened when it is coerced.*
  - *C*

```
character → int
pointer → int
int → float
float → double
   . . .
```

But, it may still lose information.
ex: 32 bit integer → 32 bit float with 24 bit mantisa)

- The generous coercion rules make typing less strict, thereby possibly
  - causing run time overhead (e.g., COBOL),
  - making code generation more complicated, and
  - in particular, making security that typing provides lost by masking serious programming errors.

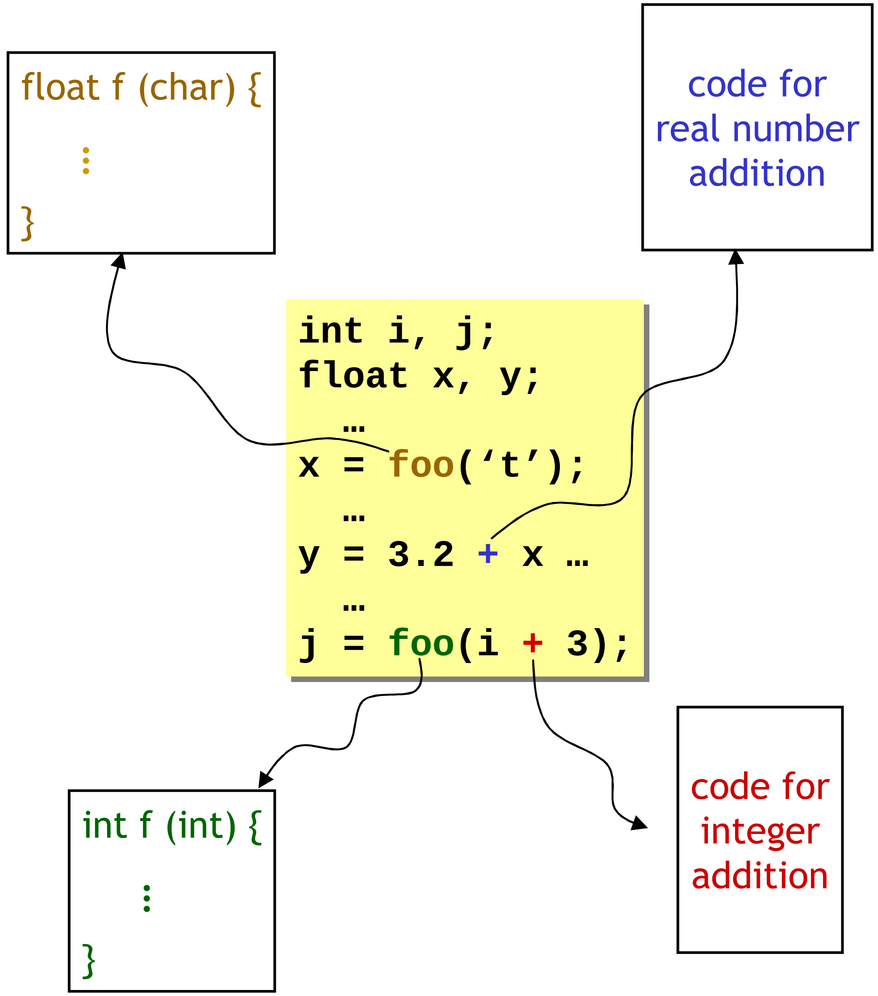
  → *For the last reason, some strongly typed languages (Ada, Pascal) has almost no coercion rules.*

Polymorphic functions

1. **ad-hoc** polymorphic functions that work on a finite number of types
   - *overloaded* functions
     - built-in → `+, *, …`
     - user-defined

       ```
       int foo(int i);
       float foo(char c);
       ```
   - functions with parameter *coercion*

     Ex: convert `real + int` to `real + real`
   - After the ambiguity is resolved, a *different piece of code* is used.



Polymorphic functions

2. **universal** polymorphic functions that work on an unlimited numbers of types
   - **inclusion** polymorphism is the type of polymorphism you're used to, where the same function can exist with the same signature in several child classes and act differently for each class
     → *subtypes*
   - **parametric** polymorphism is when a function accepts a variable as a parameter that can be of any valid type (e.g. variables in Scheme)
     → **`*`** (*dereference*), **`length, cons, car, …`**

Polymorphic functions

- **parametric** polymorphism
  → `*` (*dereference*), **`length`**, **`cons`**, **`car`**, **…**
- Typically, the *same code* is used regardless of the types of the parameters, and the functions exploit a *common structure* among different types.

  Ex: **`length`** assumes its parameters share a common data structure, a *list*

```
> (define l1 '(a b c))
> (define l2 '("t" 3))
> (define l3 '(l1 l2))
> (length l1)
3
> (length l2)
2
> (length l3)
2
```

code for **`length`**

```
(lambda (l)
  (if (null? l) 0
      (+ 1 (length (cdr l)))))
```