型パラメータ（Type Parameters）プロポーザル

2021 年 3 月 19 日版

Ian Lance Taylor、Robert Griesemer　著

柴田芳樹　訳

2021 年 5 月 1 日

目次

第1章

プロポーザル

1.1 ステータス

このプロポーザルは、Go 言語に型パラメータを使ってジェネリックプログラミングを追加するためのデザインです。このデザインは、将来の言語の変更として提案され、受付られました[*1]。この変更は、2022 年初めの Go 1.18 リリースで利用可能になります。

1.2 要約

任意の**型パラメータ**（*type parameter*）を型宣言と関数宣言に追加する Go 言語の拡張を提案します。型パラメータは、インタフェース型によって制約されます。インタフェース型は、**型制約**（*type constraints*）として使われる場合、そのインタフェース型に代入可能な型の集合を列挙することを許します。**型推論**（*type inference* ）は単一化アルゴリズム（*unification algorithm*）を使って、多くの場合、関数呼び出しでの型引数（*type arguments*）の省略を許します。このデザインは、Go 1 と完全な後方互換があります。

1.3 プロポーザルの読み方

このドキュメントは長いです。次が、読み方に関するガイドです。

- 概念を簡潔に説明しながら、概要から始めます。
- それから、必要な詳細をサンプルコードと共に示しながら、一から完全なデザインを説明します。
- デザインをすべて説明した後、実装、デザインに関連する問題点、ジェネリックスに対する他の方法との比較を説明します。
- 次に、このデザインが、実際に使われるであろう完全なコード例を示します。
- コード例の後に、細かな事柄を付録で説明します。

[*1] `https://golang.org/issue/43651`

1.4　概要

この節では、このデザインが提案している変更を簡潔に説明します。この節は、Go に似た言語でのジェネリックスの仕組みにすでに慣れ親しんだ人達を対象にしています。次に説明する概念は、この後の節で詳細に説明します。

- 関数は角括弧（`[]`）を使う追加の**型パラメータ**（*type parameter*）のリストを持つことができ、角括弧という点を除けば、普通のパラメータのリストに見えます。たとえば、次の通りです。

  ```
  func F[T any](p T) { ... }
  ```

- このような型パラメータは、通常のパラメータと関数本体内で使えます。
- 型も型パラメータのリストを持てます。たとえば、次の通りです。

  ```
  type M[T any] []T
  ```

- 個々の通常のパラメータが型を持つのと同じように、個々の型パラメータは、一つの**型制約**（*type constraint*）を持ちます。たとえば、次の通りです。

  ```
  func F[T Constraint](p T) { ... }
  ```

- 型制約は、インタフェース型です。
- 新たな事前宣言名である `any` は、あらゆる型も許す型制約です。
- 型制約として使われるインタフェース型は、複数の事前宣言型のリストを持てます。それらの型のどれか一つに一致する**型引数**（*type argument*）だけが、制約を満足します。
- **ジェネリック関数**（*generic function*）は、その型制約で許される操作だけを使えます。
- ジェネリック関数あるいは**ジェネリック型**（*generic type*）を使うには、型引数を渡す必要があります。
- **型推論**（*type inference*）により、多くの場合、関数呼び出しの型引数の省略できます。

この後の節では、これらの言語の変更をそれぞれ詳細に説明します。このデザインに基づいて書かれたジェネリックのコードが実際にどのようなものであるかを知るためにコード例まで読み飛ばしても構いません。

1.5 背景

Go に「ジェネリックプログラミングのサポート」[*2] の追加を求める多くの要望が行われました。そのイシュートラッカー[*3] 上と随時更新される文書[*4] 上で広範囲な議論が行われました。

このデザインは、パラメトリック・ポリモフィズム（*parametric polymorphism*）の形式を追加するために Go 言語を拡張すること提案しており、そこでは、型パラメータは、（いくつかのオブジェクト指向言語にあるような）宣言されたサブタイピング関係によって境界が決まるのではなく、明示的に定義された構造的制約（*structural constraints*）によって決まります。

このデザインは、2019 年 7 月 31 日に提示したドラフトと多くの類似点を持っていますが、（そのドラフトで提示した）コントラクト（*construct*）は削除されてインタフェース型で置き換えられ、構文が変更されています。

型パラメータの追加に関するさまざまな提案が行われました。それらは、脚注のリンクをたどって見つけられます。このドキュメントで示されるアイデアの多くは以前に示されたものです。このドキュメントで説明される新たな主な機能は、構文と制約としてインタフェース型を注意深く調べたことです。

このデザインは、テンプレート・メタプログラミングやコンパイル時プログラミングといった他の形式はサポートしていません。

Go コミュニティでは用語**ジェネリック**（*generic*）は広く使われているので、これ以降は、型パラメータを取る関数あるいは型を意味する表現として使います。このデザインで使われている用語ジェネリックを C++、C#、Java、Rust といった他の言語での同じ用語と混同しないでください。似ていますが同じではありません。

1.6 デザイン

簡単なコード例に基づいて、完全なデザインを段階的に説明します。

1.6.1 型パラメータ

ジェネリックコードは、**型パラメータ** (*type parameter*) と呼ぶ抽象的なデータ型を用いて記述されます。ジェネリックコードを実行すると、その型パラメータは**型引数**（*type argument*）によって置き換えられます。

次は、スライスの各要素を表示する関数であり、スライスの要素型である `T` は不明です。これは、ジェネリックプログラミングをサポートするために許されて欲しい種類の関数の例

[*2] https://github.com/golang/go/wiki/ExperienceReports#generics
[*3] https://golang.org/issue/15292
[*4] https://docs.google.com/document/d/1vrAy9gMpMoS3uaVphB32uVXX4pi-HnNjkMEgyAHX4N4/view

です。（後で、**ジェネリック型**（*generic type*）について説明します。）

```
// Print はスライスの要素を表示します。
// あらゆるスライス値に対しても、この関数を呼び出せるべきです。
func Print(s []T) { // 例にすぎず、提案している構文ではありません。
    for _, v := range s {
        fmt.Println(v)
    }
}
```

この方法で最初に行うべき決定は、型パラメータ `T` をどのように宣言すべきかということです。Go のような言語では、すべての識別子が何らかの方法で宣言されます。

この例ではデザイン上の決定を行っています。すなわち、型パラメータは、通常の**非型**（*non-type*）の関数パラメータに似ていて、そのような型パラメータは、他のパラメータと一緒に使われるべきであるということです。しかし、型パラメータは、非型パラメータとは同じではないので、パラメータのリストに現れたとしても、それらを区別したいわけです。その結果、次のデザイン上の決定を導き出されます。つまり、省略可能で型パラメータを記述する追加のリストを定義することです。

この型パラメータのリストは、通常のパラメータの前に書かれます。型パラメータのリストを通常のパラメータのリストから区別するために、型パラメータのリストは丸括弧（`()`）ではなく角括弧（`[]`）を使います。通常のパラメータが型を持つのと同じように、型パラメータは、**制約**（*constraint*）として知られる**メタ型**（*meta-type*）を持ちます。制約の詳細については後で説明します。今のところ、`any` は、あらゆる型も許されるという意味の正当な制約であるということだけを述べておきます。

```
// Print はスライスの要素を表示します。
// Print は型パラメータ T を持ち、その型パラメータのスライスである
// 単一の（非型）パラメータ s を持っています。
func Print[T any](s []T) {
    // 前と同じ
}
```

これは、関数 `Print` 内では、識別子 `T` は型パラメータであり、現在は分かっていないけれど関数が呼び出される際には既知の型であると述べています。`any` は、あらゆる型でもよいことを意味します。上記の例で示されるように、型パラメータは、通常の非型パラメータの型を記述する際に型として使えます。また、関数の本体内で型としても使えます。

通常のパラメータのリストと異なり、型パラメータのリストでは、型パラメータに対する名前は必須です。その結果、構文的な曖昧さが排除されます。そして、偶然にも、型パラメータ名を省略する理由は何もありません。

`Print` は型パラメータを持っているので、`Print` のすべての呼び出しは型引数を提供しなければなりません。後で、**型推論**（*type inference*）を用いて、多くの場合で、どのように非

型引数からこの型引数が推定できるかを説明します。今のところ、引数とは別のリストとして、型パラメータが宣言されているのと同じように型引数を渡します。型パラメータのリストと同様に、型引数のリストは角括弧を使います。

```
// []int で Print を呼び出す。
// Print は型パラメータ T を持ち、[]int を渡したいので、
// Print[int] と書くことで int の型引数をわたしています。
// 関数 Print[int] は、引数として []int を期待しています。
Print[int]([]int{1, 2, 3})

// これは次を表示します:
// 1
// 2
// 3
```

1.6.2　制約

コード例を少し複雑にしてみます。任意の型のスライスの各要素に対して String メソッドを呼び出して []string へ変換する関数に修正してみます。

```
// この関数は不正です。
func Stringify[T any](s []T) (ret []string) {
    for _, v := range s {
        ret = append(ret, v.String()) // 不正
    }
    return ret
}
```

一見するとこのコードは良さそうに思えるかもしれませんが、v は T 型であり、T はあらゆる型が可能です。つまり、T は必ずしも String メソッドを持っていません。したがって、v.String() の呼び出しは不正です。

ジェネリックプログラミングをサポートしてい他の言語でも、もちろん同じ問題が発生します。たとえば、C++ では、ジェネリック関数（C++ の用語では、関数テンプレート）は、ジェネリック型の値に対してあらゆるメソッドを呼び出せます。すなわち、C++ の手法では、v.String() の呼び出しは問題がありません。String メソッドを持たない型引数でその関数が呼び出された場合、その型引数に対する v.String の呼び出しがコンパイルされる際に、エラーが報告されます。エラーが発生する前に複数レイヤのジェネリック関数の呼び出しが存在するかもしれないため、そのエラーが長くなることがあります。その長いエラーは、何が悪かったのかを理解するために報告されなければなりません。

C++ の手法は、Go にとっては悪い選択です。その理由の一つは、Go 言語のスタイルです。Go では、この場合の String といった名前だけを参照して、それが存在すると望むよ

うなことはしません。Go では、名前が現れた時、すべての名前をその宣言に結び付けます。

別の理由は、Go は規模が拡大するプログラミングをサポートするように設計されていることです。ジェネリック関数定義（上記の `Stringify`）とそのジェネリック関数の呼び出し（示されていませんが、おそらく別の他のパッケージ）がとても離れている場合を考慮しなければなりません。一般に、すべてのジェネリックコードは、型引数がある種の要件を満たすことを期待してます。そのような要件を**制約**（*constraint*）と呼びます（他の言語は、型境界（*type bound*[*5]）、トレイト境界（*trait bound*）[*6]、あるいはコンセプト（*concept*）[*7] といった似た考えを持っています）。この場合、制約は明らかです。つまり、型は `String() string` というメソッドを持っていなければなりません。他の場合では、それほど明らかではないかもしれません。

`Stringify` がたまたま何を行うのであろうと（この場合、`String` メソッドを呼び出す）、それが行う処理から制約を導き出したくはありません。もし、そうすると、`Stringify` に小さな変更が行われると制約が変わるかもしれません。それは、小さな変更によりコードが異なる動作になり、その関数の呼び出しが突然動かなくなってしまうことを意味しています。`Stringify` が意図して制約を変更し、呼び出しもとに変更を強いるのは問題ありません。避けたいのは、誤って制約が変わってしまうような `Stringify` です。

つまり、制約は、呼び出しもとが渡す型引数とジェネリック関数内のコードの両方に対して制限を課さなければならないことを意味します。呼び出しもとは、制約を満足させる型引数だけを渡せます。ジェネリック関数は、制約が許していることだけを型引数に対して行えます。これは、重要な規則であり、私達は、Go にジェネリックプログラミングを定義するすべての試みに適用されるべきだと考えています。つまり、ジェネリックコードは、その型引数が実装していると分かっている操作だけを使えます。

