So you want to add variants to Go?

Axel Wagner

https://blog.merovius.de/

@Merovius@chaos.social

2025-11-06

Prior art

Prior art

```haskell
data Shape =
        Circle Double |           -- radius
        Square Double |           -- side
        Rectangle Double Double -- width, height
```

Prior art

```haskell
data Shape =
        Circle Double |            -- radius
        Square Double |            -- side
        Rectangle Double Double -- width, height
```

```rust
enum Shape {
        Circle { radius: f64 },
        Square { side: f64 },
        Rectangle { length: f64, height: f64 },
}
```

Prior art

```haskell
data Shape =
        Circle Double |           -- radius
        Square Double |           -- side
        Rectangle Double Double -- width, height
```

```rust
enum Shape {
        Circle { radius: f64 },
        Square { side: f64 },
        Rectangle { length: f64, height: f64 },
}
```

🤷

Prior art

```haskell
data Shape =
        Circle Double |           -- radius
        Square Double |           -- side
        Rectangle Double Double -- width, height
```

```rust
enum Shape {
        Circle { radius: f64 },
        Square { side: f64 },
        Rectangle { length: f64, height: f64 },
}
```

```c
struct Circle { double radius; };
struct Square { double side; };
struct Rectangle { double length; double height; };

union Shape {
        struct Circle circle;
        struct Square square;
        struct Rectangle rectangle;
}
```

Prior art

```haskell
data Shape =
        Circle Double |           -- radius
        Square Double |           -- side
        Rectangle Double Double -- width, height
```

```rust
enum Shape {
        Circle { radius: f64 },
        Square { side: f64 },
        Rectangle { length: f64, height: f64 },
}
```

```typescript
type Circle = { radius: number };
type Square = { side: number };
type Rectangle = { length: number; height: number };

type Shape = Circle | Square | Rectangle;
```

```c
struct Circle { double radius; };
struct Square { double side; };
struct Rectangle { double length; double height; };

union Shape {
        struct Circle circle;
        struct Square square;
        struct Rectangle rectangle;
}
```

Prior art

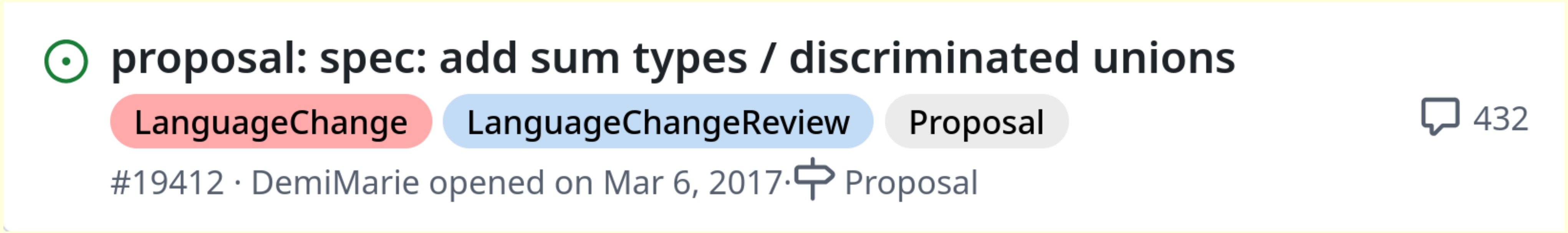
```haskell
data Shape =
        Circle Double |           -- radius
        Square Double |           -- side
        Rectangle Double Double -- width, height
```

```python
class Circle(NamedTuple):
        radius: float

class Square(NamedTuple):
        side: float

class Rectangle(NamedTuple):
        length: float
        height: float

Shape = Circle | Square | Rectangle
```

```rust
enum Shape {
        Circle { radius: f64 },
        Square { side: f64 },
        Rectangle { length: f64, height: f64 },
}
```

```typescript
type Circle = { radius: number };
type Square = { side: number };
type Rectangle = { length: number; height: number };

type Shape = Circle | Square | Rectangle;
```

```c
struct Circle { double radius; };
struct Square { double side; };
struct Rectangle { double length; double height; };

union Shape {
        struct Circle circle;
        struct Square square;
        struct Rectangle rectangle;
}
```

