Types

A Type Is A Set

```
int n;
```

- Declaring a variable as a certain type restricts values to elements of a certain set
- *Type* has a variety of definitions
- Definition: *type* is the set of values
  - plus a low-level representation
  - plus a collection of operations that can be applied to those values

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
- Tour organized 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

- *primitive type* - Any type that a program can use but cannot define for itself in the language
- *constructed type* - Any type that a programmer can define (using the primitive types)
- F# inherits its primitive types from .NET and has same types and precision, e.g. **`int`**, **`double`**, **`char`**
  - F# program cannot define a type named **`int`** that works like the predefined **`int`**
- A constructed type (F#): **`int list`**
  - Defined using the primitive type **`int`** and the **`list`** type constructor

Primitive Types

- The definition of primitive types part of language
- 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)

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

Class versus Type

The following definition is attributed to Gof book (Design Patterns) (http://stackoverflow.com/questions/468145/what-is-the-difference-between-type-and-class)

```
An object's class defines how the object is
implemented. The class defines an object's internal
state and the implementation of its operations.

In contrast, an object's type only refers to its
interface - the set of requests to which it can
respond.

An object can have many types, and objects of
different classes can have the same type.
```

Consider a *type* Stack with interface operations:

`push, pop, isEmpty`

One *class* for `String`, another *class* for `Int`

Example – Size Matters

F#

```
let rec fact n : bigint =
  match n with
    | 0 -> bigint 1
    | n -> fact (n-1) * (bigint
    n);;
fact 100L
```

bigint =

93326215443944152681699
23885626670049071596826
43816214685929638952175
99993229915608941463976
15651828625369792082722
37582511852109168640000
00000000000000000000

C++

```
int fact(int n) {
    switch (n) {
      case 0 : return 1;
      default: return fact(n-1) * n;
    }
}

void main(void) {
    cout << fact(100);
}
```

0

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.MaxValue`** in F#, **`BigInt.MaxValue`** is not defined.)
- What operations are supported?
  - Detailed definitions: rounding, exceptions, etc.
- The choice of representation (e.g. bit size) 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 based on language
- 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*:

  Scala: No

- F#: YES! `type day = M | Tu | W | Th | F | Sa | Su;;`

  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 (F#):

```
type day = M | Tu | W | Th | F | Sa | Su;;
let isWeekend x = (x = Sa || x = Su);;
isWeekend W;;      returns false
Tu;;               returns val it : day = Tu
```

- 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;` is 16

Defining Sets by Tupling

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

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

- Example:

A = {light, dark}

B = {red, blue, green}

A x B = { (light,red), (light,blue), (light, green),
(dark,red), (dark,blue), (dark, green) }

Exercise 0.1

What is the Cartesian product defined by A x B for the enumerated sets:

A = {winter, summer }
B = {hot, cold};

Aggregate and Scalar types

- Aggregate – type composed of more than one value; tuples, records, arrays, sets, etc.
- Scalar – type composed of a single value; integers, reals, enumeration, etc.

Defining Types by Tupling

- Some languages support pure tuples (F#):

```
let get1 (x : double * double) = fst x;;
```

- Others, record types or tuples with named fields:

C:

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

float getip(complex x){
       return x.ip;    }

complex y = {1.0, 2.0};

double i = getip(y); i?
```

F#:

```
type Complex = Complex of double *
       double;;

let y = Complex (1.0, 2.0);;

let getip (Complex (rp, ip)) = ip;;

let i = getip y;;
```

Exercise 0.2

```
type Complex = Complex of double
* double;;

let y = Complex (1.0, 2.0);;

let getip (Complex (rp, ip)) = ip;;

let i = getip y;;
```

1. Define a F# class *person* with *name, age, height* and *weight* attributes.
2. Assign person variable *President* values of "George", 57, 70, 175.
3. Give a function *name* that returns the name value of a person.
4. Use function *name* to return the name of the *President*.

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

- Construction:

C:
```
complex x = {1.0, 2.0};
```
F#:
```
let x = (1.0, 2.0);
```

- Selection:

C:
```
x.ip
```
F#:
```
fst x
```

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

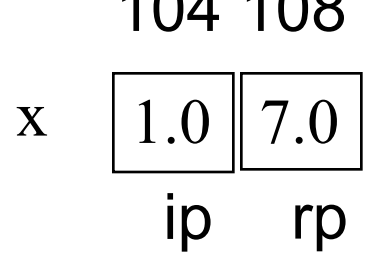
C:
```
double y = *((double *) &x);
struct person {
  char *firstname;
  char *lastname;
} p1 = {"marcia","brady"};
```

Exercise 1

