

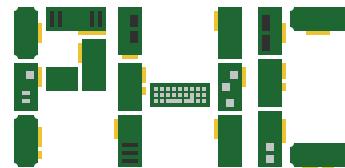
Introduzione alle Generics in Go

Chi sono?

Antonio De Lucreziis, studente di Matematica e macchinista
del PHC

Cos'è il PHC?

Il PHC è un gruppo di studenti di Matematica con interessi per, open source, Linux, self-hosting e soprattutto smanettare sia con hardware e software (veniteci pure a trovare!)



The Go 1.18 release adds support for generics. Generics are the biggest change we've made to Go since the first open source release

II Problema

```
func Min(x, y int) int {  
    if x < y {  
        return x  
    }  
  
    return y  
}
```

```
func MinInt8(x, y int8) int8 {
    if x < y {
        return x
    }

    return y
}

func MinInt16(x, y int16) int16 {
    if x < y {
        return x
    }

    return y
}

func MinFloat32(x, y float32) float32 {
    if x < y {
        return x
    }

    return y
}
```

```
...
if x < y {
    return x
}

return y
...
```

Soluzioni Pre-Generics

- Fare una funzione che prende `any` ed usare degli switch sul tipo
- Copia incollare tante volte la funzione per ogni tipo
- Utilizzare tool come `go generate`

Soluzione Post-Generics

Type Parameters

```
import "golang.org/x/exp/constraints"

func Min[T constraints.Ordered](x, y T) T {
    if x < y {
        return x
    }
    return y
}
```

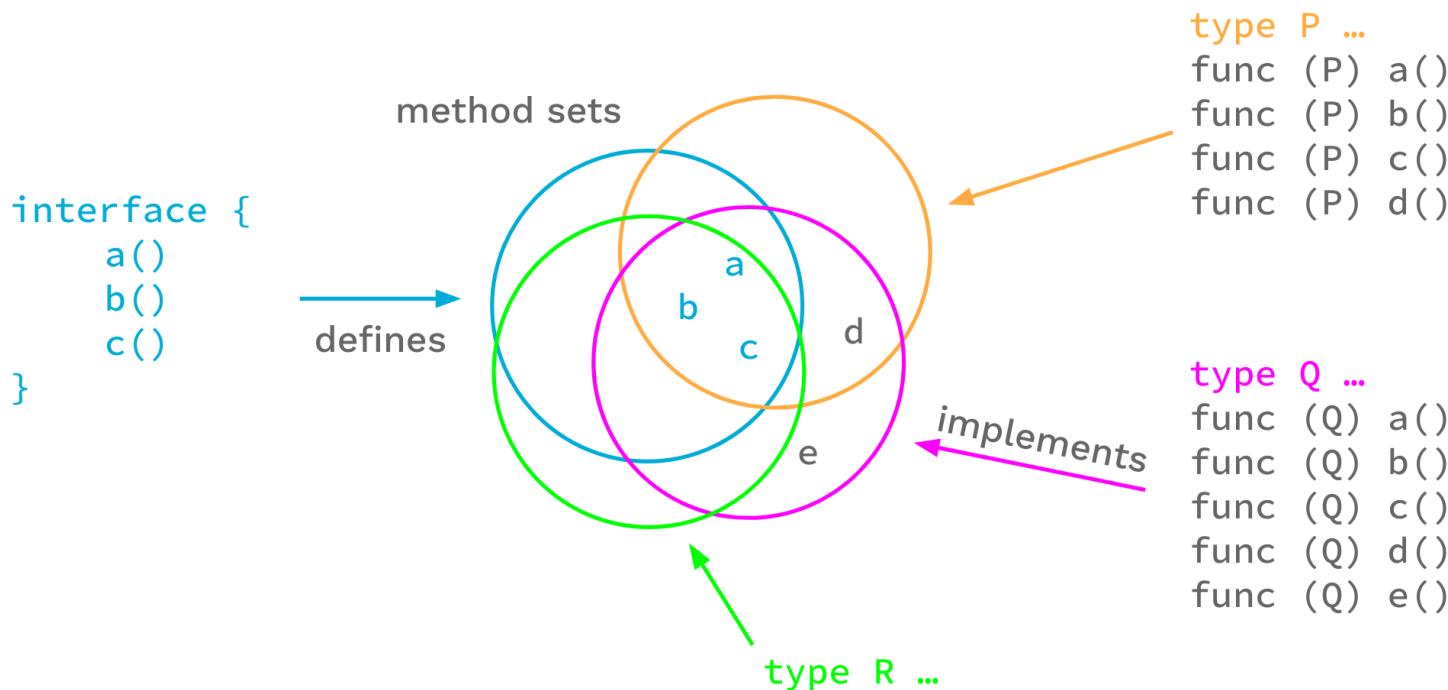
```
var a, b int = 0, 1
Min[int](a, b)
...
var a, b float32 = 3.14, 2.71
Min[float32](a, b)
```

