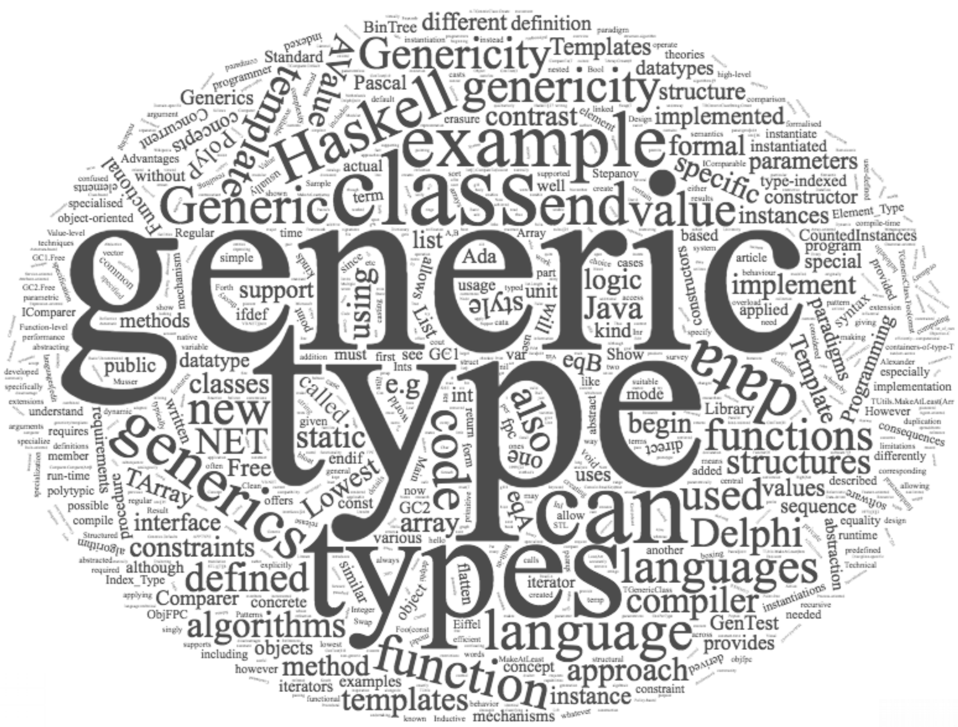
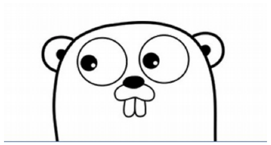
generic programming





alberto ferrari – university of parma

three approaches to generics

- **C approach** -> *leave them out*
  - ***slows programmers***
  - adds no complexity to the language
- **C++ approach** -> *compile-time specialization or macro expansion*
  - ***slows compilation***
  - generates a lot of code, much of it redundant, and needs a good linker to eliminate duplicate copies
  - the individual specializations may be efficient but the program as a whole can suffer due to poor use of the instruction cache. “I have heard of simple libraries having text segments shrink from megabytes to tens of kilobytes by revising or eliminating the use of templates”
- **Java approach** -> *box everything implicitly*
  - ***slows execution***
  - compared to the implementations the C programmer would have written or the C++ compiler would have generated, the ***Java code is smaller but less efficient*** in both time and space, because of all the implicit boxing and unboxing

generics vs templates

- syntax
  - the syntax is deliberately similar
  - <angle brackets>
- semantics
  - the semantics are deliberately different
  - **C++** templates are defined by **expansion**
    - in C++ templates, each instance of a template at a new type is compiled separately. If you use a list of integers, a list of strings, and a list of lists of string, there will be three versions of the code. If you use lists of a hundred different types, there will be a hundred versions of the code (a problem known as code bloat)
    - *code bloat is the production of code that is perceived as unnecessarily long, slow, or otherwise wasteful of resources*
  - **Java** generics are defined by **erasure**
    - in Java there is always one version of the code, so bloat does not occur
- **expansion** may lead to **more efficient** implementation than erasure, since it offers more opportunities for optimization, particularly for primitive types such as int
  - for code that is manipulating large amounts of data (for instance, large arrays in scientific computing) this difference may be significant