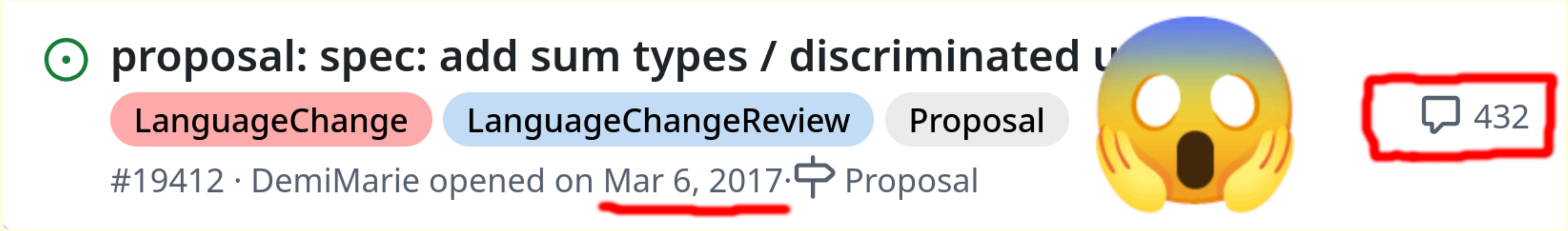


proposal: spec: add sum types / discriminated unions

LanguageChange LanguageChangeReview Proposal

432

#19412 · DemiMarie opened on Mar 6, 2017 · Proposal



proposal: spec: add sum types / discriminated u

LanguageChange LanguageChangeReview Proposal

432

#19412 · DemiMarie opened on Mar 6, 2017 · Proposal

What people want

What people want: closed list

```
type Circle struct { Radius float64 }
type Square struct { Side float64 }
type Rectangle struct { Height, Width float64 }

type Shape enum {
  Circle
  Square
  Rectangle
}

// Type error: Ellipsis is not in Shape
var s Shape = Ellipsis{}
```

What people want: closed list

```
type Circle struct { Radius float64 }
type Square struct { Side float64 }
type Rectangle struct { Height, Width float64 }

type Shape enum {
  Circle
  Square
  Rectangle
}

// Type error: Ellipsis is not in Shape
var s Shape = Ellipsis{}
```

Note: in this talk, I will **not worry** about specific syntax.

What people want: exhaustiveness check

```
func F(s Shape) {
  switch s := s.(type) {
  case Circle:
    DoCircleThing(s)
  case Square:
    DoSquareThing(s)
  // compiler error: missing case Rectangle
  }
}
```

What people want: well-defined state

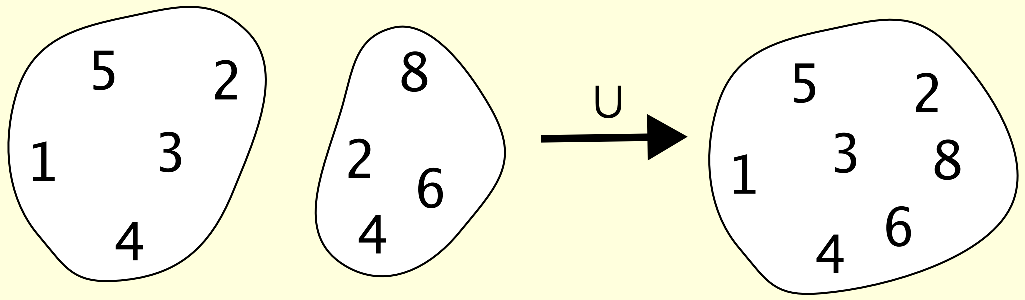
```
type Result[T any] enum {
  T
  error
}

// GetThing returns either a Thing, or an error;
// never both and never neither.
func GetThing() Result[Thing]
```