Type Inference

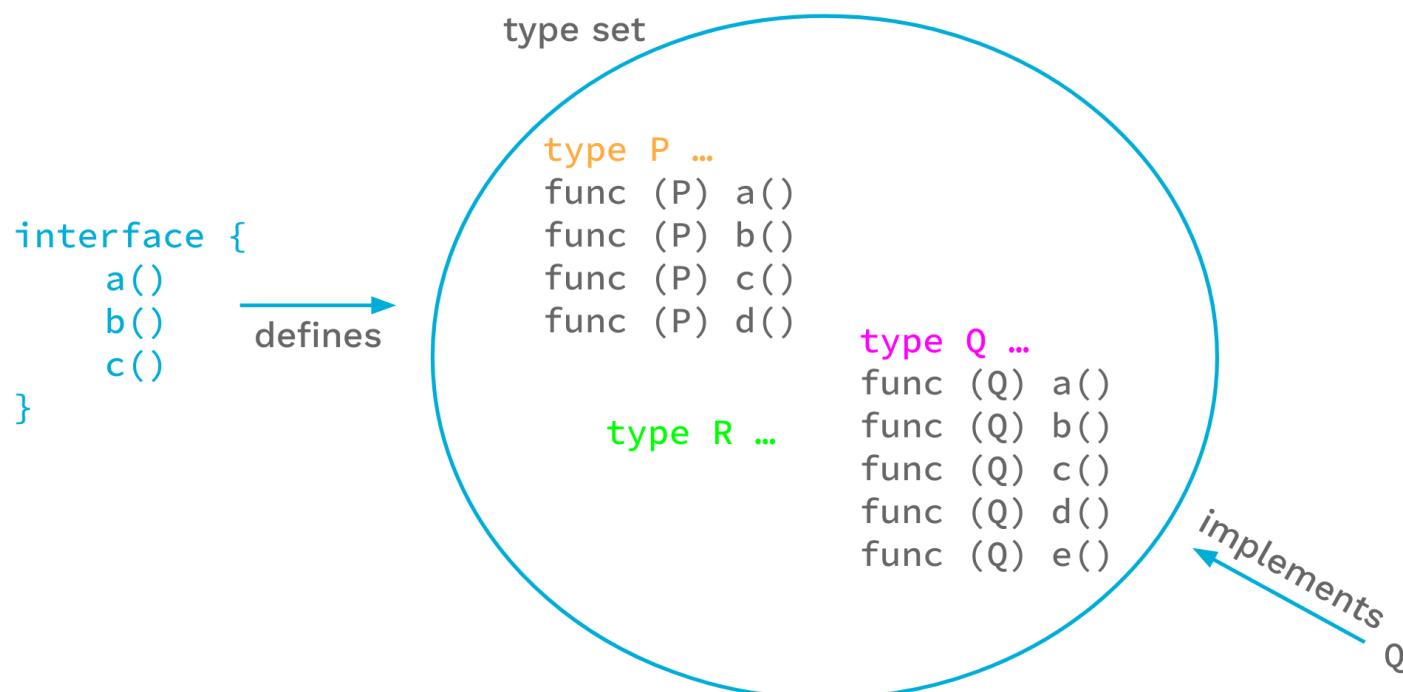
```
var a, b int = 0, 1
Min(a, b)
...
var a, b float32 = 3.14, 2.71
Min(a, b)
```

```
[T Vincolo1, R interface{ Method(), ... }, ... ]
```