1.6.3　`any` 型に許されている操作

さらに制約を説明する前に、制約が `any` の場合、何が起きるのかを簡単に説明します。ジェネリック関数が、上記の `Print` メソッドの場合と同様に、型パラメータに対して `any` 制約を使うと、そのパラメータに対してあらゆる型引数が許されます。その型パラメータの値でジェネリック関数が使える唯一の操作は、すべての型の値に対して許されている操作です。上記の例では、`Print` 関数は、その型が型パラメータ `T` である変数 `v` を宣言し、その変数を別の関数へ渡しています。

`any` 型に対して許されている操作は次の通りです。

- その型の変数を宣言すること
- その型の他の値をその型の変数へ代入すること

[*5] 訳注：Java
[*6] 訳注：Rust
[*7] 訳注：C++

- その型の変数を関数へ渡したり、関数から返したりすること
- その型の変数のアドレスを得ること
- その型の値を `interface{}`型へ変換したり代入したりすること
- `T` 型の値を `T` 型へ変換すること（許されていますが、役立ちません）
- インタフェース値をその型へ変換するために、型アサーションを使うこと
- その型を、型 `switch` の `case` として使うこと
- その型のスライスといった、その型を使うコンポジット型を定義して使うこと
- その型を `new` といった事前宣言関数へ渡すこと

将来の言語の変更では、他の操作を追加する可能性はありますが、何かを追加することは、現在何も想定されていません。

1.6.4　制約を定義する

Go は、制約に必要とするものに近い構造をすでに持っています。それは、インタフェース型です。インタフェース型は、メソッドの集まりです。インタフェース型の変数へ代入できる唯一の値は、同じメソッドの集まりを実装している型の値です。インタフェース型の値で行える唯一の操作は、すべての型に対して許されている操作に加えて、それらのメソッドを呼び出すことです。

型引数でジェネリック関数を呼び出すのは、インタフェース型の変数に代入するのに似ています。つまり、型引数は、型パラメータの制約を実装していなければなりません。ジェネリック関数を書くことは、インタフェース型の値を使うことに似ています。つまり、ジェネリックコードは、制約で許された操作（あるいはすべての型に許されている操作）だけを使えます。

したがって、このデザインでは、制約は単純にインタフェース型です。制約を実装するということは、そのインタフェース型を実装することを意味します。（1.6.10節で、二項演算子といった、メソッド呼び出し以外の操作に対する制約の定義方法を説明します。）

`Stringify` のコード例に対しては、引数がなく `string` 型の値を返す `String` メソッドを持つインタフェース型が必要です。

```
// Stringer は、型引数が String メソッドを持つことを要求し、ジェネリック関数
// が String を呼び出すことを許す型制約です。
// String メソッドは値の文字列表現を返します。
type Stringer interface {
    String() string
}
```

（ここでの説明には関係ないですが、この定義は標準ライブラリの `fmt.Stringer` 型と

同じインタフェースを定義しており、現実のコードでは単純に fmt.Stringer を使うでしょう。）

1.6.5　any 制約

ここまでで制約は単純にインタフェース型であると分かっており、any が制約として何を意味するのかを説明します。前述したように、any 制約は型引数としてすべての型を許し、すべての型に許されている操作だけを関数が使うことを許しています。それと同じインタフェース型は空インタフェース、すなわち、interface{} です。したがって、Print のコード例を次のようにも書き直せます。

```
// Print はスライスの要素を表示します。
// Print は型パラメータ T を持ち、その型パラメータのスライスである
// 単一の（非型）パラメータ s を持っています。
func Print[T interface{}](s []T) {
    // 前と同じ
}
```

しかし、型パラメータに制約を課さないジェネリック関数を書くごとに interface{} を書かなければならないのは面倒です。したがって、このデザインでは、interface{} と等価な型制約 any を提案しています。any は事前宣言名であり、ユニバースブロック（*universe block*）[*8] で暗黙に宣言されています。型制約以外の用途で any を使うのは不正です。

（注意：interface{} に対するエイリアスあるいは interface{} として定義された新たな定義型（*defined type*）として、明らかに any を一般的に利用可能にできました。しかし、ジェネリックスに関するこのデザインが、ジェネリックではないコードに対する大きな変更に繋がって欲しくはありません。interface{} に対する汎用的な名前としての any の追加は別に議論されるべきです[*9]。）

1.6.6　制約を使う

ジェネリック関数に対して、制約は型引数の型、つまりメタ型（*meta-type*）と見なせます。前に示したように、制約は、型パラメータのメタ型として、型パラメータのリストに書かれます。

```
// Stringify は s の各要素に対して String メソッドを呼び出して、
// 結果を返します。
func Stringify[T Stringer](s []T) (ret []string) {
    for _, v := range s {
```

[*8] 訳注：ソースコード全体となるレキシカルブロックです。

[*9] https://golang.org/issue/33232

```
            ret = append(ret, v.String())
        }
        return ret
    }
```

単一の型パラメータ T の後に、T に適用される制約が続きます。この場合は、Stringer です。

1.6.7 複数の型パラメータ

Stringify のコード例は単一の型パラメータしか使っていませんが、関数は複数の型パラメータを持てます。

```
// Print2 は二つの型パラメータと二つの非型パラメータを持ちます。
func Print2[T1, T2 any](s1 []T1, s2 []T2) { ... }
```

これを次のコード例と比較してみてください。

```
// Print2Same は一つの型パラメータと二つの非型パラメータを持ちます。
func Print2Same[T any](s1 []T, s2 []T) { ... }
```

Print2 では、s1 と s2 は異なる型のスライスでも構いません。Print2Same では、s1 と s2 は同じ要素型のスライスでなければなりません。

通常のパラメータはそれぞれが独自の型を持つのと同じように、型パラメータもそれぞれが独自の制約を持てます。

```
// Stringer は、String メソッドを要求する型制約です。
// String メソッドは、値の文字列表現を返すべきです。
type Stringer interface {
    String() string
}

// Plusser は、Plus メソッドを要求する型制約です。
// Plus メソッドは引数を内部の文字列に追加して、その結果を返します。
type Plusser interface {
    Plus(string) string
}

// ConcatTo は String メソッドを持つ要素のスライスと Plus メソッドを持つ
// 要素のスライスを受け取ります。二つのスライスの要素の数は同じでなければ
// なりません。ConcatTo は s の要素をストリングへ変換し、それを対応する p の
// 要素の Plus メソッドへ渡して、結果の文字列のスライスを返します。
func ConcatTo[S Stringer, P Plusser](s []S, p []P) []string {
```

```
    r := make([]string, len(s))
    for i, v := range s {
        r[i] = p[i].Plus(v.String())
    }
    return r
}
```

単一の型を複数の非型の関数パラメータに対して使えるのと同様に、単一の制約を複数の型パラメータに対して使えます。その制約は、個々の型パラメータに別々に適用されます。

```
// Stringify2 は、異なる型の二つのスライスを文字列へ変換し、
// すべての文字列を結合した結果を返します。
func Stringify2[T1, T2 Stringer](s1 []T1, s2 []T2) string {
    r := ""
    for _, v1 := range s1 {
        r += v1.String()
    }
    for _, v2 := range s2 {
        r += v2.String()
    }
    return r
}
```

1.6.8　ジェネリック型

ジェネリック関数だけではなく、もっと多くのものが欲しです。つまり、ジェネリック型も欲しいわけです。型パラメータを受け取るように型を拡張することを提案します。

```
// Vector は、任意の要素型のスライスに対する名前です。
type Vector[T any] []T
```

ある型の型パラメータは、関数の型パラメータと同じようなものです。

型定義内では、型パラメータは他の型のように使えます。

ジェネリック型を使うためには、型引数を提供しなければなりません。これは、**インスタンス化**（*instantiation*）と呼ばれます[*10]。型引数は角括弧（`[]`）内に書きます。型パラメータに対する型引数を提供することで型をインスタンス化した場合、型定義内の型パラメータを使っている箇所が対応する型引数で置換された型が生成されます。

[*10] 訳注：Java の場合、ジェネリック型に対して特定の型で型引数を指定した型は、**パラメータ化された型**（*parameterized type*）と呼ばれます。

```
// v は int 値の Vector です。
//
// これは、"Vector[int]"が正当な識別子であったとして、次のように書くのに
// 似ています。
//   type "Vector[int]" []int
//   var v "Vector[int]"
// Vector[int}を使っているところはすべてが、同じ"Vector[int]"型を参照
// します。
var v Vector[int]
```

ジェネリック型はメソッドを持てます。メソッドのレシーバ型は、レシーバーの型定義で宣言された数と同じ型パラメータを宣言しなければなりません。それらは、制約なしで宣言されます。

```
// Push はベクターの最後に値を追加します。
func (v *Vector[T]) Push(x T) { *v = append(*v, x) }
```

メソッド宣言で列挙される型パラメータは、型宣言での型パラメータと同じ名前である必要はありません。メソッドで型パラメータが使われない場合、_ を使えます。

ジェネリック型は、ある型が普通に自分自身を参照できるところでは、ジェネリック型自身を参照できます。しかし、その場合、型引数は同じ順序で列挙された型パラメータでなければなりません。この制限により、無限に再帰的に型のインスタンス化が行われるのを防ぎます[*11]。

```
// List は、T 型の値のリンクリスです。
type List[T any] struct {
    next *List[T] // List[T] へのこの参照は OK
    val  T
}


// この型は不正。
type P[T1, T2 any] struct {
    F *P[T2, T1] // 不正; [T1, T2] でなければならない。
}
```

この制限は、直接的な参照と間接的な参照の両方に適用されます。

```
// ListHead は、リンクリストのヘッド。
type ListHead[T any] struct {
    head *ListElement[T]
```

[*11] 訳注：2021 年 4 月 27 日時点での tip 版と go2go 版ではコンパイルできてしまします。

```
}

// ListElement は、ヘッドを持つリンクリスト内の要素。
// 各要素は、ヘッドへ逆参照している。
type ListElement[T any] struct {
    next *ListElement[T]
    val  T
    // ここで ListHead[T]を使うのは OK。
    // ListHead[T]は ListHead[T]を参照している ListElemnt[T]を
    // 参照している。
    // ListHead[int] を使うのは OK ではない。なぜなら、
    // ListHead[T] は ListHead[int] への間接的な参照を持つので。
    head *ListHead[T]
}
```

（注意：人々がどのようにコードを書きたいかに対する理解が深まれば、異なる型引数を使う場合を許すために、この規則を緩める可能性はあります。）

ジェネリック型の型パラメータは any 以外の制約を持てます。

```
// StringableVector は何らかの型のスライスであり、その型は
// String メソッドを持っていなければなりません。
type StringableVector[T Stringer] []T

func (s StringableVector[T]) String() string {
    var sb strings.Builder
    for i, v := range s {
        if i > 0 {
            sb.WriteString(", ")
        }
        // v は T 型であり、T の制約は Stringer なので、v.String を
        // ここで呼び出すのは OK。
        sb.WriteString(v.String())
    }
    return sb.String()
}
```

1.6.9　メソッドは追加の型引数を受け取れない

ジェネリック型のメソッドは、その型パラメータを使えますが、メソッド自身が追加の型パラメータを持つことはできません[*12]。メソッドに型引数を追加するのが有益な場合、適切にパラメータ化されたトップレベルの関数を書かなければなりません。

これに関するさらなる議論は、問題点の中（46ページ）で行っています。

[*12] 訳注：Java でのジェネリックスメソッドに相当するものが書けないということです。

1.6.10　演算子

今まで説明してきたように、制約としてインタフェース型を使っています。インタフェース型はメソッドの集まりを提供して、他は何も提供していません。つまり、ここまで説明してきた事柄では、ジェネリック関数が型パラメータの値でできることは、すべての型に許されている操作に加えて、メソッドを呼び出すことだけです。

しかし、メソッド呼び出しだけでは、表現したいことのすべてに対して十分ではありません。値のスライスから最小値の要素を返す次の単純な関数を考えてみてください。ここで、スライスは空ではないと想定しています。

```
// この関数は不正です。
func Smallest[T any](s []T) T {
    r := s[0] // スライスが空ならパニック
    for _, v := range s[1:] {
        if v < r { // 不正
            r = v
        }
    }
    return r
}
```

