Issues of programming languages
Data types

7.1-7.3; 8.1-8.7

Data types

- Introduction
- Type systems
- Type checking
- Interesting types
  - Records/arrays/strings/sets/pointers and recursive types etc.

Introduction

- What are types?
  - A type is a set of values. An expression of a program must have a type
- Why types
  - Type provides a context
    - For +
      - for integer a and b, ‘+’ in a+b means a plus of two integers
  - Avoid errors
    - Types limits the set of operations on these values. So, they avoid the accidental use of other operations on these types.

Type systems

- A type system consists of
  - A mechanism to define types and associate them with language constructs
    - Example: enumeration type

      int , string, …… (primitive type)

      enum colorType {red, blue, green};

      struct Object {int object; colorType color};

  - A set of rules for “type equivalence,” “type compatibility,” and “type inference.”

Type checking

- Type Checking is to check whether the type of an object fit into the type expected for this object in a context.
- The rules governing this checking are defined differently in different languages.
- The rules are usually based on *type equivalence* and *type compatibility.*

- There are two kinds of languages in term of type checking
  - Strongly typed.
  - Weakly typed.

Strongly typed

- Strongly typed languages always detect type errors
  - All expressions and objects must have a type
  - All operations must be applied in appropriate type contexts
- Statically typed languages are strongly typed languages in which all type checking occurs at compile time

Weakly typed

- In weakly typed languages “anything can go”
  - Characteristic of assembly language
  - Earlier scripting languages
- On the other end of the spectrum, strongly typed languages don’t allow *implicit* conversion

What is a type

- Three points of view
  - Denotational: A type is a set of values
  - Constructive: A type is “built-in” or “composite”
  - Abstraction-based: A type is a set of values and the operations on them

Denotational view

- A *type* is a set of values. A set of values is also called a *domain.* A type is a domain
- A value *has a type* implies it belongs to the corresponding domain.
- An object *has a type* implies its value must belong to the corresponding domain.
- Example:

```
enum colorType {red, blue, green};
struct Object {int object; colorType color};
```

Constructive view

- *Build-in* (primitive or elementary) types
  - Decision resulted from both math (application) and hardware.
- *Enumeration types*
- *Subrange types*
- *Composite* types
  - “combination of built-in types”

Built-in

- Character
  - Char
  - Unicode
- Numeric types
    - int: an integer but restricted to hardware.
    - float / boolean
    - character (application – display)(hardware) – one byte; now unicode – (to deal with characters in all natural languages)
- Most languages support integer and float
- Some support complex number/rational number/signed and unsigned number
- The precision of the real number leads to different types, e.g.,
    - float / double

Enumeration/subrange types

- Read the book P307-310
- Subrange

```
type water_temperature = 273 .. 373;
```

Constructive -- composite

- A *composite type* is created by applying *type constructors* to (simpler) types. Constructors are
  - *Records / structs*
  - *Arrays*
  - *Sets / Lists*
  - *Classes*

Orthogonality

- Orthogonality in expressions, statements and control-flow structures: different constructs can be combined in any legitimate combination with consistent behavior
- For types, orthogonality is equally important
  - Easier to use / to understand / to reason about their behaviors

- Example of type orthogonality
  – In C: one allows an array of object of arbitrary types
  – However, in Fortran, an array can contain only objects of non-composite types
- Allows *literal value* of any composite type
  – struct TypeEx {string name; int age} p;
  – p = (“Larry”, 37); // p = (age => 37, name=>”Larry”)
  – int abc[10];
  – abc = (0, 1, …, 9);

Type checking

- Once we have types for objects and expressions, next is to check if the use of these objects "consistent" with their types
  - e.g., int a; string b; a=b;
- There are three concepts
  - Type equivalent, two types are "same"
  - Type compatible, two types are compatible
    - Int x=10; float y = x;
  - Type inferences: infer the type of an expression from its subexpressions

