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

■ *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

Outline

- **Type Menagerie**
  - Primitive types
  - Constructed types
- **Uses For Types**
  - Type annotations and type inference
  - Type checking
  - Type equivalence issues

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

Outline

■ Type Menagerie

   – Primitive types

   – **Constructed types**

■ Uses For Types

   – Type annotations and type inference

   – Type checking

   – Type equivalence issues

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

Making Sets by Enumeration

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

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

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)

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' };
```

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:

```
Pascal:                    for C := red to blue do
P(C)

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

Making Sets by Tupling

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

$$S = X \times Y$$

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

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;
```

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?

Example: ANSI C

The members of a structure have addresses increasing in the order of their declarations. A non-field member of a structure is aligned at an addressing boundary depending on its type; therefore, there may be unnamed holes in a structure. If a pointer to a structure is cast to the type of a pointer to its first member, the result refers to the first member…

Adjacent field members of structures are packed into implementation-dependent storage units in an implementation-dependent direction...

The C Programming Language, 2nd ed.
Brian W. Kernighan and Dennis M. Ritchie

Operations on Tuple Values

■ Selection, of course:

        C: `x.ip`

        ML: `#ip x`

■ Other operations depending on how much of the representation is exposed:

```
C:      double y = *((double *) &x);
        struct person {
          char *firstname;
          char *lastname;
        } p1 = {"marcia","brady"};
```

Sets Of Vectors

■ Fixed-size vectors:

$$S = X^n$$

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

■ 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?

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

Pascal Array Example

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

begin
   Counts['a'] = 1
   etc.
```

Types Related To Vectors

■ Many variations on vector-related types:

What are the index values?

Is array size fixed at compile time (part of static type)?

What operations are supported?

Is redimensioning possible at runtime?

Are multiple dimensions allowed?

Is a higher-dimensional array the same as an array of arrays?

What is the order of elements in memory?

Is there a separate type for strings (not just array of characters)?

Is there a separate type for lists?

Making Sets by Union

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

$$S = X \bigcup Y$$

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;
```

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

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;
```

What ANSI C Says About This

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time. If a pointer to a union is cast to the type of a pointer to a member, the result refers to that member.

In general, a member of a union may not be inspected unless the value of the union as been assigned using that same member.

The C Programming Language, 2nd ed.
Brian W. Kernighan and Dennis M. Ritchie

A Middle Way: Variant Records

- Union where specific type is linked to the value of a field ("*discriminated union*")

- A variety of languages including Ada and Modula-2

Ada Variant Record 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;
```

Making Subsets

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

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

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

Example: Ada Subtypes

```
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;

subtype DISK_UNIT is PERIPHERAL(DISK);
```

Example: Lisp Types with Predicates

```
(declare (type integer x))

(declare (type (or null cons) x))

(declare (type (and number (not integer)) x))

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

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
- And perhaps additional operations that would not make sense on the supertype:

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

- Important meditation:

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

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

Making Sets of Functions

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

$$S = D \rightarrow R$$
$$= \{f \mid \operatorname{dom} f = D \land \operatorname{ran} f = R\}$$

Making Types of Functions

■ Most languages have some notion of the type of a function:

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

ML: `fun f(a:char, b:char) = (a = b);`

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

Outline

■ Type Menagerie
  – Primitive types
  – Constructed types

■ Uses For Types
  – **Type annotations and type inference**
  – Type checking
  – Type equivalence issues

Type Annotations

- Many languages require, or at least allow, type annotations on variables, functions, …

- The programmer uses them to supply static type information to the language system

- They are also a form of documentation, and make programs easier for people to read

- Part of the language is syntax for describing types (think of `*`, `->` and `list` in ML)

Intrinsic Types

■ Some languages use naming conventions to declare the types of variables

- Dialects of BASIC: **S$** is a string

- Dialects of Fortran: **I** is an integer

■ Like explicit annotations, these supply static type information to the language system and the human reader

Extreme Type Inference

- ML takes type inference to extremes

- Infers a static type for every expression and for every function

- Usually requires no annotations

Simple Type Inference

- Most languages require some simple kinds of type inference

- Constants usually have static types
  - Java: `10` has type `int`, `10L` has type `long`

- Expressions may have static types, inferred from operators and types of operands
  - Java: if `a` is `double`, `a*0` is `double` (`0.0`)

Outline

- Type Menagerie
  - Primitive types
  - Constructed types
- Uses For Types
  - Type annotations and type inference
  - **Type checking**
  - Type equivalence issues

Static Type Checking

- Static type checking determines a type for everything before running the program: variables, functions, expressions, everything

- Compile-time error messages when static types are not consistent
  - Operators: `1+"abc"`
  - Functions: `round("abc")`
  - Statements: `if "abc" then` ...