妥当なジェネリック実装であれば、この関数が書けるべきです。問題は、式 `v < r` です。これは、`T` が < 演算子をサポートしていると想定していますが、`T` に対する制約は単に `any` です。`any` 制約であるので関数 `Smallest` は、すべての型で利用できる操作だけが使えます。しかし、すべての Go の型が < をサポートしているわけではありません。あいにく、< はメソッドではないので、< を許す制約（インタフェース型）を書く明らかな方法はありません。

< をサポートする型だけを受け付ける制約を記述する方法が必要です。それを行うためには、後で述べる二つの例外は別として、言語が定義しているすべての算術演算子、比較演算子、論理演算子は、言語が事前宣言している型か、それらの事前宣言型の一つを基底型として持つ定義型とのみ一緒に使えます。すなわち、< 演算子は、`int` や `float64` といった事前宣言型か、基底型が事前宣言型の一つである定義型とだけで使えます。Go は、コンポジット型あるいは任意の定義型と一緒に < を使うことは許していません。

これは、< のための制約を書くことを試みるよりも、逆の方法を取れることを意味します。すなわち、制約がどの演算子をサポートするのかと記述する代わりに、制約がどの（基底）型を受け付けるのかを記述できます。

制約における型リスト

制約として使われるインタフェース型は、型引数として使える型を明示的にリストできます。これは、`type` 予約語とそれに続くカンマ（`,`）で区切られた型リスト（*type list*）を使って行います。たとえば、次の通りです。

```
// SignedInteger は、すべての符号付き整数型を許す型制約です。
type SignedInteger interface {
    type int, int8, int16, int32, int64
}
```

型引数が、リストされた型の一つでなければならないと、`SignedInteger` 制約は述べています。正確には、型引数あるいは型引数の基底型は、リストされた型の一つと同じなければなりません。それは、`SignedInteger` はリストされた整数型を受付、そして、リストされた整数型の一つを基底型として定義されたすべての型も受け付けることを意味します。

ジェネリック関数がこれらの制約の一つを持つ型パラメータを使う場合、リストされた型のすべてで許されていることを行なえます。それは、`<` や `<-` といった演算子、`range` ループ、あるいは、一般的な言語での構造を意味します。関数が制約にリストされた個々の型を使ってコンパイルできれば、その使い方は許されています。

制約は、型リストを一つしか持てません。

前述の `Smallest` のコード例に対しては、次ような制約を使うこともできます。

```
package constraints

// Ordered は、順序付けされた型と一致する型制約です。
// 順序付けされた型は、<、<=、>、>=の演算子をサポートしている型です。
type Ordered interface {
    type int, int8, int16, int32, int64,
        uint, uint8, uint16, uint32, uint64, uintptr,
        float32, float64,
        string
}
```

実際面では、この制約は新たな標準ライブラリパッケージである `constraints` で定義されて公開されるでしょうから、関数と型の定義で使えるようになります。

この制約でもって、次の関数を書くことができ、今度は正当な関数です。

```
// Smallest は、スライス内の最小の要素を返します。
// スライスが空ならパニックします。
func Smallest[T constraints.Ordered](s []T) T {
    r := s[0] // スライスが空ならパニック
    for _, v := range s[1:] {
```

```
            if v < r {
                r = v
            }
        }
        return r
    }
```

制約における比較可能な型

言語で事前宣言された型でだけ使える操作に関する規則に、二つの例外があると前述しました。それらの例外は、== と != であり、構造体、配列、インタフェース型に対して許されています。この二つの演算子は有益であり、あらゆる比較可能な型を受け付ける制約を書けるようになっていて欲しいわけです。

書けるようにするために、新たな事前宣言された型制約である comparable を導入します。comparable 制約を持つ型パラメータは、比較可能な型を型引数として受け付けます。その型パラメータの値で == と != を使うことを許します。

たとえば、次の関数はあらゆる比較可能な型でインスタンス化できます。

```
// Index は s 内の s のインデックスを返します。見つからなければ-1 を返します。
func Index[T comparable](s []T, x T) int {
    for i, v := range s {
        // v と x は T 型であり、T 型は comparable 制約を持っているので、
        // ==を使えます。
        if v == x {
            return i
        }
    }
    return -1
}
```

すべての制約と同様に comparable はインタフェース型なので、制約として使われる別のインタフェース型に埋め込めます。

```
// ComparableHasher は、Hash メソッドを持つすべての比較可能な型と一致する
// 型制約です。
type ComparableHasher interface {
    comparable
    Hash() uintptr
}
```

ComparableHasher 制約は、比較可能で Hash() uintptr メソッドも持つ型によって実装されます。制約として ComparableHasher を使うジェネリック関数は、その型の値を比較したり、Hash メソッドを呼び出したりできます。

どのような型も満足できない制約を作るために、comparable を使えます。

```
// ImpossibleConstraint は、満足させられる型がない型制約です。
// なぜなら、スライス型は比較可能ではないからです。
type ImpossibleConstraint interface {
    comparable
    type []int
}
```

これ自身は誤りではありませんが、このような制約を使う型パラメータをインスタンス化する方法はもちろんありません。

インタフェース型における型リスト

型リストを持つインタフェース型は、型パラメータに対する制約としてのみ使えます。通常のインタフェース型としては使えません。同じことが、事前宣言されたインタフェース型である `comparable` にも適用されます。

この制限は、Go 言語の将来のバージョンでは取り除かれるかもしれません。`nil` 値を持てますが、型リストを持つインタフェース型は合算型（*sum type*）[*13] の形式として有益かもしれません。

型引数もしくはその基底型が型リストの中にあれば、型引数はその型リストを持つ型制約を満足します。将来の Go 言語のバージョンで、型制約以外として型リストを持つインタフェース型を許すとしたなら、その規則は、そのようなインタフェースを型制約および合算型として使えるようにするでしょう。定義型のリストを使うことは、正確に一致する定義型だけがインタフェースを満足することを意味するようになり、つまり、正確に一致する型だけが合算型に代入できます。事前宣言型および（あるいは）型リテラルのリストを使うことで、それらの型の一つとして定義された型であれば、そのインタフェースを実装し、その合算型に代入できるようになります。事前宣言型あるいは型リテラルだけを受け付けて、それらの型として定義された型を拒否する合算型を書く方法はないでしょう。この制限は、合算型を使いたい人がいる場合には受け入れられるでしょう。

1.6.11　相互参照している型パラメータ

型パラメータのリスト内では、型制約は、同じリストの後方で宣言されたものを含め、他の型パラメータを参照できます。（型パラメータのスコープは、型パラメータのリストの開始から始まり、関数あるいは型宣言の終わりまでです。）

たとえば、グラフを扱える汎用アルゴリズムを含む汎用グラフパッケージを考えてみてください。そのアルゴリズムは、`Node` と `Edge` の二つの型を使います。`Node` は、`Edges() []Edge` というメソッドを持ちます。`Edge` は、`Nodes() (Node, Node)` というメソッドを持ちます。グラフは、`[]Node` として表現できます。

[*13] 訳注：https://en.wikipedia.org/wiki/Tagged_union

この単純な表現で、最短パスを見つけるといったグラフアルゴリズムを実装できます。

```
package graph

// NodeConstraint は、グラフのノード用の型制約です。
// それは、この Node に接続されている Edge を返す Edges メソッドを
// 持っていなければなりません。
type NodeConstraint[Edge any] interface {
    Edges() []Edge
}

// EdgeConstraint は、グラフのエッジ用の型制約です。
// それは、このエッジを接続している二つの Node を返す
// Nodes メソッドを持っていなければなりません。
type EdgeConstraint[Node any] interface {
    Nodes() (from, to Node)
}

// Graph は、ノードとエッジから構成されるグラフです。
type Graph[Node NodeConstraint[Edge],
    Edge EdgeConstraint[Node]] struct {
    ...
}

// New は、与えられたノードのリストの新たなグラフを返します。
func New[Node NodeConstraint[Edge],
    Edge EdgeConstraint[Node]](
    nodes []Node,
) *Graph[Node, Edge] {
    ...
}

// ShortestPath は、二つのノードの最短パスをエッジのリストとして返します。
func (g *Graph[Node, Edge]) ShortestPath(from, to Node) []Edge {
    ...
}
```

このコードには、多くの型引数とインスタンス化が含まれています。Graph における Node に対する制約では、型制約 NodeConstraint に渡されている Edge が Grpah の二つ目の型パラメータです。型パラメータ Edge で NodeConstraint をインスタンス化しているので、Node は Edge のスライスを返す Edges メソッドを持っていなければなりません。それが私達が欲しいものです。同じことが Edge に対する制約にも適用されます。そして、同じ型パラメータと制約が New 関数で繰り返されています。これが単純だとは言いませんが、可能であるとは言えます。

一見すると、これはインタフェースの普通の使い方のように見えるかもしれないことに注

意してください。Node と Edge は特定のメソッドを持ったインタフェースではない型です。graph.Graph を使うためには、Node と Edge に対して使われている型引数は、ある種のパターンに沿ったメソッドを定義しなければなりませんが、これにはインタフェース型を実際に使う必要はありません。特に、そのようなメソッドはインタフェース型を返しません。

たとえば、他のパッケージにある次の型定義を考えてみてください。

```
// Vertex はグラフ内のノードです。
type Vertex struct { ... }

// Edges は、v に接続されているエッジを返します。
func (v *Vertex) Edges() []*FromTo { ... }

// FromtTo は、グラフ内のエッジです。
type FromTo struct { ... }

// Nodes は ft に接続されているノードを返します。
func (ft *FromTo) Nodes() (*Vertex, *Vertex) { ... }
```

ここには、インタフェース型はありませんが、型引数として *Vertex と *FromTo を使って graph.Graph をインスタンス化できます。

```
var g = graph.New[*Vertex, *FromTo]([]*Vertex{ ... })
```

*Vertex と *FromTo はインタフェース型ではありませんが、一緒に使われた場合、それらは graph.Graph の制約を実装しているメソッドを定義しています。Vertex や FromTo を graph.New へ渡せないことに注意してください。なぜなら、Vertex と FromTo は、制約を実装していないからです。Edges メソッドと Nodes は、ポインタ型である *Vertex と *FromTo に対して定義されています。一方、Vertex 型と FromTo 型は何もメソッドを持っていません。

制約としてジェネリックインタフェース型を使う場合、最初に型パラメータのリストで提供されている型を型引数でインスタンス化して、それから対応する型引数をインスタンス化された制約に対して比較します。この例では、graph.New への Node 型引数は NodeConstraint[Edge] 制約を持っています。 *Vertex を Node 型引数、*FromTo を Edge 型引数として、graph.New を呼び出す場合、Node に対する制約を検査するために、コンパイラは型引数 *FromTo でもって NodeConstraint をインスタンス化します。それにより、インスタンス化された制約が生成され、この場合 Node は Edges() []*FromTo というメソッド持つという要件となり、コンパイラは *Vertex がその制約を満足するかを検証します。

Node と Edge はインタフェース型でインスタンス化される必要はありませんが、インタフェース型を使うこともできます。

```
type NodeInterface interface {
    Edges() []EdgeInterface
}
type EdgeInterface interface {
    Nodes() (NodeInterface, NodeInterface)
}
```

NodeInterface 型と EdgeInterface 型は型制約を実装しており、それらで graph.Graph をインスタンス化できます。このような方法で型をインスタンス化する大きな理由はありませんが、許されています。

型パラメータが他の型パラメータを参照できることは重要な点を示しています。つまり、コンパイラが検査できる方法で、複数の型引数が相互に参照しているジェネリックコードをインスタンス化できることが、Go へジェネリックスを追加するどのような試みでも必須要件だということです。

1.6.12 型推論

多くの場合、型引数の一部あるいは全部を明示的に書かなくてもよいように**型推論**（*type inference*）を使えます。非型引数の型から型引数を導出するために、関数呼び出しに対しては**関数引数型推論**（*function argument type inference*）を使えます。既知の型引数から未知の型引数を導出するために**制約型推論**（*constraint type inference*）を使えます。

上記の例では、ジェネリック関数あるいはジェネリック型をインスタンス化する場合、すべての型パラメータに対する型引数を常に指定していました。指定されていない型引数が推論できる場合、型引数の一部だけの指定も許されますし、型引数の全部の指定の省略も許されます。型引数の一部だけが渡された場合、それらは、リスト内の最初の方からの型パタラメータに対する引数です。