```
struct complex {
    double ip;
    double rp;
} x = {1.0, 2.0};

void main(void) {
    double *px = &x.ip;
    px++;
    cout << *px;
}
```

Assume doubles are 4 bytes



1. What is the value of **px** after: **`double *px = &x.ip;`**
2. After px++
3. What is the output?

Exercise 1 Continued

4. Define the C enum for Automobile of **Ford, Honda, Yugo.**
5. What are the **Ford, Honda, Yugo** values in C?
6. What is the Cartesian product defined by A x B :

A = {Ford, Honda, Yugo }
B = {red, blue, green}

Sets Of Vectors

- Fixed-size vectors: $S = X^n$

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

Example: $X = \{3, 5\}$

$X^3 = X \times X \times X = \{(3,3,3), (3,3,5), (3,5,3), (3,5,5), (5,3,3), (5,3,5), (5,5,3), (5,5,5)\}$

- Arbitrary-size vectors: $S = X^*$

$$= \bigcup_i X^i$$

Types Relate 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++, C#:
  - First element of an array **`a`** is **`a[0]`**
  - Indexes are always integers starting from 0
- F#:
  - 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

F# Array Example

```
let counts = Array.create 26 0;;

// Also could use: let counts := [|0;0;...;0|];;

let ch = 'd';;

Array.set counts ((int ch)-(int 'a')) 1;;

counts.[3];;
      val it : int = 1
```

etc.

Pascal Array Example

```
type
  LetterCount = array['a'..'z'] of Integer;
  TempCount = array[-100..100] of Integer;
var
  Counts: LetterCount;

begin
  Counts['a'] = 1;
  TempCount[-12] = TempCount[-12] + 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?

Exercise 2: Dynamic C++ arrays valid?

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

void main(void) {
      int n;
      cin >> n;
      complex *x = new complex[n];

      for (int i=0; i<n; i++)
            x[ i ].ip = 1.0;
      cout << x[n-1].ip;
}
```

Exercise 2: Continued

a) The following is invalid C. Why?
b) How many variables are defined of type complex?
c) How many variables are defined of type double?

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

void main(void) {
        complex x[5][6];
        x[3][4].rp = 2.0;
        x[3,4].ip = 1.0;
}
```

Making Sets by Union

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

$$S = X \bigcup Y$$

- Example: A = {a, b, c}
  B = {b, c, d}
  A ∪ B = {a, b, c, d}

Making Types by Union

- Many languages support **union** types:

C:

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

F#:

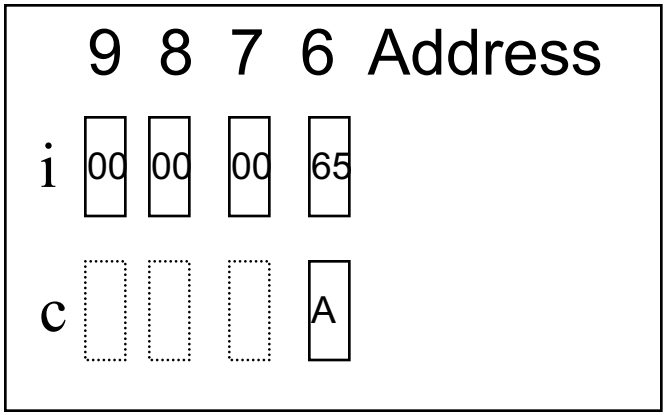
```
type element =
   I of int
   | F of double;;
```

- C - One value accessible as different types.

Representing Union Values in C

- The two representations can overlap each other in memory

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



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

```
element x;
x.i = 65;
cout << x.c;
```

'A'

Type unions – F#

- Define abstract base class and case classes
- Extract the contents by defining operations that return only **one** type of value:

```
type Element =
 | I of int
 | F of double

let getReal e =
 match e with
  | I i -> double i
  | F f -> f
```

```
let y = (F 3.0)
let z = getReal y
y is F(3.0), z is 3.0
```

```
let a = (I 4)
let b = getReal a
a is I(4), b is 4.0
```

Example – case class

```
type Element =
 | I of int
 | F of double

let double e =
 match e with
   | I y -> I(y*2)
   | F x -> F(x*2.0);;

double (I 4);
     val it : Element = I 8
double (F 4.0);
     val it : Element = F 8.0
```

What is `double` range? - domain?