parametric polymorphism

- Java has parametric polymorphism as found in functional languages such as ML and Haskell
- parametric polymorphism is a way to make a language more expressive, while still maintaining full static type-safety. Using parametric polymorphism, a function or a data type can be written generically so that it can handle values identically without depending on their type. Such functions and data types are called generic functions and generic datatypes respectively and form the basis of generic programming.
- parametric polymorphism may be contrasted with ad hoc polymorphism, in which a single polymorphic function can have a number of distinct and potentially heterogeneous implementations depending on the type of argument(s) to which it is applied. Thus, ad hoc polymorphism can generally only support a limited number of such distinct types, since a separate implementation has to be provided for each type. It is also known as function overloading or operator overloading
- early binding - ad hoc polymorphism is a dispatch mechanism: control moving through one named function is dispatched to various other functions without having to specify the exact function being called. Overloading allows multiple functions taking different types to be defined with the same name; the compiler or interpreter automatically ensures that the right function is called.
- using overloading, it is possible to have a function perform two completely different things based on the type of input passed to it (each optimized for the particular data types); this is not possible with parametric polymorphism.
- late binding - in Smalltalk, the overloading is done at run time, as the methods for each overloaded message are resolved when they are about to be executed. This happens at run time, after the program is compiled. Therefore, polymorphism is given by subtyping polymorphism as in other languages, and it is also extended in functionality by ad hoc polymorphism at run time.

Generics & Go

Go group discussion

- arguments in favor of generics
  - people can **write code once**, saving coding time
  - people can **fix a bug in one instance** without having to remember to fix it in others
  - generics **avoid boilerplate**: less coding by copying and editing
  - generics **save time testing code**: they increase the amount of code that can be type checked at compile time rather than at run time

Go & generics

- Go already supports a form of generic programming via interfaces
- People can write an abstract algorithm that works with any type that implements the interface
  - However, interfaces are limited because the methods must use specific types
  - There is no way to write an interface with a method that takes an argument of type T, for any T, and returns a value of the same type
  - There is no way to write an interface with a method that compares two values of the same type T, for any T

arguments in favor of generics in Go

- generics permit **type-safe polymorphic containers**
- Go currently has a very limited set of such containers: **slices**, and **maps** of **most** but not all **types**. Not every program can be written using a slice or map.
- look at the functions SortInts, SortFloats, SortStrings in the sort package. Or SearchInts, SearchFloats, SearchStrings. Or the Len, Less, and Swap methods of byName in package io/ioutil. ***Pure boilerplate copying***

"what we want from generics in Go"

- define generic types based on types that are not known until they are instantiated
- write algorithms to operate on values of these types
- name generic types and name specific instantiations of generic types
- restrict the set of types that may be used to instantiate a generic type, to ensure that the generic type is only instantiated with types that support the required operations
- do not require an explicit relationship between the definition of a generic type or function and its use. That is, programs should not have to explicitly say type T implements generic G
- do not require explicit instantiation of generic types or functions; they should be instantiated as needed

the downsides of generics

- generics **affect the whole language**
  - it is necessary to evaluate every single language construct to see how it will work with generics
- generics affect the whole standard library
  - it is desirable to have the standard library make effective use of generics
  - every existing package should be reconsidered to see whether it would benefit from using generics
- generics are a trade off between programmer time, compilation time, and execution time
  - Go is currently optimizing compilation time and execution time at the expense of programmer time
  - Compilation time is a significant benefit of Go
  - Can we retain compilation time benefits without sacrificing too much execution time?
- Go has a **lightweight type system**. Adding generic types inevitably makes the type system more complex. It is essential that the result remain lightweight

Conclusion

- generics will make the language
  - **safer**
  - **more efficient to use**
  - **more powerful**
- these *advantages* are harder to quantify than the *disadvantages*, but they are real
- examples of potential uses of generics in Go
  - containers
    - user-written hash tables that are compile-time type-safe, rather than converting slice keys to string and using maps
    - sorted maps
    - ...
  - generic algorithms that work with these containers in a type-safe way
    - union/intersection
    - sort, find
    - ...

Generics & Haskell

Haskell: generic & type construcor

- Haskell (refer to Tomaiuolo slides)
- Nullary type constructor has zero arguments
  - data Bool = True | False
- A type constructor may have arguments
  - Example with one argument:
  - data Tree a = Tip | Node a (Tree a) (Tree a)
- The data type is polymorphic (and a is a type variable that is to be substituted by a specific type).
  - So when used, the values will have types like Tree Int or Tree (Tree Boolean)