たとえば、次の関数を見てください。

```
func Map[F, T any](s []F, f func(F) T) []T { ... }
```

この関数は、次に示すさまざまな方法で呼び出せます。（型推論がどのように行われるかの詳細は後で説明します。この例は、型引数の完全ではないリストがどのように処理されるかを示すためのものです。）

```
var s []int
f := func(i int) int64 { return int64(i) }
var r []int64
// 二つの型引数を明示的に指定する。
r = Map[int, int64](s, f)

// F に対する最初の型引数だけを指定し、T は推論させる。
```

```
r = Map[int](s, f)

// 型引数は何も指定せずに、二つとも推論させる。
r = Map(s, f)
```

ジェネリック関数あるいはジェネリック型が、すべての型引数を指定されずに使われた場合、指定されていない型引数のどれかが推論できなければエラーになります。

（注意：型推論は、便利な機能です。それは重要な機能と考えますが、ジェネリックスのデザインに何も機能性を追加せず、使うのが便利なだけです。最初の実装から型推論を取り除いて、それが必要に思えるかを判断することも可能だったでしょう。とはいえ、この機能は追加の構文は必要としませんし、コードを読みやすくします。）

型単一化

型推論は、**型単一化**（*type unification*）に基づいています。型単一化は二つの型に適用され、その二つの型の両方もしくは片方が型パラメータであるか、あるいは型パラメータを含む型です。

型単一化は、二つの型の構造を比較します。型パラメータを無視した構造は同一ではなくてはならず、型パラメータ以外の型は等価でなければなりません。片方の型内の型パラメータは、もう片方の型内のすべての完全なサブタイプと一致しているかもしれません。構造が異なったり、型パラメータ以外の型が等価でなければ、型単一化は失敗します。成功した型単一化は、他の型（それ自身は型パラメータか、型パラメータを含んでいてもよい）を持つ型パラメータの関連付けのリストを生成します。

型単一化に関して、型パラメータを含まない二つの型は、それらが同一（*identical*）[*14] であるか、チャネルの方向を無視すれば同一であるチャネル型か、あるいは基底型が等価であれば、等価です。型推論の処理中では型が同一ではないことは許されています。なぜなら、推論が成功してもまだ制約を検査し、そして、関数引数が推論された型へ代入可能か検査するからです。

たとえば、`T1` と `T2` が型パラメータであり、`[]map[int]bool` は次のどれかで単一化できます。

- `[]map[int]bool`
- `T1`（`T1` は `[]map[int]bool` に一致）
- `[]T1`（`T1` は `map[int]bool` に一致）
- `[]map[T1]T2`（`T1` は `int` に一致し、`T2` は `bool` に一致）

（これで全部ではなく、他にも可能な単一化があります。）

一方で、`[]map[int]bool` は、次のいずれでも単一化できません。

[*14] https://golang.org/ref/spec/#Type_identity

- `int`
- `struct{}`
- `[]struct{}`
- `[]map[T1]string`

（もちろん、これで全部ではありません。単一化できない型は無限にあります。）

一般に、両方の立場の型パラメータを持てるので、場合によっては、たとえば、`T1` を `T2` あるいは `[]T2` に関連付けするかもしれません。

関数引数型推論

関数引数型推論（*function argument type inference*）は、非型引数から型引数を推論するために関数呼び出しで使われます。関数引数型推論は、型がインスタンス化される場合には使われませんし、関数がインタンス化されるけど呼ばれない場合にも使われません[*15]。

どのように行われるかを知るために、単純な `Print` 関数[*16] の呼び出し例に戻ってみましょう。

```
Print[int]([]int{1, 2, 3})
```

この関数呼び出しでの型引数 `int` は、非型引数の型から推論できます。

推論できる唯一の型引数は、関数の（非型）入力パラメータの型だけで使われている型引数です。関数の結果パラメータ型だけで使われている、あるいは、関数の本体だけで使われている型パラメータの場合、関数引数型推論はそれらの型引数を推論するのに使えません。

関数型引数を推論するには、関数の呼び出し引数の型を、関数の非型パラメータの型で単一化します。呼び出しもと側では、実際の（非型）引数の型のリストを持っており、`Print` のコード例では単に `[]int` です。関数側では、関数の非型パラメターの型のリストは、`Print` では `[]T` です。これらの両方のリストで、関数側が型パラメータを使っていない部分に対応する引数を破棄します[*17]。それから、残りの引数型に型単一化を適用しなければなりません。

関数引数型推論は、2 パスのアルゴリズムです。最初のパスでは、呼び出しもと側での型付けなし定数および関数定義でのそれらに対応する型を無視します[*18]。2 パスを使うので、場合によっては後者の引数が、型付けなし定数の型を決められます。

それらのリスト内の対応する型を単一化します。これは、関数型の型パラメータを呼び出しもと側の型との関連付けを与えてくれます。同じ型パラメータが関数側で二回以上現れるなら、それは呼び出しもと側の複数の引数型に一致します。それらの呼び出しもとの型が等価でなければ、エラーが報告されます。

[*15] 訳注：どういう意味？
[*16] 訳注：`Print` 関数の定義は、`func Print[T any](s []T) { ... }`です（4ページ）。
[*17] 訳注：`Print` メソッドの呼び出し例では、破棄される引数はありません。
[*18] 訳注：`Print` メソッドの呼び出し例では、定数は渡されていないので何も無視されません。

最初のパスの後、呼び出しもと側の型付けなし定数を検査します。型付けされていない定数がない、あるいは対応する関数型での型パラメータが他の入力型と一致していれば、型単一化は完了です[*19]。

そうでなければ、二つ目のパスでは、まだ設定されていない対応する関数型の型付けなし定数に対して、通常の方法で型付けなし定数のデフォルトの型[*20] を決定します[*21]。それから、残っている型を、今度は型付けなし定数以外と単一化します。

次のコード例を見てください。

```
    s1 := []int{1, 2, 3}
    Print(s1)
```

`[]int` を `[]T` と比較し、`T` を `int` と一致させて終わりです。一つしかない型パラメータ `T` は `int` であり、`Print` の呼び出しを、実際は `Print[int]` の呼び出しと推論します。

複雑な例として、次のコード例を考えてみてください。

```
// Map はスライス s の各要素に関数 f を呼び出して、
// 結果の新たなスライスを返します。
func Map[F, T any](s []F, f func(F) T) []T {
    r := make([]T, len(s))
    for i, v := range s {
        r[i] = f(v)
    }
    return r
}
```

二つの型パラメータである `F` と `T` は入力パラメータでどちらも使われており、関数引数型推論を行えます。次の呼び出しを考えてみてください。

```
    strs := Map([]int{1, 2, 3}, strconv.Itoa)
```

`[]int` を `[]F` で単一化し、`F` を `int` に一致させます。`func(int) string` である `strconv.Itoa` の型を `func(F) T` で単一化し、`F` を `int`、`T` を `string` に一致させます。型パラメータ `F` は二回とも `int` と一致しています。単一化が成功したので、`Map` と書かれた呼び出しは `Map[int, string]` の呼び出しです。

型付けなし定数の規則の適用を理解するために、次のコード例を見てください。

[*19] 訳注：`Print` メソッドの呼び出し例では、ここで完了です。

[*20] 訳注：型付けなし定数は 6 種類あり、デフォルトの型は、型付けなし整数は `int`、型付けなし浮動小数点数は `float64`、型付けなしルーンは `rune`、型付けなし複素数は `complex128`、型付けなしブーリンは `bool`、型付けなし文字列は `string` です。

[*21] `https://golang.org/ref/spec#Constants)`

```
// NewPari は、同じ型の値の組を返します。
func NewPair[F any](f1, f2 F) *Pair[F] { ... }
```

`NewPair(1, 2)` の呼び出しでは、両方の引数が型付けなし定数であり、最初のパスでは両方が無視されます。そうすると単一化するものはありません。最初のパスの後、二つの型付けなし定数をまだ持っています。両方がそれらのデフォルトの型である `int` に設定されます。型単一化の 2 回目のパスでは、`F` を `int` に単一化するので、最終的な呼び出しは `NewPair[int](1, 2)` です。

`NewPair(1, int64(2))` の呼び出しでは、一つ目の引数は型付けなし定数であり、一つ目のパスではそれを無視します。それから `int64` を `F` で単一化します。この時点で型付けなし定数に対応する型パラメータは決定するので、最終的な呼び出しは、`NewPair[int64](1, int64(2))` です。

`NewPair(1, 2.5)` の呼び出しでは、両方の引数は型付けなし定数であり、二つ目のパスへ進みます。今度は、最初の定数は `int` に、二つ目の定数は `float64` に設定します。それから、`F` を `int` と `float64` の両方で単一化しようとするので、単一化は失敗し、コンパイルエラーが報告されます。

最初に述べたように、関数引数型推論は制約に関係なく行われます。最初に、関数に使われる型引数を決めるために関数型引数推論を使います。そして、それがうまくいけば、それらの型引数が（指定されていれば）制約を実装しているか検査します。

関数引数型推論がうまくいった後は、コンパイラは、関数呼び出しに関して引数がパラメータに代入できるかをさらに検査しなければならないことに注意してください[*22]。

制約型推論

制約型推論（*constraint type inference*）は、型パラメータ制約に基づいて他の型引数から型引数を推論することを許します。制約型推論は、関数が何らかの他の型パラメータの要素に対する型名を持ちたい場合や、関数が何らかの他の型パラメータに基づいている型へ制約を適用したい場合に役立ちます。

型パラメータが制約を持ち、その制約内の型リスト内の一つの型と正確に一致する場合にだけ、制約型推論は型を推論できます。このような制約を**構造的制約**（*structural constraint*）と呼びます。なぜなら、その型リストは型パラメータの構造を記述しているからです。構造的制約は型リストに加えてメソッドを持っていても構いませんが、メソッドは制約型推論では無視されます。制約型推論が役立つためには、制約型は一つ以上の型パラメータを普通は参照しています。

制約型推論は、関数引数型推論の後に適用されます。型引数がまだ分かっていない型パラメータが少なくとも一つ以上ある場合にだけ適用されます。

[*22] 訳注：`NewPair(1, int64(2))` の呼び出しでは、`F` が `int64` となりますが、型付けなし定数である `1` はそれに代入可能です。

ここで述べるアルゴリズムは複雑に思わるかもしれませんが、典型的な具体的例に対しては、制約型推論が何を導き出すかを理解するのは簡単です。アルゴリズムの説明の後で、コード例を示します。

型パラメータから型引数へのマッピングを作成することから始めます。そのマッピングを、型引数がすでに既知のすべての型パラメータで初期化します。

構造的制約を持つ個々の型パラメータに対して、その型パラメータを制約の型リスト内の一つの型で単一化します。これは、型パラメータをその制約と関連付けるという効果になります。その結果を保持しているマッピングへ追加します。単一化が型パラメータの関連性を見つけたら、それもマッピングへ追加します。一つの型パラメータに対して複数の関連付けがある場合、一つのマッピングエントリを生成するために、そのような関連性を単一化します。型パラメータが他のパラメータに直接関連付けされている場合、つまり、両方の型パラメータが同一の型に一致しなければならない場合、個々のパラメータの関連付けをまとめて単一化します。これらのさまざまな単一化のどれかが失敗した場合、制約型推論は失敗します。

すべての型パラメータを構造的制約でマージした後、さまざまな型パラメータから型（他の型パラメータか、もしくは他の型パラメータを含む型）へのマッピングを持ちます。何も型パラメータを含まない既知の型引数 `A` へマップされる型引数 `T` を探して、処理を続けます。マッピング内で型引数に `T` が現れた場所では、`T` を `A` で置換します。すべての型パラメータが置換されるまでこの処理を繰り返します。

要素制約のコード例

制約型推論が役立つ場合のコード例として、数値のスライスである**定義型**（*defined type*）[*23] を受け取り、各要素の数値を倍にした同じ定義型のインスタンスを返す関数を考えてみます。

