Semantics

Aditya Thakur

Scope in Go

- https://play.golang.org/p/TW23ukHm6ew
- https://play.golang.org/p/2yCBYxxBlk_k
- https://play.golang.org/p/Uigll4QhfNd
- https://play.golang.org/p/MMTqTqnv0Mz

Types

A *type* is a set of values, together with a collection of operations on those values

```
int n;
```

Types (cont'd)

- Simple types/basic types/primitive types
  - From which all other types are constructed
  - Specified either using keywords or predefined identifiers
- Constructed types
  - Any type that a program can define for itself using the primitive types

Type Constructors

A type constructor is a standard set operation applied to a set of types to construct a new type

- Examples:

Cartesian product type constructor
int X char X double

```
struct IntCharReal {
int i;
char c;
double r;
};
struct IntCharReal x;
x.i = 42;
```

Union type constructor

```
union IntOrReal {
 int i;
 double r;
};
union IntOrReal u;
int num =  u.i;
```

Types in C

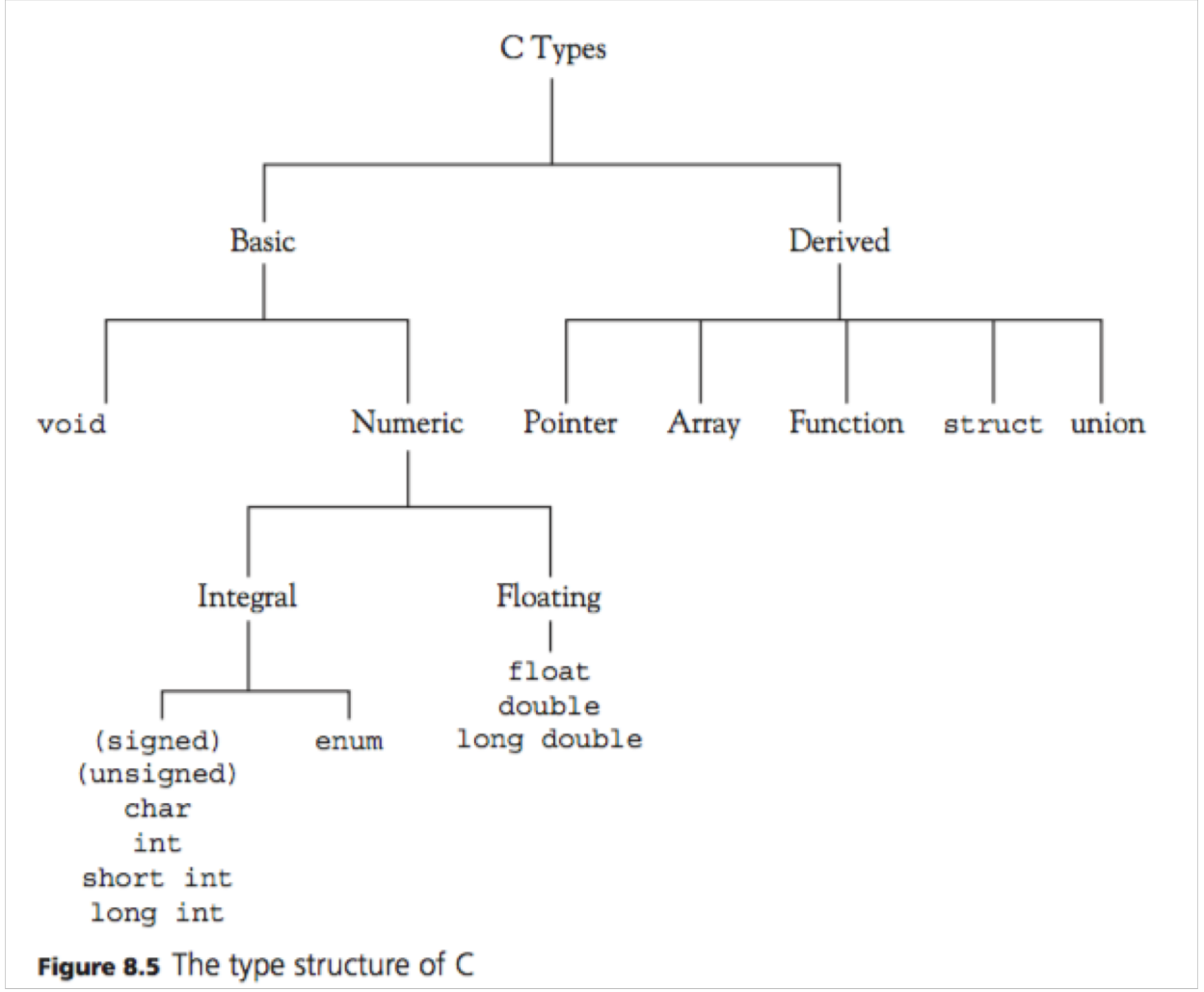

Figure 8.5 The type structure of C

Types in Java

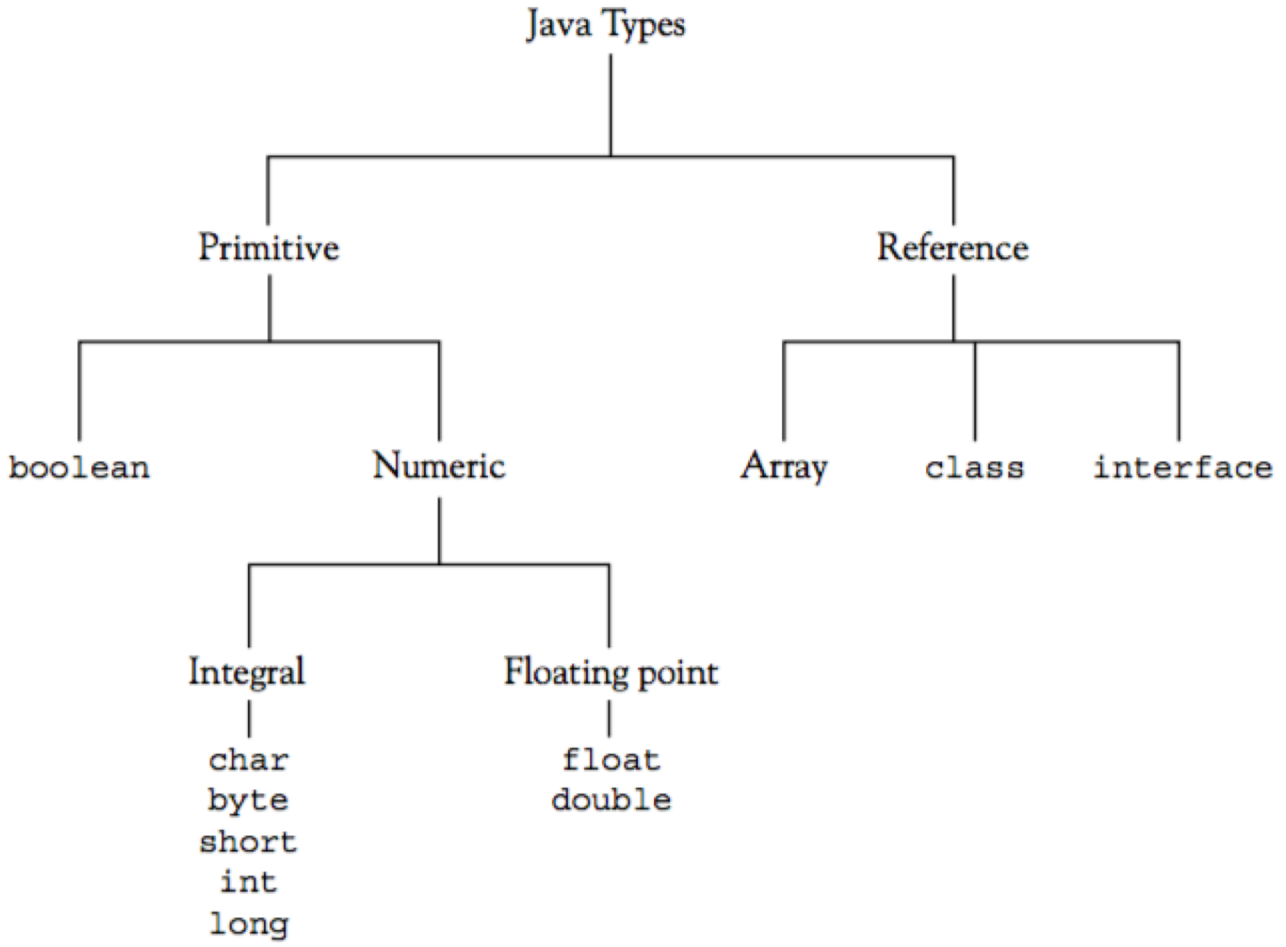

Figure 8.6 The type structure of Java

Types in Go

- Basic types
  - https://tour.golang.org/basics/11
- Struct types
  - https://tour.golang.org/moretypes/2

Benefits of Type Information

- Program organization and documentation
  - Separate types for separate concepts
  - Represent domain-specific concepts
  - Indicate intended use of declared identifiers
    - Types can be checked, unlike program comments
- Detect and prevent errors
  - 3 + true - “Bill”
- Support Optimization
  - Example: Short integers require fewer bits

Type Equivalence

When are two types the same?