Type equivalence

- Structural equivalence
  - Two types are *structurally equivalent* if (in constructive view) they have the same components put together in the same way
  - Example
    - typedef struct {int a; int b} type1;
    - typedef struct {int a; int b} type2;
    - typedef struct {int b; int a} type3;
    - type 1 and type2 are equivalent. But type1 and type3 are equivalent in some languages but not in others

– Examples

```c
typedef struct{
  char *name;
  char *addre;
  int age;
} student;
```

```c
typedef struct{
  char *name;
  char *addre;
  int age;
} school;
```

```c
student s1;
school s2;

s1 = s2; // intended?
```

Name equivalence

- Name alias

```
type celsius_temp = float;
type fahrenheit_temp = float;
celsius_temp t1; fahrentheit_temp t2;
t1 = t2; // could be a mistake
```

- Two types are *name equivalent* if they have the “same” name
  - *Strict name equivalence*: two types have the same name
  - *Loose name equivalence*: aliases of a type are considered the same as the type

- Yet another equivalence (Ada)
  - Replace alias by more detailed structure: subtypes and derived types
  - Subtypes are taken as the same while derived not
  - Example

```
subtype stack_element is integer
stack_element s1; int s2; s1 = s2 // ok
type celsius_temp is new integer
type fahrentheit_temp is new integer
celsius_temp t1; fahrenheit_temp t2;
t1 = t2 // not equivalent, compiler complains
```

Type conversion / cast

- Problem: suppose we use strict name equivalence. But sometimes we need to use a value across different types
  - int a=10; float b=a;

  We need type conversion / cast
- Example
  - int a=10; float b = (float) a;

- Converting type cast
  - int a=10; float b = (float) a;

    the representation of 10 is changed to float representation in b.
- Non-converting type cast
  - Representation of the value is not changed (mainly to pointer type)

- In C++
  - static cast (converting type cast)
    - double d=10.5
    - int n = static_cast <int> (d);
  - reinterpret cast (non-converting type cast)
    - struct {int name; string address} *p;
    - char *q
    - q = reinterpret_cast <char *> (p);
  - dynamic cast (on pointer types only)
    - http://www.cplusplus.com/doc/tutorial/typecasting/

Type compatibility

- Usually, type equivalence is not required by most languages. Instead,
- A value's type must be *compatible* with that of the context it appears.
- Type compatible is different in different languages

- For example, in Ada
  - Two types T and S are compatible in Ada if any of the following conditions are true:
    - T and S are equivalent
    - T is a subtype of S
    - S is a subtype of T
    - T and S are arrays with the same number elements and same type of elements

Type coercion

- Whenever a language allows a value of one type to be used in a context that expects another, the language must perform an automatic, implicit conversion to the expected type. This conversion is called *type coercion.*

```
short int s;
unsigned long int l;
...
s=l;
```

Overloading and coercion

- Example
  - a+b
  - + is usually either the addition of two integers or two real numbers.
  - In a language without coercion, the a, b must be both integer or both real numbers
  - In a language with coercion,
    - +: real addition if either a or b is real
    - +: integer addition if both a and b are integer

Universal reference type

- In system programming or in writing general-purpose container objects (stacks, sets …), *universal reference type* is introduced
  - e.g., in C
    - void *
  - in Java
    - Object

- Example: stack in java
  - Last in first out. The behavior matters. The type of the objects in the stack doesn't matter. So, we assume the stack elements have the generic reference type: Object.
  - The methods of `Stack` are declared as

```
Object peek();
Object pop();
Object push(Object);
```

– Any type is compatible with the generic type. But a type cast is needed to convert an Object type to a specific type

```
String s="hello world";
Stack myStack= new Stack();
myStack.push(s); // Type coercion
String y;
y = (String) myStack.pop(); //converting type cast needed
```

Type inference

- Simple expressions
  - A simple object (like variable, constant) is a simple *expression.*
  - If *e1, …, en* are expressions, f(e1, …, en) is a simple *expression* where f is either an operator (predefined) or a function (user defined)
- E.g., operators: `:=, +, ?,`