Design space

Unions and Sums

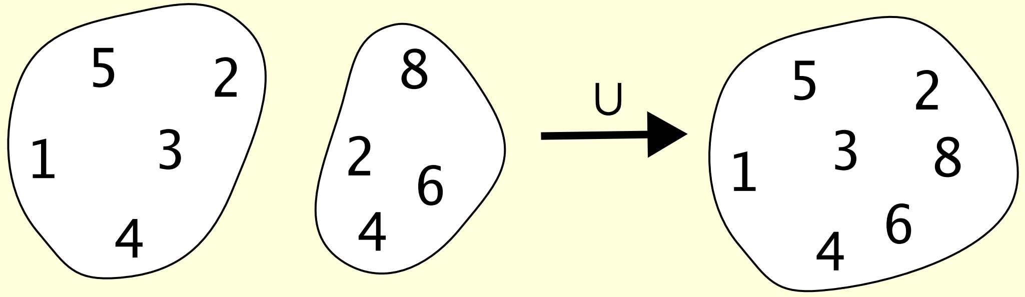
Unions and Sums

Union:

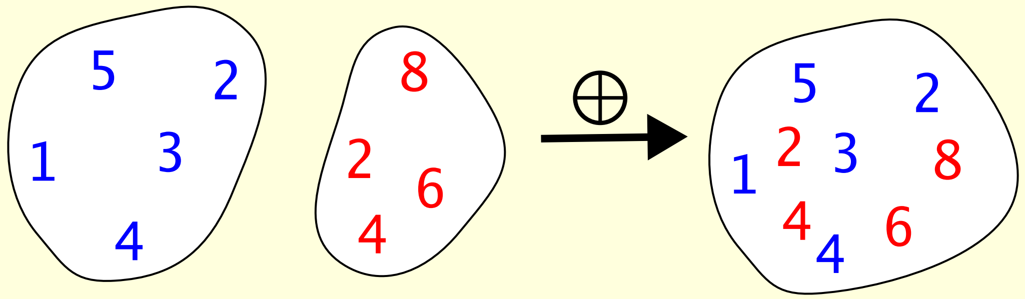


Unions and Sums

Union:



Sum:



Unions and Sums (concretely)

Typescript has unions:

```typescript
type MyUnion = number | string;
let x: MyUnion;
x = 42;
x = "Hello, world";
```

Rust has sums:

```rust
enum MySum {
    Int(i64),
    String(&'static str),
}


fn main() {
    let mut x: MySum;
    x = MySum::Int(42);
    x = MySum::String("Hello, world");
}
```

Unions and Sums (concretely)

Pretending go had unions:

```go
type MyUnion union {
  int,
  string,
}


func main() {
  var x MyUnion
  x = 42;
  x = "Hello, world";
}
```

Rust has sums:

```rust
enum MySum {
    Int(i64),
    String(&'static str),
}


fn main() {
    let mut x: MySum;
    x = MySum::Int(42);
    x = MySum::String("Hello, world");
}
```

Unions and Sums (concretely)

Pretending go had unions:

```
type MyUnion union {
  int,
  string,
  time.Duration,
}

func main() {
  var x MyUnion
  x = 42; // int or time.Duration?
  x = "Hello, world";
}
```

Rust has sums:

```
enum MySum {
    Int(i64),
    String(&'static str),
    Duration(i64),
}

fn main() {
    let mut x: MySum;
    x = MySum::Int(42); // clearly Int
    x = MySum::String("Hello, world");
}
```

Unpacking

Unpacking

- Overlapping cases?

Unpacking

- Overlapping cases?
- Exhaustiveness check?

Unpacking

- Overlapping cases?
- Exhaustiveness check?
- Forced `default`?

Unpacking

- Overlapping cases?
- Exhaustiveness check?
- Forced `default`?
- Pattern matching?