- Most modern languages are statically typed

Dynamic Typing

- In some languages, programs are not statically type-checked before being run

- They are usually still *dynamically* type-checked

- At runtime, the language system checks that operands are of suitable types for operators

Example: Lisp

■ This Lisp function adds two numbers:

```
(defun f (a b) (+ a b))
```

■ It won't work if **a** or **b** is not a number

■ An improper call, like **(f nil nil)**, is not caught at compile time

■ It is caught at runtime – that is dynamic typing

It Still Uses Types

- Although dynamic typing does not type everything at compile time, it still uses types

- In a way, it uses them even more than static typing

- It needs to have types to check at runtime

- So the language system must store type information with values in memory

Static And Dynamic Typing

■ Not quite a black-and-white picture

■ Statically typed languages often use some dynamic typing

- – Subtypes can cause this

- – Everything is typed at compile time, but compile-time type may have subtypes

- – At runtime, it may be necessary to check a value's membership in a subtype

- – This problem arises in object-oriented languages especially – more in Chapter 13

Static And Dynamic Typing

■ **Dynamically typed languages often use some static typing**

  – Static types can be inferred for parts of Lisp programs, using constant types and declarations

  – Lisp compilers can use static type information to generate better code, eliminating runtime type checks

Explicit Runtime Type Tests

■ Some languages allow explicit runtime type tests:

- – Java: test object type with **`instanceof`** operator
- – Modula-3: branch on object type with **`typecase`** statement

■ These require type information to be present at runtime, even when the language is mostly statically typed

Strong Typing, Weak Typing

■ The purpose of type-checking is to prevent the application of operations to incorrect types of operands

■ In some languages, like ML and Java, the type-checking is thorough enough to guarantee this—that's *strong typing*

■ Many languages (like C) fall short of this: there are holes in the type system that add flexibility but weaken the guarantee

Outline

■ Type Menagerie
  – Primitive types
  – Constructed types

■ Uses For Types
  – Type declarations and inference
  – Static and dynamic typing
  – Type equivalence issues

Type Equivalence

- When are two types the same?

- An important question for static and dynamic type checking

- For instance, a language might permit `a:=b` if `b` has "the same" type as `a`

- Different languages decide type equivalence in different ways

Type Equivalence

- *Name equivalence*: types are the same if and only if they have the same name

- *Structural equivalence*: types are the same if and only if they are built from the same primitive types using the same type constructors in the same order

- Not the only two ways to decide equivalence, just the two easiest to explain

- Languages often use odd variations or combinations

Type Equivalence Example

```
type irpair1 = int * real;
type irpair2 = int * real;
fun f(x:irpair1) = #1 x;
```

■ What happens if you try to pass **f** a parameter of type **irpair2**?

   – Name equivalence does not permit this: **irpair2** and **irpair1** are different names

   – Structural equivalence does permit this, since the types are constructed identically

■ ML does permit it

Type Equivalence Example

```
var
   Counts1: array['a'..'z'] of Integer;
   Counts2: array['a'..'z'] of Integer;
```

■ What happens if you try to assign **Counts1** to **Counts2**?

- Name equivalence does not permit this: the types of **Counts1** and **Counts2** are unnamed
- Structural equivalence does permit this, since the types are constructed identically

■ Most Pascal systems do not permit it

Conclusion

- A key question for type systems: how much of the representation is exposed?

- Some programmers prefer languages like C that expose many implementation details
  - They offer the power to cut through type abstractions, when it is useful or efficient or fun to do so

- Others prefer languages like ML that hide all implementation details (*abstract types*)
  - Clean, mathematical interfaces make it easier to write correct programs, and to prove them correct