定義型[*24] の要件を無視したら、型リストを使って次のような関数を書くのは容易です。

```
// Double は、s のすべての要素を倍にした新たなスライスを返します。
func Double[E constraints.Number](s []E) []E {
    r := make([]E, len(s))
    for i, v := range s {
        r[i] = v + v
    }
    return r
}
```

しかし、この定義では、定義されたスライス型で関数を呼び出すと、結果はその定義型にはなりません。

[*23] 訳注：`type` 宣言で定義された型を指します。Go 1.9 で**型エイリアス**（*type alias*）が導入される前は**名前付き型**（*named type*）と呼ばれていました。

[*24] `https://golang.org/ref/spec#Type_definitions`

```
// MySlice は int のスライス。
type MySlice []int

// V1 の型は、MySlice ではなく、[]int です。
// ここでは、関数引数型推論を使っていますが、制約型推論は使っていません。
var V1 = Double(MySlice{1})
```

新たな型パラメータを導入することで行いたいことが行なえます。

```
// SC は、何らかの型 E のスライスであるべき型を制約します。
type SC[E any] interface {
    type []E
}

// DoubleDefined は、s の要素を倍にした新たなスライスを返し、
// その新たなスライスは s と同じ型でもあります。
func DoubleDefined[S SC[E], E constraints.Number](s S) S {
    // 上記では []E を渡していましたが、
    // ここでは S を make に渡していることに注意してください。
    r := make(S, len(s))
    for i, v := range s {
        r[i] = v + v
    }
    return r
}
```

これで、明示的な型引数を使えば、正しい型が得られます。

```
// V2 の型は MySlice。
var V2 = DoubleDefined[MySlice, int](MySlice{1})
```

関数引数型推論はそれ自身では、この場合に型引数を推論するには十分ではありません。なぜなら、型パラメータ E は入力パラメータでは使われていないからです。関数引数型推論と制約型推論を組み合わせることでうまくいきます。

```
// The type of V3 will be MySlice.
var V3 = DoubleDefined(MySlice{1})
```

最初に関数引数型推論を適用します。引数の型が MySlice と分かっています。関数引数型推論は、型パラメータ S を MySlice に一致させます。

それから制約型推論に進みます。一つの型引数が S であることが分かっています。型引数 S は構造的型制約を持っていることが分かります。

既知の型引数のマッピングを作成します。

```
{S -> MySlice}
```

それから、個々の型パラメータを単一の型をリストしている構造的制約で単一化します。この場合、構造的制約は単一の型 `[]E` を持つ `SC[E]` であるので、`S` を `[]E` で単一化します。`S` に対するマッピングはすでに持っているので、`[]E` を `MySlice` で単一化します。`MySlice` は `[]int` と定義されているので、`E` を `int` に関連付けます。これで、次のマッピングを持っています。

```
{S -> MySlice, E -> int}
```

それから、`E` を `int` で置き換えますが、何も変更しません。そして、終わりです。`DoubleDefined` の呼び出しに対する型引数は、`[MySlice, int]` です。

この例は、何らかの他の型パラメータの要素に対する型名を設定するために制約型推論を使える方法を示しています。この場合、`S` の要素型を `E` として命名して、それから `E` にさらに制約を適用できます。今回の場合は、それが数値であることを要求しています。

ポインタメソッドの例

文字列に基づいて値を初期化する `Set(string)` メソッドを持つ `T` 型を期待する次の関数を考えてみてください。

```
// Setter は、文字列から値を設定する\texttt{Set}メソッドを型が実装すること
// を要求する型制約です。
type Setter interface {
    Set(string)
}

// FromStrings は文字列のスライスを受け取り、Set メソッドを呼び出して、
// 個々の返された値を設定した T のスライスを返します。
//
// T は結果パラメータだけに使われているので、関数引数型推論は、
// この関数呼び出しでは適用できないことに注意してください。
func FromStrings[T Setter](s []string) []T {
    result := make([]T, len(s))
    for i, v := range s {
        result[i].Set(v)
    }
    return result
}
```

では、呼び出しのコードを見ていきましょう（次のコード例は不正です）。

```
// Settable は、文字列から設定できる整数型です。
type Settable int

// Set は、文字列から*p の値を設定します。
func (p *Settable) Set(s string) {
    i, _ := strconv.Atoi(s) // 現実のコードはエラーを無視すべきではない
    *p = Settable(i)
}

func F() {
    // 不正
    nums := FromStrings[Settable]([]string{"1", "2"})
    // ここで、nums は []Settable{1, 2}であって欲しい。
    ...
}
```

目標は、[]Settable 型のスライスを得るために FromStrings を使うことです。あいにく、このコード例は不正であり、コンパイルされません。

問題は、FromStrings が Set(string) メソッドを持つ型を要求していることです。関数 F は Settable で FromStrings をインスタンス化しようとしていますが、Settable は Set メソッドを持っていません。Set メソッドを持っている型は、*Settable です。

では、代わりに *Settable を使って F を書き直してみます。

```
func F() {
    // コンパイルされますが、望むようには動作しません。
    // 実行時に Set メソッドが呼び出された時にパニックになります。
    nums := FromStrings[*Settable]([]string{"1", "2"})
    ...
}
```

これはコンパイルされますが、あいにく実行時にパニックになります。問題は、FromStrings が []T 型のスライスを生成することです。*Settable でインスタンス化された場合、[]*Settable 型のスライスを意味します。FromStrings が result[i].Set(v) を呼び出した場合、result[i] に保存されているポインタに対して Set メソッドを呼び出します。そのポインタは nil です。Settable.Set メソッドは nil レシーバに対して呼び出され、nil 参照エラーによりパニックになります。

ポインタ型である Settable は制約を実装していませんが、コードは実際にはポインタ型ではない Settable を使いたいのです。引数として Settable を受け取るがポインタメソッドを呼び出せる FromStrings を書く方法が必要です。繰り返しますが、Settable は Set メソッドを持っていないので使えません。そして、Settable 型のスライスを生成できないので *Settable も使えません。

できることは、両方の型を渡すことです。

```
// Setter2 は、型が文字列からアタを設定する Set メソッドを実装していて、
// その型がそれ自身の型パラメータへのポインタであることを要求する型制約です。
type Setter2[B any] interface {
    Set(string)
    type *B
}

// FromStrings2 は文字列のスライスを受け取り、Set メソッドを呼び出して、
// 個々の返された値を設定した\texttt{T}のスライスを返します。
//
// T 型のスライスを返すが、*T（ここでは PT）に対するメソッドを呼び出せるように
// 二つの異なる型パラメータを使います。
// Setter2 制約は、PT が T へのポインタであることを保証します。
func FromStrings2[T any, PT Setter2[T]](s []string) []T {
    result := make([]T, len(s))
    for i, v := range s {
        // &result[i] の型は*T であり、それは Setter2 の型リストに
        // あるので、それを PT へ変換できます。
        p := PT(&result[i])
        // PT は Set メソッドを持っています。
        p.Set(v)
    }
    return result
}
```

これで、次のように FromStrings2 を呼び出せます。

```
func F2() {
    // FromStrings2 は二つの型パラメータを受け取る。二つ目の型パラメータは、
    // 一つ目の型パラメータへのポインタでなければならない。
    // Settable は前述の通り。
    nums := FromStrings2[Settable, *Settable]([]string{"1", "2"})
    // これで num は、[]Settable{1, 2}。
    ...
}
```

この方法は期待通りに機能しますが、型引数として Settable を繰り返さなければならないのは、ぎこちないです。幸い、制約型推論はぎこちなさを低減します。制約型推論を使って次のように書き直せます。

```
func F3() {
    // ここでは、一つの型引数を渡すだけです。
    nums := FromStrings2[Settable]([]string{"1", "2"})
    // これで num は、[]Settable{1, 2}。
    ...
```

```
}
```

型引数 Settable を渡すのを避ける方法はありません。しかし、その型引数が渡されたことで、制約型推論は、型パラメータ PT に対して型引数 *Settable を推論できます。

前と同じように、既知の型引数のマッピングを作成します。

```
{T -> Settable}
```

次に、個々の型パラメータを構造的制約で単一化します。この場合、PT は Setter2[T] の一つの型である *T で単一化します。その結果、マッピングは次のようになります。

```
{T -> Settable, PT -> *T}
```

次に、全体の T を Settable で置き換えると、次のようになります。

```
{T -> Settable, PT -> *Settable}
```

これは何も変更しないので、これで終了です。両方の型引数が分かりました。

この例は、何らかの他の型パラメータに基づく型へ制約を適用するために制約型推論をどのように使えるかを示しています。この場合、*T である TP は Set メソッドを持っていなければならないと述べています。それを、呼び出しもとが明示的に *T へ言及せずに行えます。

制約型推論後に制約を適用する

制約型推論が制約に基づいて型引数を推論するために使われる場合であっても、型引数が決定した後に制約をまだ検査しなければなりません。

上記の FromStrings2 のコード例では、Setter2 制約に基づく PT に対する型引数を導き出せました。しかし、そうする際に、型リストを調べただけであり、メソッドは調べていません。たとえ、制約型推論が成功したとして、制約を満足するメソッドが存在するかをさらに検証しなければなりません。

たとえば、次の不正なコードを考えてみてください。

```
// Unsettable は Set メソッドを持たない型です。
type Unsettable int

func F4() {
    // この呼び出しは不正です。
    nums := FromString2[Unsettable]([]string{"1", "2"})
    ...
}
```

この呼び出しが行われた場合、以前と同様に制約型推論を適用します。それは、以前と同

様に成功し、型引数は [Unsettable, *Unsettable] であると推論されます。制約型推論が完了した後に、*Unsettable が制約 Setter2[Unsettable] を実装しているか検査します。*Unsettable は Set メソッドを持っていないので、制約の検査は失敗し、このコードはコンパイルされません。

1.6.13　制約内で自身を参照する型を使う

引数が型自身であるメソッドを持つ型引数を要求することがジェネリック関数にとって役立つことがあります。たとえば、比較メソッドでは自然とそうなります。（ここでは、演算子ではなくメソッドについて述べていることに注意してください。）求める値を見つけたか検査するために Equal メソッドを使う Index メソッドを書きたいとします。次のようなコードを書くでしょう。

```
// Index は s 内の e のインデックスを返します。見つからなければ、-1 を返します。
func Index[T Equaler](s []T, e T) int {
    for i, v := range s {
        if e.Equal(v) {
            return i
        }
    }
    return -1
}
```

Equaler 制約を書くためには、渡される型引数を参照できる制約を書かなければなりません。最も簡単な方法は、制約は定義型である必要がないという事実を利用して、単純にインタフェース型リテラルにできます。そうすれば、そのインタフェース型リテラルは型パラメータを参照できます。

```
// Index は s 内の e のインデックスを返します。見つからなければ、-1 を返します。
func Index[T interface { Equal(T) bool }](s []T, e T) int {
    // 前と同じ
}
```

このバージョンの Index は、次の equalInt と同様な型で使えます。

```
// equalInt は、Equaler を実装している int のバージョンです。
type equalInt int

// Equal メソッドにより equalInt は Equaler 制約を実装します。
func (a equalInt) Equal(b equalInt) bool { return a == b }

// // indexEqualInts は s 内の e のインデックスを返します。
// 見つからなければ、-1 を返します。
```

```
func indexEqualInt(s []equalInt, e equalInt) int {
    // 型引数 equalInt はここでは明瞭にするために示しています。
    // 関数型引数推論により省略できます。
    return Index[equalInt](s, e)
}
```

このコード例では、Index へ equalInt を渡した場合、equalInt が制約である interface { Equal(T) bool }を実装しているか検査しています。その制約は型パラメータを持っているので、その型パラメータである equalInt で置き換えます。その結果、interface { Equal(equalInt) bool }となります。equalInt 型はシグニチャが一致する Equal メソッドを持っており、何も問題がなく、コンパイルは成功します。

訳注：ここで示された Index 関数の定義は冗長なので、次のように Equaler 制約と Index 関数を定義できます。

```
type Equaler[T any] interface {
    Equal(o T) bool
}
func Index[T Equaler[T]](s []T, e T) int {
    // ....
}
```