Unpacking

- Overlapping cases?
- Exhaustiveness check?
- Forced `default`?
- Pattern matching?

```
fn degenerate(s: &Shape) -> bool {
  match s {
    Shape::Circle(0.0) => true,
    Shape::Square(0.0) => true,
    Shape::Rectangle(0.0, _) => true,
    Shape::Rectangle(_, 0.0) => true,
    _ => false,
  }
}
```

Gradual code repair

Principle: APIs should be interoperable, when moved from one package to another.

Gradual code repair

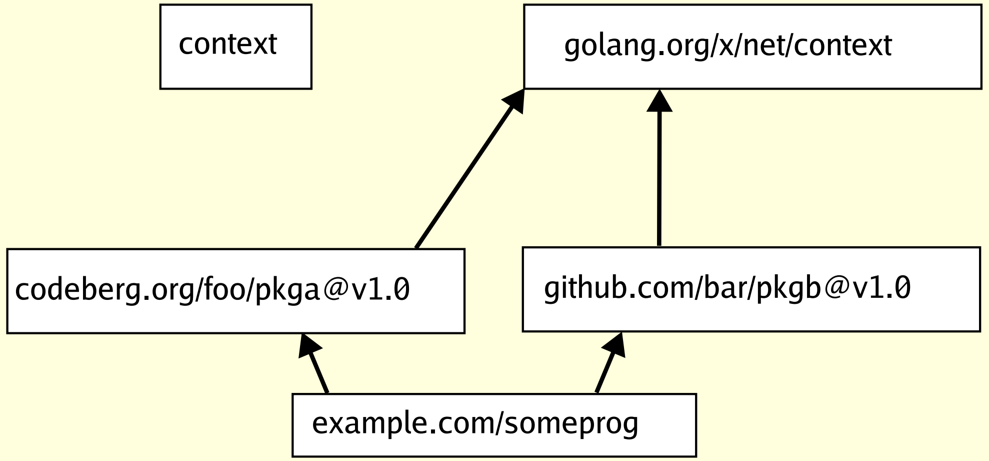
Principle: APIs should be interoperable, when moved from one package to another.

```
package context // import "golang.org/x/net/context"
import "context"
type Context = context.Context
```

Gradual code repair

Principle: APIs should be interoperable, when moved from one package to another.

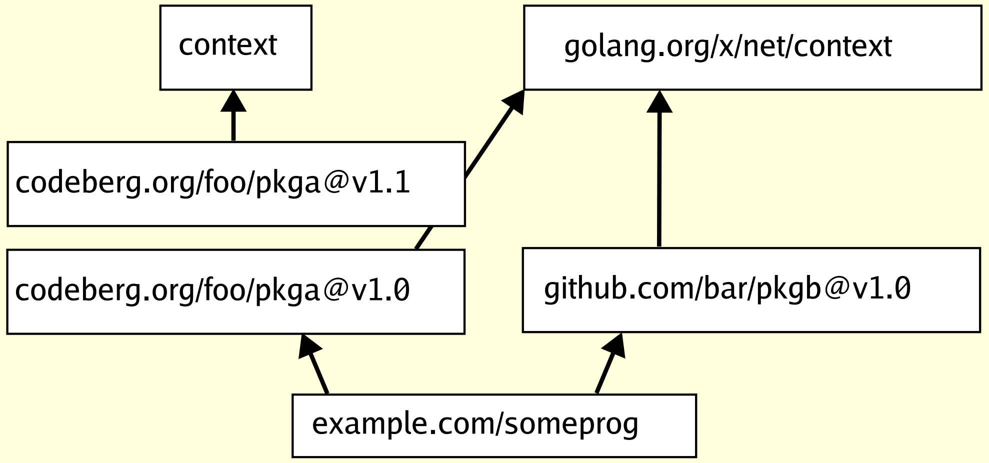
```go
package context // import "golang.org/x/net/context"
import "context"
type Context = context.Context
```



Gradual code repair

Principle: APIs should be interoperable, when moved from one package to another.

