Types

Classification of Values

Values and Types

- Basic types: types of atomic values
  - int, bool, character, real, symbol
- Compound types: types of compound values
  - List, record, array, tuple, struct, ref, pointer
  - Built from type constructors
    - int arr[100] → arr: array(int,100)
    - **(3, 4, "abc")** : **int * int * string**
    - int *x → x : **pointer(int)**
    - **int f(int x) { return x + 5}** → f : **int→int**
- Values of different types
  - have different layouts
  - have different operations
- Explicit vs. implicit type conversion of values

Types in Programming

- A type is a collection of computable values that share some structural property
  - Represent concepts from problem domain
    - **Accounts, banks, employees, students**
  - Represent different implementation of values
    - **Integers, strings, floating points, lists, records, tuples …**
- Languages use types to
  - Support organization of concepts
    - Separate types for separate concepts from problem domain
  - Identify and prevent errors
    - Prevent meaningless computation

      3 + true - "Bill"
  - Support efficient translation (by compilers)
    - Short integers require fewer bits
    - Access record component by a known offset
    - Use integer units for integer operations

The Type System

- Each language has a type system that includes
  - A collection of basic types and compound types
  - For each basic/compound type, rules on
    - How to build values of the type
      - integers(eg.,1,23); floating point numbers(e.g., 3.5, 0.12)
      - Symbols(‘abc); chars (‘a’, ‘b’); strings(“abc”); lists: ‘(abc 3)
      - Type constructors for arrays, structs, records, etc.
    - How to operate on values of the type
      - Evaluation, equality, introduction and elimination operations
      - Each operation is defined only on specific types of operands and returns only a specific type of values
  - Introduction of new types (optional)
    - Type declaration rules on how to introduce new types
- Error checking
  - A *type error* occurs if an operation is applied to operands outside its domain

Type Declaration and Equivalence

- Type declarations: introduce new types(user-defined types)
  - Transparent declaration: introduce a synonym for another type
    - typedef struct { int a, b; } mystruct;
    - typedef mystruct yourstruct;
  - Opaque declaration: introduce a new type
    - struct XYZ { int a, b,c; };

- Type equivalence: struct s {int a,b; }=struct t {int a,b; } ?
  - Structural equivalence: yes
    - s and t are the same basic type or
    - s and t are built using the same compound type constructor with the same components
  - Name equivalence: no
    - S and t are different names
    - Names uniquely define compound type expressions
  - In C, name equivalence for records/structs, structural equivalence for all other types

Type Error

- When a value is misinterpreted or misused with unintended semantics, a type error
  - May cause hardware error

    function call x() where x is not a function

    - may cause jump to instruction that does not contain a legal op code
  - May simply return incorrect value

    int_add(3, 4.5)

    - not a hardware error
    - bit pattern of 4.5 can be interpreted as an integer
    - just as much an error as x() above

Type Safety Of Languages

- A language is type-safe if it never allows any undetected type error to occur at runtime
  - E.g., raise a runtime exception instead of segmentation fault
- Which languages are type-safe? Which are not?
  - BCPL family, including C and C++
    - Not type-safe: casts, pointer arithmetic, …
  - Algol family, Pascal, Ada
    - Almost type-safe
    - Dangling pointers: pointers to locations that have been de-allocated
    - No language with explicit de-allocation of memory is fully type-safe
  - Type-safe languages with garbage collection
    - Lisp, ML, Smalltalk, Java
    - Dynamically typed: Lisp, Smalltalk
    - Statically typed: ML, JAVA

Type Checking

- Type checking: discover and report type errors
  - Can be done at compile-time or run-time, or both
- Run-time(dynamic) type checking
  - Check type safety before evaluating each operation
  - Example: in Lisp/Scheme, before evaluating (car x), check to make sure x is a non-empty list
- Compile-time(static) type checking
  - Each variable/expression must have a single type: it can have only values of this type
  - Type system: rules for statically deciding types of expressions
    - Specify the proper usage of each operator
    - Reject expressions that cannot be typed according to rules
    - Explicit vs. implicit type conversion
  - Example: In C/C++/Java, if a function f is declared int f(float x), the compiler ensures that f is invoked only with float-type expressions

Static vs Dynamic Type Checking

- Both prevent type errors
- Run-time checking: check before each operation
  - Pros: flexibility and safety
    - Variables/expressions could have arbitrary types
    - Can detect all type errors (language is type safe)
  - Cons: slow down execution, and error detection may be too late
- Compile-time checking
  - Pros: efficiency (no runtime overhead) and early error detection
  - Cons: flexibility and safety
    - Every variable/function can have only a single type: need to define a different function for each input type
    - Cannot detect some type errors, e.g., accessing arrays out-of-bound, dangling pointers
- Combination of compile and runtime checking
  - Example: Java (array bound check at runtime)