1.6.14　型パラメータの値はボックス化されない

Go の現在の実装では、インタフェース値は常にポインタを保持しています。インタフェース変数へポインタではない値を入れるとその値はボックス化（*boxed*）されます。実際の値はヒープ上あるいはスタック上のどこかに保存されて、そのインタフェース値は保存された場所へのポインタを保持しています。

このデザインでは、ジェネリック型の値はボックス化されません。たとえば、前に示した FromStrings2 を見返してみます。それを Settable でインスタンス化した場合、[]Settable 型の値を返します。たとえば、次のようにコードを書けます。

```
// Settable は、文字列から値を設定できる整数型です。
type Settable int

// Set は文字列から*p の値を設定する。
func (p *Settable) Set(s string) {
    // 前と同じ
}

func F() {
```

```
    // nums の型は []Settable。
    nums := FromStrings2[Settable]([]string{"1", "2"})
    // Settable は int へ直接変換できます。
    // これは first に 1 を設定します。
    first := int(nums[0])
    ...
}
```

Settable 型でインスタンス化された FromStrings2 を呼び出した場合、[]Settable を得ます。そのスライスの要素は Settable の値であり、つまり整数です。そのスライスの要素は、ジェネリック関数により生成されて設定されても、ボックス化されません。

同様に、ジェネリック型がインスタンス化された場合、それは期待された型を構成要素として持ちます。

```
type Pair[F1, F2 any] struct {
    first  F1
    second F2
}
```

これがインスタンス化された場合、フィールドはボックス化されず、予期せぬメモリ割り当ては行われません。Pair[int, string] 型は、struct { first int; second string }に変換されます。

1.6.15　型リストの詳細

それほど重要ではないですが、注意が必要な詳細を説明するために、ここで型リストへ戻りましょう。これから説明する詳細は、追加の規則や概念ではありませんが、型リストの仕組みの結果です。

型リストとメソッドの両方がある制約

前に Setter2 で説明したように、制約は型リストとメソッドの両方を持てます。

```
// StringableSignedInteger は 1) 符号付き整数として定義され、
// 2) String メソッドを持つすべての型と一致する型制約です。
type StringableSignedInteger interface {
    type int, int8, int16, int32, int64
    String() string
}
```

この制約は、期待型がリストされた型の一つで、String() string メソッドも持っている型を許します。StringableSignedInteger 制約は明示的に int を列挙していますが、int 型自身は型引数として許されません。なぜなら、String メソッドを持っていない

からです。許される型引数の一つ例は次のように定義された MyInt です。

```
// MyInt は文字列化可能な int です。
type MyInt int

// String メソッドは mi の文字列表現を返します。
func (mi MyInt) String() string {
    return fmt.Sprintf("MyInt(%d)", mi)
}
```

型リスト内のメソッドを持つ型

型リストを使うことで、ジェネリック関数は、型リスト内のすべての型によって許されている操作（演算）を使えます。しかし、それはメソッドに対しては適用されません。型リスト内のすべての型が同じシグニチャの同じメソッドをサポートしていても、ジェネリック関数はそのメソッドを呼び出せません。ジェネリック関数は、前の節で説明したように制約内に明示的に書かれたメソッドだけを呼び出せます。

次は、同じメソッドを持つ複数の型の例です。この例は、不正です。MyInt と MyFloat の両方が String メソッドを持っていますが、ToString 関数はそのメソッドを呼び出すことを許されていません。

```
// MyInt は String メソッドを持っています。
type MyInt int

func (i MyInt) String() string {
    return strconv.Itoa(int(i))
}

// MyFloat も String メソッドを持っています。
type MyFloat float64

func (f MyFloat) String() string {
    return strconv.FormatFloat(float64(f), 'g', -1, 64)
}

// MyIntOrFloat は、MyInt か MyFloat を受け付ける型制約です。
type MyIntOrFloat interface {
    type MyInt, MyFloat
}

// ToString は値を文字列へ変換します。
// この関数は、不正です。
func ToString[T MyIntOrFloat](v T) string {
    return v.String() // 不正
```

```
}
```

String メソッドの呼び出しを許すためには、制約内に明示的に記述する必要があります。

```
// MyIntOrFloatStringer は MyInt あるいは MyFloat を受け付けて、String
// メソッドを定義しています。MyInt と MyFloat の両方が String メソッドを持って
// いることに注意してください。MyInt と MyFloat のどちらも String メソッドを
// 持っていなければ、型リストがそれらを列挙していたとしても、制約を満足しませ
// ん。制約を満足するために、型は、（あれば）型リストと一致して、かつ、（あれば）
// すべてのメソッドを実装していなければなりません。
type MyIntOrFloatStringer interface {
    type MyInt, MyFloat
    String() string
}

// ToString2 は、値を文字列へ変換します。
func ToString2[T MyIntOrFloatStringer](v T) string {
    return v.String()
}
```

この規則の理由は、型が特定のメソッドを持っているかがすぐにはっきりしていない、埋め込み型パラメータが関係している複雑な場合を簡単にするためです。

制約内のコンポジット型

制約内の型は、**型リテラル**（*type literal*）[*25] でも構いません。

```
type byteseq interface {
    type string, []byte
}
```

通常の規則が適用されます。つまり、この制約に対する型引数は、string あるいは []byte、もしくは、これらの型の一つとして定義された型です。この制約を持つジェネリック関数は、string と []byte の両方で許される操作を使えます。

byteseq 制約は、string 型あるいは []byte 型のどちらかに対しても処理できるジェネリック関数を書くのを許しています[*26]。

[*25] 訳注：既存の型から作られる型を指します。

[*26] 訳注：if len(a) == 1 の場合、append で a[0]... と書かれています。T が string の場合の挙動は、書籍『プログラミング言語 Go』には記述されていませんが、Go 1 がリリースされた時点で言語仕様に次のように記述されています。

> As a special case, append also accepts a first argument assignable to type []byte with a second argument of string type followed by .... This form appends the bytes of the string.

```
// Join は、文字列値を作成するために、その最初の引数の要素を結合します。
// sep は、結果の要素の間に入れられます。
// Join は、string 型と []bytes 型を処理します。
func Join[T byteseq](a []T, sep T) (ret T) {
    if len(a) == 0 {
        // 結果パラメータをゼロ値として使う。
        // イシュ節でのゼロ値の説明を参照してください。
        return ret
    }
    if len(a) == 1 {
        // a[0] が string か []byte であると分かっています。
        // string あるいは []byte を []byte へ append して、[]byte を生成
        // します。その []byte を []byte（何も変換なし）あるいは string へ
        // 変換できます。
        return T(append([]byte(nil), a[0]...))
    }
    // string と []byte の両方に len を呼び出せるので、
    // sep に対して len を呼び出せます。
    n := len(sep) * (len(a) - 1)
    for _, v := range a {
        // string か []byte へ len を呼び出せる別のケース。
        n += len(v)
    }

    b := make([]byte, n)
    // string あるは []byte の引数で []byte への copy を呼び出せます。
    bp := copy(b, a[0])
    for _, s := range a[1:] {
        bp += copy(b[bp:], sep)
        bp += copy(b[bp:], s)
    }
    // 上と同様に b を []byte か string へ変換できます。
    return T(b)
}
```

コンポジット型（文字列、ポインタ、配列、スライス、構造体、関数、マップ、チャネル）に対しては、追加の制限を課します。型リストに列挙されたすべての型に対して、演算子が（あれば）入力の同じ型を受け入れて、同じ結果の型を生成するなら、その操作を行えます。明確にするために、この追加の制限は、コンポジット型が型リストに列挙された場合にだけ強制されます。何らかの型パラメータ `T` に対する `var v []T` などのように、コンポジット型が型リストにない型パラメータから形成される場合には適用されません。

```
// structField は、すべてが x と名付けられたフィールドを
// 持つ構造体のリストを持つ型制約です。
type structField interface {
```

```
    type struct { a int; x int },
        struct { b int; x float64 },
        struct { c int; x uint64 }
}

// この関数は不正です。
func IncrementX[T structField](p *T) {
    v := p.x // 不正：p.x の型は、リスト内のすべての型で同じではない。
    v++
    p.x = v
}

// sliceOrMap は、スライスかマップに対する型制約です。
type sliceOrMap interface {
    type []int, map[int]int
}

// Entry は、スライスの i 番目のエントリか、キーが i のマップ内の値を返します。
// 演算子の結果が常に int なので、この関数は正当です。
func Entry[T sliceOrMap](c T, i int) int {
    // これは、スライスのインデックス操作か、マップのキー検索です。
    // どちらであっても、インデックスと結果の型は int 型です。
    return c[i]
}

// sliceOrFloatMap は、スライスかマップに対する型制約です。
type sliceOrFloatMap interface {
    type []int, map[float64]int
}

// この関数は不正です。
// このコード例では、インデックス操作の入力型は、（スライスに対しては）int か
//（マップに対しては）float64 のどちらかなので、操作は許されません。
func FloatEntry[T sliceOrFloatMap](c T) int {
    return c[1.0] // 不正：入力型は int か float64。
}
```

この制限を課すことで、ジェネリック関数内での操作の型について推論が容易になります。型リストの個々の要素に対して、何らかの操作を適用することで得られる型の集合の和集合となるような型の値といった概念を導入しなくて済みます。

（注意：開発者がどのようにコードを書きたいかに対する理解が深まれば、将来、この制限を緩和することもあり得ます。）

型リスト内の型パラメータ

制約内の型リテラルは、その制約の型パラメータを参照できます。次のコード例では、ジェネリック関数 Map は二つの型パラメータを受け取ります。最初の型パラメータは、二つ目の型パラメータのスライスを基底型に持つことを要求しています。二つ目のスライスのパラメータに対する制約はありません。

```
// SliceConstraint は、型パラメータのスライスに一致する型制約です。
type SliceConstraint[T any] interface {
    type []T
}

// Map は何らかの要素型のスライスと変換関数を受け取り、個々の要素に関数を
// 適用した結果のスライスを返します。
// Map は、たとえ定義型であっても、スライスの引数と同じ型のスライスを返します。
func Map[S SliceConstraint[E], E any](s S, f func(E) E) S {
    r := make(S, len(s))
    for i, v := range s {
        r[i] = f(v)
    }
    return r
}

// MySlice は単純な定義型です。
type MySlice []int

// DoubleMySlice は MySlice 型の値を受け取り、
// 各要素の値を倍にした値を持つ新たな MySlice の値を返します。
func DoubleMySlice(s MySlice) MySlice {
    // ここで明示的に記述されている型引数は、推論もできます。
    v := Map[MySlice, int](s, func(e int) int { return 2 * e })
    // ここで v は MySlice 型であり、[]int 型ではありません。
    return v
}
```

型制約推論（1.6.12節）の説明の初めで、これの他のコード例も示しています。

型変換

二つの型パラメータである From と To を持つ関数内では、From の制約で受け入れられるすべての型が To の制約で受け入れられるすべての型へ変換できるならば、From 型の値を To 型の値へ変換できます。どちらかの型パラメータが型リストを持っていなければ、型変換は許されません。

これは、型リストで列挙されているすべての型で許される操作をジェネリック関数が使え

るという一般規則から得られる帰結です。

たとえば、次の通りです。

```
type integer interface {
    type int, int8, int16, int32, int64,
        uint, uint8, uint16, uint32, uint64, uintptr
}

func Convert[To, From integer](from From) To {
    to := To(from)
    if From(to) != from {
        panic("conversion out of range")
    }
    return to
}
```

Go がすべての整数型を他のすべての整数型へ変換することをを許しているので、`Convert` での型変換は許されます。

型付けなし定数

関数によっては型付けなし定数を使います。型付けなし定数は、型パラメータの制約によって受け入れられるすべての型で許されているなら、型パラメータの値と一緒に使うことが許されます。

型変換と同様に、これは、型リストで列挙されているすべての型で許される操作をジェネリック関数が使えるという一般規則から得られる帰結です。

```
type integer interface {
    type int, int8, int16, int32, int64,
        uint, uint8, uint16, uint32, uint64, uintptr
}

func Add10[T integer](s []T) {
    for i, v := range s {
        s[i] = v + 10 // OK：10 はすべての整数型へ変換できます。
    }
}

// This function is INVALID.
func Add1024[T integer](s []T) {
    for i, v := range s {
        s[i] = v + 1024 // 不正：1024 は int8 と uint8 では許されていない。

    }
}
```