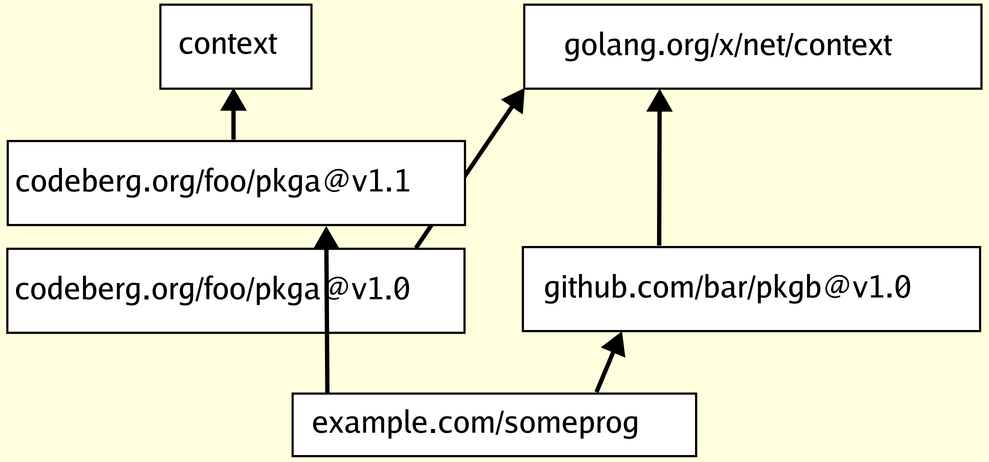
```
package context // import "golang.org/x/net/context"
import "context"
type Context = context.Context
```



Gradual code repair

Principle: APIs should be interoperable, when moved from one package to another.

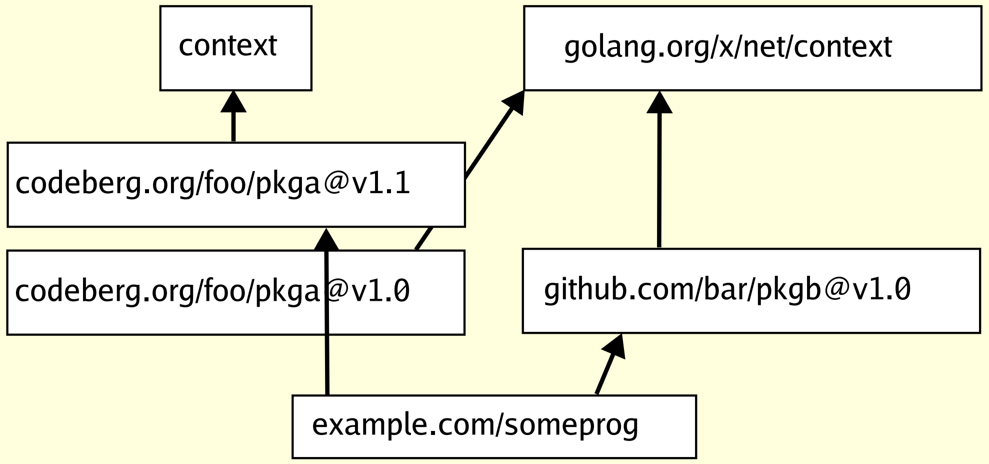
```
package context // import "golang.org/x/net/context"
import "context"
type Context = context.Context
```



Gradual code repair

Principle: APIs should be interoperable, when moved from one package to another.

```
package context // import "golang.org/x/net/context"
import "context"
type Context = context.Context
```



Exhaustive switch checking breaks gradual repair!