Type Sets



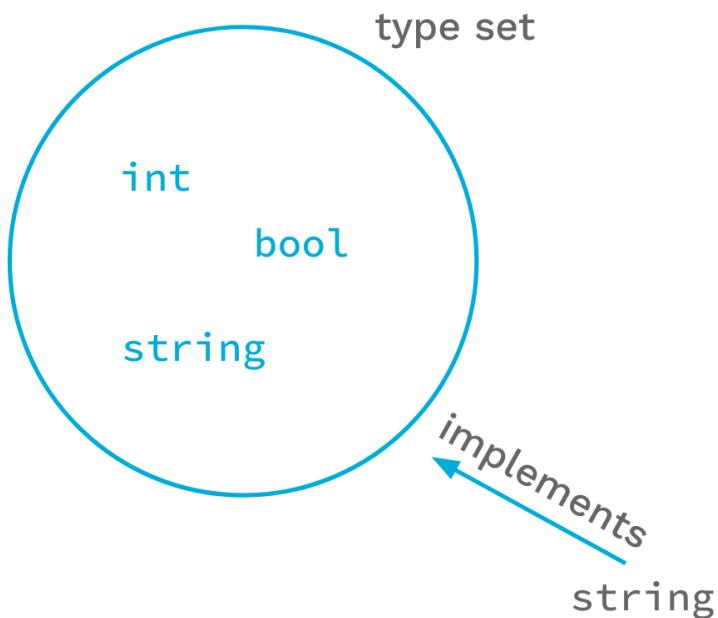
Type Sets



Type Sets

```
interface {  
    int|string|bool  
}
```

defines



<https://go.dev/blog/intro-generics>

Type Sets (Sintassi)

- `[T interface{}]` si può anche scrivere `[T any]`
- `[T interface{ int | float32 }]` si può anche scrivere `[T int | float32]`

Type Sets

```
func Somma[T float32|float64](x, y T) T {  
    return x + y  
}
```

```
type Liter float64
```

```
var a, b int = 1, 2  
Somma(a, b) // Ok
```

```
var a, b Liter = 1, 2  
Somma(a, b) // Errore
```

Type Sets

```
func Somma[T ~float32|~float64](x, y T) T {  
    return x + y  
}
```

```
type Liter float64
```

```
var a, b int = 1, 2  
Somma(a, b) // Ok
```

```
var a, b Liter = 1, 2  
Somma(a, b) // Ok
```

Type Sets

```
package constraints

...
type Float interface {
    ~float32 | ~float64
}
...
```

Type Sets

```
package constraints

...

type Ordered interface {
    Integer | Float | ~string
}

type Float interface {
    ~float32 | ~float64
}

type Integer interface {
    Signed | Unsigned
}

type Signed interface {
    ~int | ~int8 | ~int16 | ~int32 | ~int64
}

type Unsigned interface {
    ~uint | ~uint8 | ~uint16 | ~uint32 | ~uint64 | ~uintptr
}

...
```

Tipi Generici

```
type Stack[T any] []T
```

```
func (s *Stack[T]) Push(value T) {
    *s = append(*s, value)
}
```

```
func (s Stack[T]) Peek() T {
    return s[len(s)-1]
}
```

```
func (s Stack[T]) Len() int {
    return len(s)
}
```

```
func (s *Stack[T]) Pop() (T, bool) {
    items := *s

    if len(items) == 0 {
        var zero T
        return zero, false
    }

    newStack, poppedValue := items[:len(items)-1], items[len(items)-1]
    *s = newStack

    return poppedValue, true
}
```

Per ora ci tocca utilizzare questa funzione di *utility*

```
func Zero[T any]() T {  
    var zero T  
    return zero  
}
```

🔗 [43651-type-parameters.md#the-zero-value](#)

Pattern: Tipi Contenitore

Tipi generici nativi

- `[n]T`

Array di `n` elementi per il tipo `T`

- `[]T`

Slice per il tipo `T`

- `map[K]V`

Mappe con chiavi `K` e valori `V`

- `chan T`

Canali per elementi di tipo `T`

golang.org/x/exp/slices

- `func Index[E comparable](s []E, v E) int`
- `func Equal[E comparable](s1, s2 []E) bool`
- `func Sort[E constraints.Ordered](x []E)`
- `func SortFunc[E any](x []E, less func(a, b E) bool)`
- e molte altre...

golang.org/x/exp/maps

- `func Keys[M ~map[K]V, K comparable, V any](m M) []K`
- `func Values[M ~map[K]V, K comparable, V any](m M) []V`
- e molte altre...