埋め込まれた制約内の型リスト

ある制約が別の制約を埋め込んでいる場合、最終的な制約の型制約は、関連するすべての型リストの共通集合です。複数の埋め込まれた型がある場合、共通集合は、型引数がすべての埋め込まれた型の要件を満足しなければならないという特性を維持します。

```
// Addable は + 演算子をサポートしている型です。
type Addable interface {
    type int, int8, int16, int32, int64,
        uint, uint8, uint16, uint32, uint64, uintptr,
        float32, float64, complex64, complex128,
        string
}

// Byteseq はバイトシーケンスであり、string か []byte です。
type Byteseq interface {
    type string, []byte
}

// AddableByteseq は、+ をサポートしているバイトシーケンスです。
// これは、Addable かつ Byteseq であるすべての型です。
// 言い換えれば、string 型だけです。
type AddableByteseq interface {
    Addable
    Byteseq
}
```

型リストに対する全般的な注意

制約に型を明示的に列挙することは洗練されていないように思えるかもしれませんが、呼び出し場所でどの型引数が許されるのかと、ジェネリック関数でどの操作が許されるのかの両方が明確です。

（現在、計画はありませんが）将来、言語が演算子法（*operator method*）[*27] をサポートするように変更になるなら、制約は、他の種類の演算子を扱うように演算子法を扱うでしょう。

事前定義型の数は常に決まっており、それらの型がサポートする演算子の数も常に決まっています。将来の言語の変更は、その事実を基本的に変更することはないでしょうから、制約で型を列挙することは役立ち続けます。

この方法は、すべての可能な演算子を処理すること試みていません。期待するのは、型リストへコンポジット型を入れるのではなく、コンポジット型が、ジェネリック関数と型宣言

[*27] 訳注：演算子法（えんざんしほう）とは、解析学の問題、特に微分方程式を、代数的問題（普通は多項式方程式）に変換して解く方法（出典：Wikipedia）。

でコンポジット型を使って普通に処理されることです。たとえば、スライスへのインデックスをスライスの要素型 `T` に対してパラメータ化して、パラメータや `[]T` 型の変数を使う関数を期待します。

上記の `DoubleMySlice` のコード例で示したように、この方法は、コンポジット型を受け取って返し、引数型と同じ結果型を返すジェネリック関数を書くのを面倒にします。定義されたコンポジット型はよくあることではないのですが、実際に定義されています。この面倒さは、この方法の短所です。制約型推論は、呼び出し場所で役立ちます。

1.6.16　リフレクション

いかなる方法によっても、`reflect` パッケージの変更は提案していません。型や関数がインスタンス化された場合、そのすべての型パラメータは通常の非ジェネリック型となります。インスタンス化された型の `reflect.Type` 値の `String` メソッドは、角括弧内に型引数を持つ名前を返します。たとえば、`List[int]` です。

非ジェネリックコードが、インスタンス化せずにジェネリックコードを参照できません。したがって、インスタンス化されていないジェネリック型や関数に対するリフレクション情報はありません。

1.6.17　実装

ジェネリックは, 進みが遅いプログラマ、遅いコンパイラ、あるいは遅い実行時間のどれかの選択を要求する、と Russ Cox が述べているのはよく知られています。

私達は、このデザインは、異なる実装の選択肢を許すと考えています。コードは、型引数の集まりごとに別々にコンパイルされるかもしれませんし、メソッド呼び出しでのインタフェース型に似たように各型引数が処理されるかもしれないし、これらの二つの組み合わせかもしれません。

言い換えると、このデザインは、遅いプログラマを選択せずに、遅いコンパイラ（型引数の集まりごとに別々にコンパイルする）か遅い実行時間（型引数の値に対する個々の操作に対してメソッド呼び出しを使う）の間を実装が決めることを許しています。

1.6.18　まとめ

このドキュメントは長く、詳細なものとなっていますが、実際のデザインは数個の主な点に集約されます。

- 関数と型は型パラメータを持つことができ、型パラメータは制約を使って定義され、制約はインタフェース型です。
- 制約は、型引数に対する必須のメソッドと許される型を記述します。
- 制約は、型パラメータに対して許されるメソッドと操作を記述します。

- 型パラメータを持つ関数を呼び出す際に、型推論は多くの場合で型引数の省略を許します。

このデザインは、完全に後方互換です。

私達は、このデザインが、必要以上にGo言語を複雑にせずに、Go言語でのジェネリックプログラミングに対する人々の必要性を解決していると考えています。

このデザインで数年間経験せずに、Go言語への影響を正確に知ることはできません。そう言っても、推測を次に述べます。

複雑性

Goの優れた側面の一つは、その簡潔性です。このデザインは、明らかにGo言語を複雑にしています。

私達は、増加した複雑性は、ジェネリックコードを書くよりも、きちんと書かれたジェネリックコードを読むことに対しては小さいと考えています。当然ながら、開発者は型パラメータを宣言するための新たな構文を学ばなければなりません。この新たな構文、そして、インタフェースで型リストを新たにサポートすることは、このデザインにおける唯一の新たな構文上の構造です。ジェネリック関数内のコードは、後で示すコード例から分かるように通常のGoのコードのように読めます。`[]int` から `[]T` とするのは簡単な転換です。型パラメータ制約は、型を記述するドキュメントとしての役割を実質的に果たします。

ほとんどのパッケージはジェネリック型やジェネリック関数を定義しないと思いますが、多くのパッケージがどこかで定義されたジェネリック型やジェネリック関数を使うことでしょう。一般的に、ジェネリック関数はジェネリックではない関数と同じように機能します。つまり、単純に呼び出すだけです。型推論は、型引数を明示的に書く必要がないということです。型推論の規則は、予想できるように設計されています。つまり、型引数は正確に推論されるか、呼び出しが失敗して明示的な型パラメータが必要とされるかのどちらかです。型推論は、型等価（*type equivalence*）を使い、似ているが等価ではない二つの型を解決しようとはしないので、とても複雑にはならないようにしています。

ジェネリック型を使うパッケージは、明示的な型引数を渡さなければなりません。それに対する構文は分かりやすいです。唯一の変更は、関数へだけではなく、型へ引数を渡していることです。

概して、デザインで予期しない驚きがないように務めました。それがうまく行ったかは時間がたってみなければ分かりません。

広がり

数個の新たなパッケージが標準ライブラリに追加される期待されます。新たな `slices` パッケージは、既存の `bytes` パッケージや `strings` パッケージに似て、任意の要素型のスライスに対して操作を行うでしょう。新たな `map` パッケージと `chans` パッケージは、個々の要素型ごとに現在は複製されているアルゴリズムを提供するでしょう。`sets` パッケージ

が追加されるかもしれません。

新たな constraints パッケージは、すべての整数型やすべての数値型を許すような制約といった標準制約を提供するでしょう。

container/list や container/ring などのパッケージ、それに sync.Map や sync/atomic.Value などの型は、新たな名前を使うか、新たなバージョンのパッケージを使うことで、コンパイル時の型安全になるように更新されるでしょう。

math パッケージは、ずっと人気がある Min 関数や Max 関数といった、すべての数値型に対する単純な標準アルゴリズムを提供するように拡張されるでしょう。

sort パッケージにジェネリック版を追加するかもしれません。

新たな特殊用途のコンパイル時型安全なコンテナ型が開発されることはあり得ます。

C++ STL のイテレータ型のような方法が広く使われることは期待していません。Go では、その種の考えは、インタフェース型を使ってより自然に表現されます。C++ の観点からは、イテレータに対してインタフェース型を使うのは、抽象化ペナルティを持つと見なせます。つまり、実行時効率は、実質的にすべてのコードをインライン化している C++ の方法よりは低くなります。しかし、私達は、Go プログラマはその種のペナルティを受け入れると考えています。

多くのコンテナ型が開発されれば、標準 Iterator インタフェースを開発するかもしれません。そうすると、それは range 節で Iterator を使えるような何らかの機構を追加するように Go 言語を修正する圧力となるかもしれません。しかし、これはとても不確かです。

効率性

ジェネリックコードからどの種の効率性を開発者が期待するのかは、明らかではありません。

ジェネリック型というよりは、ジェネリック関数はインタフェースの基づく方法を使っておそらくコンパイルできます。それはコンパイル時間を最適化し、関数は一度だけしかコンパイルされませんが、実行時のコストが伴うでしょう。

ジェネリック型は、型引数の集まりごとに複数回コンパイルされるのが最も必然かもしれません。これは、明らかにコンパイル時のコストを伴いますが、実行時のコストは伴うべきではありません。型パラメータに依存している各要素へアクセスするために特殊用途のメソッドを使って、インタフェース型と同様にジェネリック型を実装すると選択もできます。

この領域で開発者が何を期待するかは、経験を積まなければ分からないでしょう。

省略

私達は、このデザインは、ジェネリックプログラミングに対する基本要件を満たしていると考えています。しかし、サポートされていない多くのプログラミング構造があります。

- 特化（*specialization*）はサポートしていない。特定の型引数を扱うように設計されたジェネリック関数の複数バージョンを書く方法はありません。

- メタプログラミング（*metaprogramming*）はサポートしていない。実行時に実行されるコードを生成するために、コンパイル時に実行されるコードを書く方法はありません。
- 高レベルの抽象化はサポートしていない。型引数を持つ関数は、呼び出すかインスタンス化する以外に使う方法はありません。ジェネリック型は、インスタンス化する以外に使う方法はありません。
- 汎用型記述（*general type description*）はサポートしていない。ジェネリック関数で演算子を使うために、制約は、型が持つべき特性を記述するのではなく、特定の型を列挙します。これは理解するのは容易ですが、時には、制限になるかもしれません。
- 関数パラメータの共変性（*covariance*）や反変性（*contravariance*）はサポートしていません。
- 演算子法（*operator method*）はサポートしてません。コンパイル時安全な汎用コンテナを書けますが、通常のメソッドでアクセスできるだけであり、`c[k]` といった構文ではアクセスできません。
- カリー化（*currying*）はサポートしていません。ヘルパー関数やラッパー型を使わずに、ジェネリック関数やジェネリック型を部分的にインスタンス化する方法はありません。すべての型引数は明示的に渡されるか、インスタンス化されるときに推論されなければなりません。
- 可変個型パラメータ（*variadic type parameters*）はサポートしていません。型パラメータと普通のパラメータの両方を異なる個数受け取る単一のジェネリック関数を書くことを許す可変個型パラメータはサポートしていません。
- アダプター（*adaptors*）はサポートしていません。制約をまだ実装していない型引数をサポートするために使うアダプターを制約が定義する方法はありません。たとえば、`Equal` メソッドの観点から==演算子を定義したり、その逆を定義したりする方法はありません。
  定数といった非型値に対するパラメータ化はサポートしていません。これは、配列で最も発生し、`type Matrix[n int] [n][n]float64` と書けるのが便利かもしれません。要素のデフォルト値といったコンテナ型に対する意味がある値を指定できることも役立かもしれません。

問題点

このデザインには、詳細に説明すべき問題点があります。私達は、全体としてのデザインの比較すれば、比較的ささいなことと考えていますが、意見を聞いて、議論する必要があります。

問題点：ゼロ値

このデザインは、型パラメータのゼロ値に対する単純な表現を含んでいません。たとえば、ポインタを使っているオプショナル値の次の実装を考えてみてください。

```
type Optional[T any] struct {
    p *T
}

func (o Optional[T]) Val() T {
    if o.p != nil {
        return *o.p
    }
    var zero T
    return zero
}
```

`o.p == nil` の場合、`T` のゼロ値を返したいですが、それを書く方法はありません。`return nil` と書けたらよいのでしょうが、たとえば `T` が `int` であればうまくいきません。その場合、`return 0` と書かなければならなくなります。そして、もちろん、`return nil` か `return 0` のどちらかをサポートする制約を書く方法はありません。