What is `double (double (F (4.0)))`?

Is strict type checking still in force?

Example – case class – Lists

Is `List(I(5), F(5.0))` valid?

Is `map(double, List(F(4.3), I(5), F(3.0)))` valid?

```
type Element =
 | I of int
 | F of double

let double e =
 match e with
  | I y -> I(y*2)
  | F x -> F(x*2.0);;

let rec map f L =
 match L with
  | [] -> []
  | h::t -> f h::map f t;;

map double [F 4.3; I 5; F 3.0];;
    val it : Element list = [F 8.6; I 10; F 6.0]
```

Exercise 2.9 – case class

```
type Element =
 | I of int
 | F of double

let add e1 e2 =
 match (e1, e2) with
   | (I x, I y) -> I(x+y)
   | (F x, F y) -> F(x+y)
   | (F x, I y) -> F(x+(double y))
   | (I x, F y) -> F((double x)+y);;
```

1. Define function *gt* that returns true when the first element is greater than the second. `gt I(5) F(5.3)` returns `false`

2. Define function *max* to return the maximum of two *element*s using function *gt*.

Exercise 2.9 – case class – Lists

```
type Element =
 | I of int
 | F of double

let add e1 e2 =
 match (e1, e2) with
  | (I x, I y) -> I(x+y)
  | (F x, F y) -> F(x+y)
  | (F x, I y) -> F(x+(double y))
  | (I x, F y) -> F((double x)+y);;

let rec reduce f a L =
 match L with
  | [] -> a
  | h::t -> f h (reduce f a t);;
```

3. *reduce* and *add* sums the list of elements: [F 4.3; I 4; F 2.7]

4. *reduce* and *max* returns the maximum of list: [F 4.3; I 4; F 2.7]

Loosely Typed Unions

- Some languages expose union implementation details
- 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;          int
float x = e.f;      float
cout << x;
```

On 32-bit Intel processor output is:

1.4013e-043

Exercise 3

```
union element {
   int i;
   char c;
};

void main(void) {
   union element e;
   e.i = -12345;
   e.c = 'A';
   int i = e.i;
   cout << i;
}
```

1. Is the output of the program:
   a) 65 — integer value of e.c, ‘A’
   b) -12345 — integer value of e.i
   c) -12479 — integer value of e.i after ‘A’ replaced the low byte
2. What makes the difference?

Exercise 3

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

void main(void) {
   element x, y, z;
   y.i = 3;
   z.f = 5.4;
   x = add(y,z);
}
```

3. Write the C function **add** of two **`element`** types and that returns the sum as an **`element`**.
4. What is the value of x at the end of execution?

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;
```

```
DriveC : PERIPHERAL(DISK);
DriveC.Line_count = 5000;   Invalid - No Line_count
DriveC.Cylinder = 3;        Valid - Cylinder
```

Making Subsets

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

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

- S is the set of elements x from X where P(x) is true. P is a filter function.

Exercise 3.1

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

```
let predicate1 x = x%2=1;;
let predicate2 x = x<3;;

let rec filter f L =
 match L with
   | [] -> []
   | h::t when f h -> h::filter f t
   | h::t -> filter f t;;
```

- What is returned in the following?

```
filter predicate1 [1;2;3;4;5];;
filter predicate2 [1;2;3;4;5];;
```

Making Subtypes

- Some languages support subtypes, with more or less generality

  - Less general: Pascal subranges

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

  - More general: Ada subtypes

```
type DAY is (MON,TUE,WED,THU,FRI,SAT,SUN);
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);
```

A **DISK_UNIT** is a record of: **HoursWorking, Cylinder, Track**

Example: Lisp Types with Predicates