Strutture Dati Generiche

Esempio notevole: <https://github.com/zyedidia/generic> (1K★ su GitHub)

- `mapset.Set[T comparable]`, set basato su un dizionario.
- `multimap.MultiMap[K, V]`, dizionario con anche più di un valore per chiave.
- `stack.Stack[T]`, slice ma con un'interfaccia più simpatica rispetto al modo idiomatico del Go.
- `cache.Cache[K comparable, V any]`, dizionario basato su `map[K]V` con una taglia massima e rimuove gli elementi usando la strategia LRU.
- `bimap.Bimap[K, V comparable]`, dizionario bi-direzionale.
- `hashmap.Map[K, V any]`, implementazione alternativa di `map[K]V` con supporto per *copy-on-write*.
- e molte altre...

Anti-Pattern (1)

Utility HTTP

```
// library code
type Validator interface {
    Validate() error
}

func DecodeAndValidateJSON[T Validator](r *http.Request) (T, error) {
    var value T
    if err := json.NewDecoder(r.Body).Decode(&value); err != nil {
        var zero T
        return zero, err
    }

    if err := value.Validate(); err != nil {
        var zero T
        return zero, err
    }

    return value, nil
}
```

```
// client code
type FooRequest struct {
    A int      `json:"a"`
    B string   `json:"b"`
}

func (foo FooRequest) Validate() error {
    if foo.A < 0 {
        return fmt.Errorf(`parameter "a" cannot be lesser than zero`)
    }
    if !strings.HasPrefix(foo.B, "baz-") {
        return fmt.Errorf(`parameter "b" has wrong prefix`)
    }

    return nil
}
```

```
foo, err := DecodeAndValidateJSON[FooRequest](r)
if err != nil {
    http.Error(w, err.Error(), http.StatusInternalServerError)
    return
}
```

```
func DecodeAndValidateJSON(r *http.Request, target Validator) error {
    err := json.NewDecoder(r.Body).Decode(target)
    if err != nil {
        return err
    }

    if err := target.Validate(); err != nil {
        return err
    }

    return nil
}
```

```
var foo FooRequest
if err := DecodeAndValidateJSON(r, &foo); err != nil {
    http.Error(w, err.Error(), http.StatusInternalServerError)
    return
}
```

Anti-Pattern (2)

Generics vs Interface

Memento Quiz

```
func WriteOneByte(w io.Writer, data byte) {
    w.Write([]byte{data})
}

d := &bytes.Buffer{}
WriteOneByte(d, 42)
```

```
func WriteOneByte[T io.Writer](w T, data byte) {
    w.Write([]byte{data})
}

d := &bytes.Buffer{}
WriteOneByte[*bytes.Buffer](d, 42)
```

BenchmarkInterface		
BenchmarkInterface-4	135735110	9.017 ns/op
BenchmarkGeneric		
BenchmarkGeneric-4	50947912	22.26 ns/op

```
//go:noinline
func WriteOneByte(w io.Writer, data byte) {
    w.Write([]byte{data})
}

...
d := &bytes.Buffer{}
WriteOneByte(d, 42)
```

BenchmarkInterface		
BenchmarkInterface-4	135735110	9.017 ns/op
BenchmarkInterfaceNoInline		
BenchmarkInterfaceNoInline-4	46183813	23.64 ns/op
BenchmarkGeneric		
BenchmarkGeneric-4	50947912	22.26 ns/op

```
d := &bytes.Buffer{} /* (*bytes.Buffer) */  
WriteOneByte(d /* (io.Writer) */, 42)
```



```
d := &bytes.Buffer{} /* (*bytes.Buffer) */  
(io.Writer).Write(d /* (io.Writer) */, []byte{ 42 })
```



```
d := &bytes.Buffer{} /* (*bytes.Buffer) */  
(*bytes.Buffer).Write(d /* (*bytes.Buffer) */, []byte{ 42 })
```

Go 1.18 Implementation of Generics via Dictionaries and Gcshape Stenciling

- A **gcshape** (or *gcshape grouping*) is a collection of types that all share the same instantiation of a generic function/method.
- Two concrete types are in the same gcshape grouping if and only if they have the same **underlying type** or they are **both pointer types**.
- To avoid creating a different function instantiation for each generic call with distinct type arguments (which would be pure stenciling), we **pass a dictionary along with every call**.