  `if … then … else …`

  user defined functions: …

- Relational expressions (Pascal)
  - A simple expression is a *relational expression*
  - If *e1* and *e2* are simple expression, then *e1 <rel-operator> e2* is a *relational expression*
- For a full example of expression see BNF at http://pascal.comsci.us/syntax/expression/index.html

- Problem: given the type of subexpressions, what’s the type of the overall expression?
  - The result of arithmetic operator has same type as the operands
  - The result of comparison is boolean
  - The result of a function call is return type of this function
  - The result of an assignment has the type of the left hand side variable

- In some cases (like subrange and composite type), it is not easy

```
type Atype = 0..20;
     Btype = 10..20;
var a: Atype;
    b: Btype;
...
a+b;
```

Type inference in ML

- What type information can be inferred?
  - i+1: since + is either on integer or real and1 is integer, i is of int
  - i=n: since i is int and = is on same types, n is of int
  - fib_helper 0 1 0: f1, f2, i: int since func arguments have to have the same type
  - if i=n then f2 else …: since f2 is int, the return type of fib_helper is int
  - Type of fib_helper: int * int * int -> int

```
fun fib n =
    let fun fib_helper f1 f2 i =
        if i = n
             then f2
            else fib_helper f2 (f1 + f2) (i + 1)
    in
        fib_helper 0 1 0
    end;
```

Type correctness

- A program is *type correct* if there is a unique (inferred or defined) type for every expression
  - e.g.,

```
let fun isquare x =
      x * x; (* Defaults to int -> int *)
in
      isquare(1.2) // type incorrect
```

instead

```
let fun rsquare x:real =
      x * x; (* real -> real *)
in
      isquare(1.2) // type correct
```

Polymorphism in ML

- Type inference from function `twice`
  - Assume x is of type ‘a, what’s the return type of f? the type of f?
  - Parametric polymorphism: the function `twice` can be applied to values of different types in different calls.

```
fun twice f x = f (f x);
      (* twice = fn : ('a -> 'a) 'a -> 'a *)
-twice (fn x => x / 2.0) 1.0;
  value: 0.25; type‘a: real
-twice (fn x => x ^ "ee") "whoop";
  value: "whoopeeee“; type ‘a: string
```

Type inference

Type inference using unification

- By function call we have
  - type(f1) = type(0), type(f2)=type(1), type(i)=type(j)
- By i + 1, type(i) = int
- Solving the equations (by unification), we have type(j) = int

```
……
    let fun fib_helper f1 f2 i =
        if i = n
             then f2
        else fib_helper f2 (f1 + f2) (i + 1)
    in
        fib_helper 0 1 j
……
```

- General constraints (inference) on types of expressions
  - Same identifier in the same scope have the same type.
  - if exp1 then exp2 else exp3:
    - type(exp1)=bool, type(exp2)=type(exp3)
  - The parameters in the func call have the same types as the parameters in the func def
    - e.g., for `let func f x = … in f y`
    - we have type(y) = type(x)

- The eqations put together are called *unification problems.*
- Unification algorithm is used to find the types of all expressions

Interesting types

- Records and variants (section 8.1)
  - It is a natural construct to model “objects” in an application
  - How it is implemented (memory layout)
- Arrays (section 8.2)
  - A natural construct to model a set of elements
  - How it is implemented (memory layout)
- Strings (8.3)
  - We have clear needs
  - How it is implemented
- Sets (8.4)
  - A fundamental math concept (and thus very useful in applications)
  - Implementation

- Pointers and recursive types (8.5)
- Lists (8.6)
- Files and Input/Output (8.7)
- Equality testing and Assignment (7.4)
  - Shallow (testing, assignment)
  - Deep (testing, assignment)

Summary

- What is a type?
- Type checking / inference
- Commonly used types and their implementation