Chapter 6 Types

A Type Is a Set

```
int n;
```

- When you declare that a variable has a certain type, you are saying that the values the variable can have are elements of a certain set

把语言进行抽象化

- *A type is a set of values*
  - plus a low-level representation
  - plus a collection of operations that can be applied to those values

Today: a Tour of Types

- There are too many to cover them all
- Instead, a short tour of the type menagerie
- Most ways you can construct a set in mathematics are also ways to construct a type in some programming language
- We will organize the tour around that connection

Primitive vs. Constructed Types

- Any type that a program can use but cannot define for itself is a *primitive type* in the language
- Any type that a program can define for itself (using the primitive types) is a *constructed type*
- Some primitive types in ML: **`int`**, **`real`**, **`char`**
  - An ML program cannot define a type named **`int`** that works like the predefined **`int`**
- A constructed type: **`int list`**
  - Defined using the primitive type **`int`** and the **`list`** type constructor

Primitive Types

- The definition of a language says what the primitive types are
- Some languages define the primitive types more strictly than others:
  - Some define the primitive types exactly (Java)
  - Others leave some wiggle room—the primitive types may be different sets in different implementations of the language (C, ML)

Comparing Integral Types

C:

`char`
`unsigned char`
`short int`
`unsigned short int`
`int`
`unsigned int`
`long int`
`unsigned long int`

No standard implementation, but longer sizes must provide at least as much range as shorter sizes.

Java:

`byte` (1-byte signed)
`char` (2-byte unsigned)
`short` (2-byte signed)
`int` (4-byte signed)
`long` (8-byte signed)

Scheme:

`integer`
Integers of unbounded range

Issues

- What sets do the primitive types signify?
  - How much is part of the language specification, how much left up to the implementation?
  - If necessary, how can a program find out? (**`INT_MAX`** in C, **`Int.maxInt`** in ML, etc.)
- What operations are supported?
  - Detailed definitions: rounding, exceptions, etc.
- The choice of representation is a critical part of these decisions

7

Constructed Types

- Additional types defined using the language
- Today: enumerations, tuples, arrays, strings, lists, unions, subtypes, and function types
- For each one, there is connection between how *sets* are defined mathematically, and how *types* are defined in programming languages

8

Making Sets by Enumeration

- Mathematically, we can construct sets by just listing all the elements:

$$S = \{a, b, c\}$$

9

Making Types by Enumeration

- Many languages support *enumerated types*:

C: `enum coin {penny, nickel, dime, quarter};`
Ada: `type GENDER is (MALE, FEMALE);`
Pascal: `type primaryColors = (red, green, blue);`
ML: `datatype day = M | Tu | W | Th | F | Sa | Su;`

- These define a new type (= set)
- They also define a collection of named constants of that type (= elements)

10

Representing Enumeration Values

- A common representation is to treat the values of an enumeration as small integers
- This may even be exposed to the programmer, as it is in C:

```
enum coin { penny = 1, nickel = 5, dime = 10, quarter = 25 };

enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
               NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };
```

11

Operations on Enumeration Values

- Equality test:

```
fun isWeekend x = (x = Sa orelse x = Su);
```

- If the integer nature of the representation is exposed, a language will allow some or all integer operations:

C: `int x = penny + nickel + dime;`

12

Making Sets by Tupling

- The Cartesian product of two or more sets defines sets of tuples:

$$S = X \times Y$$
$$= \{(x, y) \mid x \in X \wedge y \in Y\}$$

13

Making Types by Tupling

- Some languages support pure tuples:

```
fun get1 (x : real * real) = #1 x;
```

- Many others support record types, which are just tuples with named fields:

C:

```
struct complex {
  double rp;
  double ip;
};
```

ML:

```
type complex = {
  rp:real,
  ip:real
};
fun getip (x : complex) = #ip x;
```

14

Representing Tuple Values

- A common representation is to just place the elements side-by-side in memory
- But there are lots of details:
  - in what order?
  - with "holes" to align elements (e.g. on word boundaries) in memory?
  - is any or all of this visible to the programmer?

Sets of Vectors

- Fixed-size vectors:

$$S = X^n = \{(x_1, \ldots, x_n) \mid \forall i \,.\, x_i \in X\}$$

  - X^n is the set of all vectors of length n that have elements from X in every position.
- Arbitrary-size vectors:

$$S = X^* = \bigcup_i X^i$$

Types Related to Vectors

- Arrays, strings and lists
- Like tuples, but with many variations
- One example: indexes
  - What are the index values?
  - Is the array size fixed at compile time?

17

Index Values

- Java, C, C++:
  - First element of an array **`a`** is **`a[0]`**
  - Indexes are always integers starting from 0
- Pascal is more flexible:
  - Various index types are possible: integers, characters, enumerations, subranges
  - Starting index chosen by the programmer
  - Ending index too: size is fixed at compile time

18

Pascal Array Example

```
type
   array-identifier = array[index-type] of element-type;
```

indicates the name of the array type.

specifies the subscript of the array.

specifies the types of values that are going to be stored

```
type
   vector = array [ 1..25] of real;
var
   velocity: vector;
```

19

Pascal Array Example

```
type
   array-identifier = array[index-type] of element-type;
```

```
type
  LetterCount = array['a'..'z'] of Integer;
var
  Counts: LetterCount;
begin
  Counts['a'] = 1
```

etc.