Quindi nella maggior parte dei casi se ci ritroviamo a scrivere una funzione generica con un **parametro vincolato ad un'interfaccia** forse dobbiamo porci qualche domanda

Pattern: Type-safe Database

Vediamo un analogo di `PhantomData<T>` dal Rust per rendere *type-safe* l'interfaccia di una libreria

Proviamo ad usare questa tecnica per rendere *type-safe*
l'interfaccia con `*sql.DB`

```
type DatabaseRef[T any] string
```

```
package tables

// tables metadata
var Users = database.Table[User]{ ... }
var Products = database.Table[Product]{ ... }
```

```
userRef1 := DatabaseRef[User]("j.smith@example.org")
...
// Ok
user1, err := database.Read(dbConn, tables.Users, userRef1)
// Errore
user2, err := database.Read(dbConn, tables.Products, userRef1)
```

```
package database

type WithPK interface {
    PrimaryKey() *string
}

type Ref[T WithPK] string

type Table[T WithPK] struct {
    Name      string
    PkColumn string
    Columns   func(*T) []any
}

...

func Read[T WithPK](d DB, t Table[T], ref Ref[T]) (*T, error)
```

```
package database

func Create[T WithPK](d DB, t Table[T], row T) (Ref[T], error)

func Insert[T WithPK](d DB, t Table[T], row T) (Ref[T], error)

func Read[T WithPK](d DB, t Table[T], ref Ref[T]) (*T, error)

func Update[T WithPK](d DB, t Table[T], row T) error

func Delete[T WithPK](d DB, t Table[T], id Ref[T]) error
```

```
func Read[T WithPK](d DB, t Table[T], ref Ref[T]) (*T, error) {
    result := d.QueryRow(
        fmt.Sprintf(
            `SELECT * FROM %s WHERE %s = ?`,
            t.Name, t.PkColumn,
        ),
        string(ref),
    )

    var value T
    if err := result.Scan(t.Columns(&value)...); err != nil {
        return nil, err
    }

    return &value, nil
}
```

```
package model

type User struct {
    Username string
    FullName string
    Age      int
}

func (u *User) PrimaryKey() *string {
    return &u.Username
}
```

```
package tables

var Users = Table[User]{
    Name: "users",
    PkColumn: "username",
    Columns: func(u *User) []any {
        return []any{ &u.Username, &u.FullName, &u.Age }
    }
}
```

Quindi possiamo anche utilizzare le **generics** per rendere **type-safe** l'interfaccia di qualcosa che inizialmente non lo era.

Pattern: *Channels*

Alcune utility per lavorare meglio con i *channel*

```
func trySend[T any](c chan<- T, v T) bool {
    select {
    case c <- v:
        return true
    default:
        return false
    }
}
```

```
func raceSame[T any](cs ...<-chan T) T {
    done := make(chan T)
    defer close(done)

    for _, c := range cs {
        go func(c <-chan T) {
            trySend(done, <-c)
        }(c)
    }

    return <-done
}
```

```
type Awaiter interface {
    Await()
}

type awaiterChan[T any] <-chan T

func (ac awaiterChan[T]) Await() { <-ac }
```

```
type targetChan[T any] struct {
    c      <-chan T
    target *T
}

func (tc targetChan[T]) Await() { *tc.target = <-tc.c }
```

```
func race(rs ...Awaiter) {
    done := make(chan struct{})
    defer close(done)

    for _, r := range rs {
        go func(r Awaiter) {
            r.Await()
            trySend(done, struct{}{})
        }(r)
    }

    <-done
}
```

```
var result2 int
var result3 float64

raceAny(
    awaiterChan[string](c1),
    targetChan[int]{c2, &result2},
    targetChan[float64]{c3, &result3},
)
fmt.Println(result2, result3)
```

```
var result2 int
var result3 float64

// Variante più pulita di questa utility
channels.Race(
    channels.Awaiter(c1),
    channels.Awaiter(c2, channels.WithTarget(&result2)),
    channels.Awaiter(c3, channels.WithTarget(&result3)),
)
fmt.Println(result2, result3)
```

$$1 + 1 = 2$$

Proof checking in Go

Premesse

Definiamo i possibili "tipi" delle nostre espressioni

```
type Bool interface{ isBool() }
```

```
type Nat interface{ isNat() }
```

```
type Nat2Nat interface{ isNat2Nat() }
```