Type Inference

- Static type checking in C/C++/Java

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

  - Programmer has to declare the types of all variables
  - Compilers evaluate the types of expressions and check agreement

- Type inference: extension to static type checking

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

  - Programmers are not required to declare types for variables
  - Compilers figure out agreeable types of all expressions
    - Solving constraints based on how expressions are used

A Simple Example

- What is the type of f in the Scheme code?

  ```
  (define f (lambda (x) (+ 2 x)))
  > f:  int → int
  ```

- How does it work
  - + has two types: int*int->int, real*real->real
  - 2 : int has only one type
  - This implies + : int*int -> int
  - Therefore, need x : int
  - Therefore f(x:int) = 2+x has type int → int

+ is overloaded because it has two types. Most operators in a static type system have a single type

Type Inference Example

- Function Definition
  - (define f (lambda (g x) (g (g x))))
  - f : (t → t)*t → t

Step 1: Assume a type for each variable:
  g : ‘g
  x : ‘x
  f : ‘f = ‘g * ‘x -> ‘f_ret
Step 2: Consider each operation and derive constraints on type variables:
  operation (g x) requires ‘g = ‘g_input -> ‘g_ret and ‘g_input = ‘x
      (i.e., g is a function which can take x as parameter)
  operation (g (g x)) requires ‘g_ret = ‘g_input
      (i.e., g can take g_ret as parameter)
    and ‘g_ret = ‘f_ret (i.e., g_ret is returned as f_ret)
Step 3: Group all equivalent types ‘g_input = ‘g_ret = ‘x = ‘f_ret
  f : (‘x -> ‘x) * ‘x -> ‘x

Type Inference Example

- Without knowing anything about variables, can we guess the type of each variable and expression?

```
(define Add (lambda (exp num)
  (cond ((null? exp) exp)
        ((cons? exp) (cons (Add (car exp) num) (Add (cdr exp) num)))
        ( (number? exp) (+ exp num))
        (else exp)))
```

Each pre-defined operator requires its operands to have specific types. E.g., (car x) ➔ x must be a list

(car exp)/(cdr exp) ➔ exp : list
(+ exp num) ➔ exp : number   num: number

So exp could be a number or a list
➔ type error in statically typed languages

Polymorphism

- A function (operator) is polymorphic if it can operate on different types of input values
  - Dynamic type checking supports arbitrary polymorphic functions.
  - Can we support polymorphic functions in compiled languages?
- Parametric polymorphism
  - Operate on types parameterized with type variables

    nil : ‘a list   cons : ‘a*(‘a list) → ‘a list
- Ad hoc polymorphism (operator overloading)
  - Reuse the same operator for different types; use a different implementation for each type definition

    + : int->int;   + : real->real
- Subtype polymorphism: define relations between types
  - Unify multiple types with a base type, e.g., C union, ML datatype
  - Inheritance in object-oriented programming (Truck is a subclass of Car)

```
void IncreaseSpeed(Car* c, int incr) { c->speed()+=incr; }
Truck truck; IncreaseSpeed(&truck, 50);
```

Parametric vs. Ad hoc Polymorphism

- Parametric polymorphism (type variables)

  ```
  (define first (lambda (x) (car x)))
  ```

  x: ‘a list (any kind of list);   first : ‘a list → ‘a

  A single implementation (algorithm) is used for all different types of input

- Ad hoc polymorphism (operator overloading)

  ```
  (define Add (lambda (x y)
                (if (number? y) (+ x  y)
                    (cons x y))))
  ```

  When applied to numbers:

  x: number; y : number; Add: number*number→ number

  When applied to lists

  x: ‘a; y : ‘a list; Add: ‘a * ‘a list → ‘a list

  Different implementations ((+ x y) vs. (cons x y)) are used for different types

- Dynamically typed languages (e.g., Lisp/Scheme) supports both parametric and ad hoc polymorphism
  - What about C/Java/C++?

Summary

- Types are important in modern languages
  - Program organization and documentation
  - Prevent program errors
  - Provide important information to compiler
- Static type checking and inference
  - Type checking
    - Based on types of variables and literal values, determine types of expressions
  - Type inference
    - Determine best type for an expression, based on known information about symbols in the expression
- Polymorphism
  - Parametric polymorphism
    - Single algorithm (function) can have many types
  - Overloading
    - Symbol with multiple meanings, resolved at compile time