20

Making Sets by Union 集合,(数据类型).

- We can make a new set by taking the union of existing sets:

$$S = X \bigcup Y$$

并集

21

Making Types by Union

- Many languages support union types:

C:

```
union element {
  int i;
  float f;
};
```

ML:

```
datatype element =
  I of int |
  F of real;
```

22

Representing Union Values

- You can have the two representations overlap each other in memory

```
union element {
  int i;
  char *p;
} u; /* sizeof(u) ==
         max(sizeof(u.i),sizeof(u.p)) */
```

- This representation may or may not be exposed to the programmer

23

Examples

```
#include <stdio.h>
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

Here, we can see that the values of **`i`** and **`f`** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **`str`** member is getting printed very well.

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

24

Strictly Typed Unions

- In ML, all you can do with a union is extract the contents
- And you have to say what to do with each type of value in the union:

```
datatype element =
  I of int |
  F of real;

fun getReal (F x) = x
  | getReal (I x) = real x;
```


Loosely Typed Unions

- Some languages expose the details of union implementation
- Programs can take advantage of the fact that the specific type of a value is lost:

```
union element {
  int i;
  float f;
};

union element e;
e.i = 100;
float x = e.f;
```

A Middle Way: Variant Records

- Union where specific type is linked to the value of a field ("***discriminated union***")
- The idea is to support unions as part of a record that also contains an enumeration. The value of the enumeration field determines the inner type of the union part.
- A variety of languages including Ada and Modula-2

27

Ada Variant Record Example

Creates an enumerated type **DEVICE**

Creates a record type **PERIPHERAL**

Value of **Unit** determines which of the union elements are present in the record.

```
type DEVICE is (PRINTER, DISK);

type PERIPHERAL(Unit: DEVICE) is
  record
    HoursWorking: INTEGER;
    case Unit is
      when PRINTER =>
        Line_count: INTEGER;
      when DISK =>
        Cylinder: INTEGER;
        Track: INTEGER;
    end case;
  end record;
```

PERIPHERAL contains a member called **Unit** of type **DEVICE**

28

Making Subsets

- We can define the subset selected by any predicate P:

$$S = \{x \in X \mid P(x)\}$$

29

Making Subtypes

- Some languages support subtypes, with more or less generality
  - Less general: Pascal subranges

```
type digit = 0..9;
```

  - More general: Ada subtypes

```
subtype DIGIT is INTEGER range 0..9;
subtype WEEKDAY is DAY range MON..FRI;
```

  - Most general: Lisp types with predicates

30

Example: Ada Subtypes

- From pervious example:

```
type DEVICE is (PRINTER, DISK);
type PERIPHERAL(Unit: DEVICE) is
  record
    HoursWorking: INTEGER;
    case Unit is
      when PRINTER =>
        Line_count: INTEGER;
      when DISK =>
        Cylinder: INTEGER;
        Track: INTEGER;
    end case;
  end record;
```

- A type that includes only those that are printers can be defined as follows:

```
subtype PRINTER_UNIT is PERIPHERAL(PRINTER);
```

31

Example: Lisp Types with Predicates

- Lisp supports a general kind of subtype, allowing any Lisp function to act as the predicate that selects members of the subtype.

```
(declare (type integer x))

(declare (type (and integer (satisfies evenp)) x))
```

- This declares the variable x to be an even integer. It says that the type of x is a subtype of integer, including those elements of the integer set for which the function evenp returns.

32

Representing Subtype Values

- Usually, we just use the same representation for the subtype as for the supertype
- Questions:
  - Do you try to shorten it if you can? Does `X: 1..9` take the same space as `X: Integer`?
  - Do you enforce the subtyping? Is `X := 10` legal? What about `X := X + 1`?

Operations on Subtype Values

- Usually, supports all the same operations that are supported on the supertype

```
function toDigit(X: Digit): Char;
```

A subtype is a subset of values, but it can support a superset of operations.

- Additional operations defined on the subtypes may not make sense on the supertype:

A Word about Classes

- This is a key idea of object-oriented programming
- In class-based object-oriented languages, a *class* can be a type: data and operations on that data, bundled together
- A *subclass* is a subtype: it includes a subset of the objects, but supports a superset of the operations
- More about this in Chapter 13

35

Making Sets of Functions

- We can define the set of functions with a given domain and range:

定义域 值域
doman → range

$$S = D \to R$$
$$= \{ f \mid \text{dom } f = D \wedge \text{ran } f = R \}$$

36

Making Types of Functions

- Most languages have a similar idea of function types, which specify the function's domain and range.

C:
```
int f(char a, char b) {
  return a==b;
}
```

ML:
```
fun f(a:char, b:char) = (a = b);
    ( char * char -> bool )
```


Operations on Function Values

- Of course, we need to *call* functions
- We have taken it for granted that other types of values could be passed as parameters, bound to variables, and so on
- Can't take that for granted with function values: many languages support nothing beyond function call
- We will see more operations in ML

Questions?

39