- Structural equivalence
  - Types are the same if they have the same structure

```
typedef struct {int a; char b;} X, Y;
typedef struct {char b; int a;} Z;
typedef struct {int first; char second;} W;
```

Are types X, Y, Z, and W equivalent?

- Name equivalence
  - https://play.golang.org/p/GDmL5VqNUce

Type Checking

- Compile time or static
  - C, C++, Java, Go, Haskell, ML, etc.
  - Example: C/Java: f(x)
    - f : A -> B and x: A
- Run time or dynamic
  - Lisp, Scheme, etc.
  - Example: Lisp: (car x)
    - Makes sure x is list a before taking car
- Both static and dynamic prevent type errors
- There are tradeoffs
  - Dynamic: slower
  - Static: restrict program flexibility

Type Annotations vs.Type Inference

- Many languages require, or at least allow type annotations
  - Programmer supplies explicit type information to the language system
  - Examples:
    - Java requires a definition for each variable
    - ML allows to attach type annotations to variables, functions and expressions
- Most languages systems also collect information from other sources
  - Constants have types
  - Expressions have types (depending on its operands)
  - Types can be inferred!

Type Checking and Type Inference

- Standard type checking

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

  - Look at body of each function and use declared types of identifiers to check agreement

- Type inference

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

  - Look at code without type information and figure out what types could have been declared

Basic Types in Go

- bool
- string
- int, int8, in16, int32, int64
- uint, uint8, uint16, uint32, uint64, uintptr
- byte (alias for uint8)
- rune (alias for int32, represents a unicode)
- float32, float64
- complex64, complex128

https://tour.golang.org/basics/11

More Types

- Pointers https://play.golang.org/p/tgSutr3fGmr
- Arrays and slices https://tour.golang.org/moretypes/6
- Maps https://play.golang.org/p/U1B_gxkv7vL

User-defined Types

- Based on basic types
  - https://play.golang.org/p/P8kE3sJWx1r
- Structs
  - https://play.golang.org/p/3wOX38kj-FT
  - Structs inside structs https://play.golang.org/p/5wv9utsqXZM
- All types can have operations associated with them

Type errors

- Types help catch common mistakes
- https://play.golang.org/p/cJDAvlwkVFf
- https://play.golang.org/p/K0OKFsiFm_r

Type Conversion

- https://tour.golang.org/basics/13
- https://play.golang.org/p/bMIbGkzZhLF

Interfaces

- An interface is an abstract type: it only defines a set of methods; nothing about the internal representation.
- A type *satisfies* an interface if it possesses all the methods the interface requires.
- The ability to replace one type for another that satisfies the same interface is called *substitutability.*
- https://tour.golang.org/methods/9
- https://play.golang.org/p/iEmPmAX5FMN
- https://play.golang.org/p/J-0zOGsKzxT

Type assertions

- Provides access to an interface value's underlying concrete value.
- https://play.golang.org/p/CWPHr0VcZvz
- Similar to `dynamic_cast` in C++

Empty interface

- The interface{} type is the interface that has no methods.
- *All types satisfy the empty interface*.
- https://play.golang.org/p/LbJvtn-NdWl

Empty interface and Type assertions

- https://tour.golang.org/methods/15

Empty interface and type switches

- https://play.golang.org/p/sDOlRsHcaJF
- https://play.golang.org/p/xHT-WXZ-Kei
- Also see `expr/print.go` in hw2 for an example of type switches

Function values

- Functions are first-class values in Go
  - Function values have types, may be assigned to variables, passed to and returned from functions
- A function value may be called like any other function
- https://play.golang.org/p/bC1VTNdTxai

Closures

- A closure is a function value that references variables from outside its body.
- Note that the lifetime of a variable is not determined by its scope
- https://play.golang.org/p/N_mHpPrS0wr
- https://play.golang.org/p/odkhcRsAlif
- https://play.golang.org/p/jcGg3dndii2
- https://play.golang.org/p/FO9J0EB5nm4
  - https://play.golang.org/p/a4pSFX0QzNe

Type Embedding

- Take existing types and both extend and change their behavior
- https://play.golang.org/p/U3tBEcWmDyU
- Compare the above code to https://play.golang.org/p/5wv9utsqXZM

Exporting and unexporting identifiers

- Package-level visibility of identifiers
- Identifier is unexported or unknown to code outside the package iff the identifier starts with a lowercase letter.
- Note that identifiers are (un)exported, not values.

Object-Oriented Programming

- Encapsulation and information hiding
  - Package-level exports in Go
  - Public and private keywords in C++
- Composition and inheritance
  - “has-a” vs “is-a”
  - Type embedding in Go vs inheritance in C++
- Polymorphism
  - Interfaces in Go similar to abstract base classes in C++
  - Templates in C++
  - Operator and function overloading in C++