Premesse

Trick per codificare higher-kinded types in Go

```
type V[ H Nat2Nat, T Nat ] Nat
```

Assiomi dei Naturali

```
type Zero Nat
type Succ Nat2Nat

// Alcuni alias utili
type One = V[Succ, Zero]
type Two = V[Succ, V[Succ, Zero]]
type Three = V[Succ, V[Succ, V[Succ, Zero]]]
```

Uguaglianza

```
type Eq[A, B any] Bool

// Eq_Refl ovvero l'assioma
//   forall x : x = x
func Eq_Reflexive[T any]() Eq[T, T] {
    panic("axiom: comptime only")
}

// Eq_Symmetric ovvero l'assioma
//   forall a, b: a = b => b = a
func Eq_Symmetric[A, B any](_ Eq[A, B]) Eq[B, A] {
    panic("axiom: comptime only")
}

// Eq_Transitive ovvero l'assioma
//   forall a, b, c: a = b e b = c => a = c
func Eq_Transitive[A, B, C any](_ Eq[A, B], _ Eq[B, C]) Eq[A, C] {
    panic("axiom: comptime only")
}
```

Uguaglianza e Sostituzione

Per ogni funzione F , ovvero tipo vincolato all'interfaccia
Nat2Nat vorremmo dire che

$$\text{Eq}[A , B] \xrightarrow{F} \text{Eq}[F[A] , F[B]]$$

Uguaglianza e Sostituzione

Data una funzione ed una dimostrazione che due cose sono uguali allora possiamo applicare la funzione ed ottenere altre cose uguali

```
// Function_Eq ovvero l'assioma
//  forall f function, forall a, b nat: a = b  => f(a) = f(b)
func Function_Eq[F Nat2Nat, A, B Nat](_ Eq[A, B]) Eq[V[F, A], V[F, B]] {
    panic("axiom: comptime only")
}
```

Assiomi dell'addizione

```
type Plus[L, R Nat] Nat

// "n + 0 = n"

// Plus_Zero ovvero l'assioma
//   forall n, m: n + succ(m) = succ(n + m)
func Plus_Zero[N Nat]() Eq[Plus[N, Zero], N] {
    panic("axiom: comptime only")
}

// "n + (m + 1) = (n + m) + 1"

// Plus_Sum ovvero l'assioma
//   forall n, m: n + succ(m) = succ(n + m)
func Plus_Sum[N, M Nat]() Eq[
    Plus[N, V[Succ, M]],
    V[Succ, Plus[N, M]],
] { panic("axiom: comptime only") }
```

$$1 + 1 = 2$$

```
func Theorem_OnePlusOneEqTwo() Eq[Plus[One, One], Two] {
    // 1 + 0 = 1
    // en1 :: Eq[ Plus[One, Zero], One ]
    en1 := Plus_Zero[One]()

    // (1 + 0) + 1 = 2
    // en2 :: Eq[ V[Succ, Plus[One, Zero]], Two ]
    en2 := Function_Eq[Succ](en1)

    // 1 + 1 = (1 + 0) + 1
    // en3 :: Eq[ Plus[One, One], V[Succ, Plus[One, Zero]] ]
    en3 := Plus_Sum[One, Zero]()

    return Eq_Transitive(en3, en2)
}
```

Conclusione

Regole generali

Per scrivere *codice generico* in Go

- Se l'implementazione dell'operazione che vogliamo supportare non dipende del tipo usato allora conviene usare dei **type-parameter**
- Se invece dipende dal tipo usato allora è meglio usare delle **interfacce**
- Se invece dipende sia dal tipo e deve anche funzionare per tipi che non supportano metodi (ad esempio per i tipi primitivi) allora conviene usare **reflection**

Fine :C

Domande

Bibliografia

- <https://go.dev/blog/intro-generics>
- <https://go.dev/blog/when-generics>
- <https://github.com/golang/proposal/blob/master/design/generics-implementation-dictionaries-go1.18.md>