```
(declare (type integer x))
```

x is an integer

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

x is a number and not an integer

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

x is an integer and is even

Representing Subtype Values

- Usually, we just use the same representation for the subtype as for the supertype
- Questions:
  - Can the subtype occupy less memory? Does `X: 1..9` take the same number of bits 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 support a superset of operations.

Example:

Subtype: hours = 1 .. 12

Operations: Predicates: AM, PM, Afternoon

Special arithmetic: 9am+5=2pm

A Word About Classes

- Classes - 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 specialization of a superset
- 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

The set of functions S that map values from domain D into the range R:

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

Making Types of Functions

- Most languages have some notion of the type of a function:
- What is the **domain** and **range** of the following?

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

F#:
```
let f (a : char) (b : char) : bool = a = b;;
val f : a:char -> b:char -> bool
```

Exercise 3.2

- Give the *domain, range* and *signature* of the following:

1.
```
let f1 x = x > -1;;
```

2.
```
let f2 x =
 match x with
   | 0 -> true
   | z -> z < 1;;
```

3.
```
let f3 (x : 'a) =
 match box x with
   | :? int as i when i = 0 -> true
   | :? double as z -> z < -1.0;;
```

4.
```
let rec f4 (L : int list) =
 match L with
   | [] -> 0
   | h::t -> h + f4 t;;
```

Operations on Function Values

- *Call* functions
- Take for granted that other types of values could be passed as parameters, bound to variables, and so on
- Not with function values: many languages support nothing beyond function call
- We will see more operations in F#

Outline

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

Type Annotations – (e.g. float x)

- Many languages require (Java), or at least allow (F#), type annotations on variables, functions, …
- The programmer uses them to supply static type information to the language system
- Also a form of documentation, and make programs easier for people to read
- Part of the language is syntax for describing types (think of **`tuples`**, **`->`** and **`list`** in F#)

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

Type Inference

- F# makes type inference
- Attempts to infer a static type for every expression and for every function
- Usually requires no programmer annotations
- What is the *domain* and *range* signature of the following?

```
3+4;;

3.0+4.0;;

let f x y = x + y;;
```

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 execution (during compile) variables, functions, expressions, everything
- Compile-time error messages when static types are not consistent
  - Operators: **`1/"abc"`**
  - Functions: **`round("abc")`**
  - Statements: **`if "abc" then`** …
- Many modern languages (e.g. F#) are statically typed

Dynamic Typing

- In some languages, programs are not statically type-checked before being run
- Still *dynamically* type-checked usually
- At runtime, the language system checks that operands are of suitable types for operators

Python example:
```
def add(x,y):
    return x+y;
```

add(3, 4.2) is 7.2

add(“3”, “4.2”) is 34.2

Exercise 4 – Match the JavaScript

a) Nan
b) 7
c) 12
d) Hello4
e) HelloThere

```
function plus(x, y) {   return x + y;    }

function times(x, y) { return x * y;    }
```

1. times("Hello", 4)	5. times("3", 4)
2. plus(3, 4)	6. plus("Hello", "There")
3. times("3", "4")	
4. plus("Hello", 4)	7. times(3, 4)

Exercise 4 – Match the JavaScript

1. NaN
2. 7
3. 12
4. Hello4
5. 12
6. HelloThere
7. 12

```
function plus(x, y) {    return x + y;    }

function times(x, y) { return x * y;    }
```

1. times("Hello", 4)	5. times("3", 4)
2. plus(3, 4)	6. plus("Hello", "There")
3. times("3", "4")	
4. plus("Hello", 4)	7. times(3, 4)

Exercise 4 – Match the C++

a) Compile error

b) prints 2

c) prints 1

d) prints b

char c = '1'; c = c + '1'; cout << c;	char c = '1'; c = c + 1; cout << c;
char c = '1'; c = c + "1" ; cout << c;	char c = '1'; c = c + 1.6 ; cout << c;

Exercise 4 – Match the C++

a) Compile error

b) prints 2

c) prints 1

d) prints b

d)	char c = '1'; c = c + '1'; cout << c;	b)	char c = '1'; c = c + 1; cout << c;
a)	char c = '1'; c = c + "1" ; cout << c;	b)	char c = '1'; c = c + 1.6 ; cout << c;

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

Dynamic Typing 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

Example: “abc” + “xyz”

Valid only if operation + defined on strings.

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 class 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 F# and C#, the type-checking is thorough enough to guarantee this—that's *strong typing*
- Many languages (e.g. C and VB 6.0) fall short of this: there are holes in the type system that add flexibility but weaken the guarantee

Strong Typing, Weak Typing

- F# is strongly typed but performs type inference
- Does F# enforce strong type checking on variables in the following?

```
let add x y = x + y;;
```

```
let add x y = x::y;;
```

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 assignment **`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

F# Type Equivalence Example

```
type irpair1 = int * double;;
type irpair2 = int * double;;
let (y:irpair1) = (1,2.0);;
let f (x:irpair2) = fst 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
- F# does permit it based on structure

Type Equivalence Example

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

- What happens if you try to assign **`Counts1`** := **`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

Exercise 5

The following C code fails to compile. Does C use *name* or *structural* equivalence?

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

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

```
void main(void) {
    complexA xA = { 1.0, 2.0 };
    complexB xB;
    xB = xA;                Error
}
```

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