Zero values

Every Go type needs a **zero value**, which should be represented by `0` bytes

Zero values

Every Go type needs a **zero value**, which should be represented by `0` bytes

1. Use zero value of “default term”

Zero values

Every Go type needs a **zero value**, which should be represented by `0` bytes

1. Use zero value of “default term”
   - Default is explicitly marked
   - Default is implied by order (usually the first term)

Zero values

Every Go type needs a **zero value**, which should be represented by `0` bytes

1. Use zero value of “default term”
   - Default is explicitly marked
   - Default is implied by order (usually the first term)
2. Use sentinel for “no value”

Variants in Go

Union elements

```
type Shape interface{ Circle | Square | Rectangle }

func F[S Shape](v S) {}
// cannot use type Shape outside a type constraint:
//   interface contains type constraints
func G(v Shape) {}
```

Union elements

```go
type Shape interface{ Circle | Square | Rectangle }

func F[S Shape](v S) {}


func G(v Shape) {
  // unpacking via type switch:
  switch v := v.(type) {
  case Circle:
  case Square:
  case Rectangle:
  }
}
```

Proposal: #57644

Consequence: nested unions

```
type Signed interface { int8 | … | int64 }
type Unsigned interface{ uint8 | … | uint64 }

type Integer interface { Signed | Unsigned }
```

Consequence: nested unions

```
type Signed interface { int8 | … | int64 }
type Unsigned interface{ uint8 | … | uint64 }

type Integer interface { Signed | Unsigned }
type Integer2 interface{ int8 | … | int64 | uint8 | … | uint64 }
```

Should remain the same ⇒ no nested unions.

Consequence: interface terms

```
type Result[T] interface{ T | error }
```

Consequence: interface terms

```go
// error: term cannot be a type parameter
// error: cannot use error in union (error contains methods)
type Result[T] interface{ T | error }
```

Interfaces with methods disallowed $\Longrightarrow$ no interfaces in unions.

More details: blog post and talk “Constraining Complexity in the Generics Design”

Consequence: zero value

```
type Union interface{ int | string }
```

Must remain valid ⇒ cannot require explicit default.

Consequence: zero value

```
type Union interface{ int | string }
```

Must remain valid ⟹ cannot require explicit default.

```
type Union2 interface{ string | int }
```

Should remain the same as `Union` ⟹ cannot use order for default.

Consequence: zero value

```
type Union interface{ int | string }
```

Must remain valid ⇒ cannot require explicit default.

```
type Union2 interface{ string | int }
```

Should remain the same as `Union` ⇒ cannot use order for default.

⇒ We must make the zero value `nil`.

Consequence: type switch

```
type Reader interface{ *bytes.Reader | *strings.Reader | int }
func F(r Reader) {
  switch r := r.(type) {
  case io.Reader:
  case *bytes.Reader: // never taken
  // case int not handled
  }
}
```

Consistency:

- No exhaustiveness check
- No forced default
- Overlapping cases allowed (choose first match)

Sum types

Alternative: add proper sum types.

Sum types

Alternative: add proper sum types.

Relatively easy to design (e.g. Proposal: unions as sigma types (#54685)).

Sum types

Alternative: add proper sum types.

Relatively easy to design (e.g. Proposal: unions as sigma types (#54685)).

Downside: Two concepts of “closed list of types”, with different uses, different syntax and different restrictions.

Sum types

Alternative: add proper sum types.

Relatively easy to design (e.g. Proposal: unions as sigma types (#54685)).

Downside: Two concepts of “closed list of types”, with different uses, different syntax and different restrictions.

$\Longrightarrow$ Probably too confusing and hard to learn.

Conclusion

There are three realistic options:

Conclusion

There are three realistic options:

1. Use existing union–elements: relatively low benefit, likely disappointing.

Conclusion

There are three realistic options:

1. Use existing union–elements: relatively low benefit, likely disappointing.
2. Add proper sum types: too redundant with union elements.

Conclusion

There are three realistic options:

1. Use existing union-elements: relatively low benefit, likely disappointing.
2. Add proper sum types: too redundant with union elements.
3. Do nothing and leave Go without variants.

Conclusion

There are three realistic options:

1. Use existing union–elements: relatively low benefit, likely disappointing.
2. Add proper sum types: too redundant with union elements.
3. Do nothing and leave Go without variants.

As the first two options are not clearly good, we are so far stuck with the third.

Thank you