この問題への対応は次の通りです。

- 上記のように `var zero T` を使うことです。既存のデザインでうまく機能しますが、余分なコードを必要とします。
- 謎めいていますが、既存のデザインで機能する`*new(T)` を使うことです。
- 結果に対してのみですが、結果パラメータに名前付けして、ゼロ値を返すために空 `return`[*28] を使います。
- すべてのジェネリック型 のゼロ値として `nil` を許すようにデザインを拡張する（issue 22729[*29] を参照）。
- `T` が型パラメータとして、その型のゼロ値を示すために `T{}`を許すようにデザインを拡張する。
- issue 19642[*30] で提案されているように、（`return` あるいは関数呼び出しを含む）代入の右辺に `_`（アンダースコア）を使うことを許すように Go 言語を変更する。

[*28] 訳注：『プログラミング言語 Go』では、空リターン（*bare return*）と記述されているのですが、言語仕様には正式な名称は記述されておらず、このデザインドキュメントでは *naked return* と記述されています。
[*29] https://golang.org/issue/22729
[*30] https://golang.org/issue/19642

- issue 21182[*31] で提案されているように、結果の型のゼロ値を返すために `return ...` を許すように Go 言語を変更する。

どのような対応を行うかを決める前に、このデザインで多くの経験を積む必要があると考えています。

問題点：一致した事前宣言型の特定

このデザインは、型引数と一致した基底型を検査する方法を提供していません。型引数を空インタフェースへ変換して、型アサーションか型 switch を使うという少しぎこちない方法で、実際の型引数を検査できます。しかし、それは、基底型と同じではない実際の型引数を検査することになってしまいます。

次が、その違いを示すコード例です。

```
type Float interface {
    type float32, float64
}

func NewtonSqrt[T Float](v T) T {
    var iterations int
    switch (interface{})(v).(type) {
    case float32:
        iterations = 4
    case float64:
        iterations = 5
    default:
        panic(fmt.Sprintf("unexpected type %T", v))
    }
    // 残りのコードは省略。
}

type MyFloat float32

var G = NewtonSqrt(MyFloat(64))
```

このコードは、`G` を初期化したときにパニックします。なぜなら、`NewtonSqrt` 関数内の `v` の型は `float32` でも `float64` でもなく、`MyFloat` になるからです。この関数が実際に行いたいことは、`v` の型ではなく、`v` が制約内で一致した型です。

これを行う一つの方法は、`T` 型は常に制約で定義された型に一致すると規定して、`T` 型に対して型 switch を許すことです。制約が明示的な型を列挙していて、制約に列挙された型だけが `case` として許されれば、この種の型 switch は許されるでしょう。

[*31] https://golang.org/issue/21182

問題点：変換可能性を表現できない

このデザインは、二つの異なる型パラメータ間の変換可能性を表現する方法を提供していません。たとえば、次の関数を書く方法はありません。

```
// Copy は src から dst へ、変換しながらコピーします。
// Copy は、コピーされた項目数を返し、それは dst と src の長さの短い方です。
// この実装は不正です。
func Copy[T1, T2 any](dst []T1, src []T2) int {
    for i, x := range src {
        if i > len(dst) {
            return i
        }
        dst[i] = T1(x) // 不正
    }
    return len(src)
}
```

T2 型から T1 型への変換は、どちらの型に対してもその変換を許す制約がないからです。もっと悪いことに、一般にそのような制約を書く方法はありません。T1 と T2 が何らかの型リストを要求できるような特殊な状況では、この関数は、型リストを使った型変換（1.6.15節）で説明したように書けます。しかし、たとえば、T1 がインタフェース型で、T2 がそのインタフェースを実装した型の場合に対する制約を書く方法ありません。

T1 がインタフェース型であれば、空インタフェース型への変換と型アサーションを使って書けることは注目すべき点です。しかし、もちろんコンパイル時の型安全ではありません。

```
// Copy は src から dst へ、変換しながらコピーします。
// Copy は、コピーされた項目数を返し、それは dst と src の長さの短い方です。
func Copy[T1, T2 any](dst []T1, src []T2) int {
    for i, x := range src {
        if i > len(dst) {
            return i
        }
        dst[i] = (interface{})(x).(T1)
    }
    return len(src)
}
```

問題点：パラメータ化されたメソッドはサポートしない

このデザインは、メソッドに固有の型パラメターの宣言をメソッドが行うことを許していません。レシーバは型パラメータを持てますが、メソッドが型パラメータを追加できませ

ん[*32]。

Goでは、メソッドの主な役割の一つは、型がインタフェースを実装することを許すことです。パラメータ化されたメソッドにインタフェースを実装することを許すことが合理的に可能であるかは、明確ではありません。たとえば、パラメータ化されたメソッドに対する明らかな構文を使っている次のコード例を考えてみてください。このコードは、問題点を明瞭にするために複数のパッケージを使っています。

```
package p1

// S はパラメータ化されたメソッド Identity を持つ型。
type S struct{}

// Identity は、任意の型で機能する単純な恒等（identity）メソッドです。
func (S) Identity[T any](v T) T { return v }
```

```
package p2

// HasIdentity は、パラメータ化された Identity メソッドを持つ
// すべての型に一致するインタフェース。
type HasIdentity interface {
    Identity[T any](T) T
}
```

```
package p3

import "p2"

// CheckIdentity は、Identity メソッドが存在するか検査します。
// この関数はパラメータ化されたメソッドを呼び出しますが、
// 関数自身はパラメータ化されていないことに注意してください。
func CheckIdentity(v interface{}) {
    if vi, ok := v.(p2.HasIdentity); ok {
        if got := vi.Identity[int](0); got != 0 {
            panic(got)
        }
    }
}
```

```
package p4

import (
    "p1"
```

[*32] 訳注：Javaでのジェネリックメソッドと同じ機能は提供されないということです。

```
    "p3"
)

// CheckSIdentity は S の値を CheckIdentity へ渡します。
func CheckSIdentity() {
    p3.CheckIdentity(p1.S{})
}
```

この例では、パラメータ化されたメソッドを持つ p1.S 型と、パラメータ化されたメソッドを持つ p2.HasIdentity 型を持っています。したがって、p3.CheckIdentity 関数は、int 引数で vi.Identity を呼び出し、p4.CheckSIdentity からの呼び出しで p1.S.Identity[int] が呼び出されます。しかし、p3 パッケージは p1.S については何も知りません。プログラムのどこかに p1.S.Identity への他の呼び出しがないかもしれません。どこかで p1.S.Identity[int] をインスタンス化する必要がありますが、どのように行うのでしょうか。

リンク時にインスタンス化できるでしょうが、一般的な場合、CheckIdentity へ渡されるかもしれない型の集合を決めるために、リンカーがプログラムの完全な呼び出しグラフを走査する必要があります。そして、ユーザからの文字列入力に基づいてメソッドをリフレクションで検索するかもしれないように型リフレクションが関係している一般的な場合、その操作は十分ではありません。したがって、一般に、パラメータ化されたメソッドをリンカーでインスタンス化するには、すべての可能な型引数に対してすべてのパラメータ化されたメソッドをインスタンス化する必要があるかもしれず、それは受け入れられないと思われます。

あるいは、実行時にインスタンス化できます。実行時のインスタンス化は、一般的に何らかの JIT を使うか、何らかのリフレクションに基づく方法を使うためにコードをコンパイルすることを意味します。どちらの方法も実装するには複雑すぎるでしょうし、実行時に驚くほど遅くなるでしょう。

あるいは、パラメータ化されたメソッドはインタフェースを実装しないと決められますが、そうすると、メソッドがなぜ必要なのかはとても不明瞭になります。インタフェースを無視したら、パラメータ化されたメソッドは、パラメータ化された関数として実装できます。

したがって、パラメータ化されたメソッドは一見して明らかに有益に思えますが、それが何を意味して、どのように実装するかを決めなければなりません。

問題点：ポインタメソッドを要求する方法はありません

場合によっては、パラメータ化された関数は、アドレス化可能な値に対して常にメソッドを呼び出すように書くのが自然です。たとえば、スライスの各要素に対してメソッドを呼び出す場合にはそうなります。そのような場合、関数は、そのメソッドがスライスの要素型のポインタメソッドの集合にあることを要求するだけです。このデザインで説明されている型制約は、その要件を記述する方法を提供していません。

たとえば、前に示した Stringify のコード例（1.6.6節）の変形を考えてみてください。

```
// Stringify2 は、S の各要素に対して String メソッドを呼び出し、
// その結果を返します。
func Stringify2[T Stringer](s []T) (ret []string) {
    for i := range s {
        ret = append(ret, s[i].String())
    }
    return ret
}
```

`[]bytes.Buffer` があり、それを `[]string` へ変換したいと考えてみてください。`Stringify2` 関数は、ここでは役立ちません。`Stringify2[bytes.Buffer]` と書きたいですが書けません。なぜなら、`bytes.Buffer` は `String` メソッドを持っていないからです。`Stringify2[*bytes.Buffer]` と書いてもダメです。なぜなら、関数が `[]*bytes.Buffer` を期待しますが、あるのは `[]bytes.Buffer` だからです。

前に「ポインタメソッドの例」（26ページ）で同様なケースを説明しました。その時は、問題を単純にするために制約型推論を使いました。ここでは、それは役立ちません。なぜなら、`Stringify2` はポインタメソッドを呼び出すことについては何も関心がないからです。それは、`String` メソッドを持つ型を求めているだけであり、そのメソッドが、値メソッドの集合ではなく、ポインタメソッドの集合にだけ含まれていれば問題ありません。しかし、メソッドが値メソッドの集合に含まれるような場合も受け付けたいのです。たとえば、`[]*bytes.Buffer` を持っていたとしてもです[*33]。

必要とするのは、型制約をポインタメソッドの集合か値メソッドの就業のどちらかに適用すると書ける方法です。その場合、関数の本体は、その型のアドレス化可能な値に対してメソッドを呼び出すことを要求されます。

この問題が実際にどのくらい頻繁に発生するかは明らかではありません。

問題点：浮動小数と複素数に関連がない

制約型推論は、スライス型の要素に名前を付けさせてくれて、他の同様な型分解を適用するために名前を付けさせてくれます。しかし、浮動小数型と複素数型を関連付ける方法はありません。たとえば、このデザインでは事前宣言の `real` 関数、`imag` 関数、`complex` 関数を書く方法はありません。引数型が `complex64` であれば結果の型は `float32` になるを、記述する方法はありません。

一つの可能な方法は、型制約として `real(T)` を許すことです。それは、「複素数型 `T` に関連付けられている浮動小数型」を意味します。同様に、`complex(T)` は、浮動小数点型 `T` に関連付けられている複素数型を意味します。制約型推論は、呼び出しコードを単純にするでしょう。しかし、他の型制約とは異なったものになります。

[*33] 訳注：意味不明

破棄されたアイデア

このデザインは完璧ではありませんし、改善するための方法はあるでしょう。そうは言っても、すでに詳細を検討した多くのアイデアがあります。この節では、同じ議論を避けることを期待して、それらのアイデアの一部を説明します。アイデアは、FQA の形式で示されています。

破棄されたアイデア：コントラクトはどうなったのか

ジェネリックの初期のドラフトデザインは、コントラクト（*contract*）と呼ばれる新たな言語構造を使って制約を実装していました。型リストは、インタフェース型ではなく、コントラクトにだけ書けました。しかし、多くの開発者にとって、コントラクトとインタフェース型の違いを理解するは難しかったです。また、コントラクトは、対応するインタフェースの集合として表現できることも分かったのです。つまり、コントラクトがなくても表現力は失われないと分かりました。インタフェース型だけを使うことでデザインを単純にするこにしました。

破棄されたアイデア：型リストの代わりに、なぜメソッドを使わないのか

型リストは奇妙である。
すべての演算になぜメソッドを書かないのか。

メソッド名として演算子トークンを許すことは可能であり、`+(T) T` といったメソッドになります。あいにく、それだけでは十分ではありません。`<<(`**整数**`) T` といったシフトや `[](`**整数**`) T` といったインデックス指定操作のために、任意の整数型に一致する型を記述する何らかの方法が必要になり、それは単一の `int` 型に限定されません。`==(T)` **型付けなし** `bool` といった操作のために型付けなしブーリアン型も必要となります。変換といった操作のために新たな記法を導入したり、型に対して範囲をループすることを表現したりする必要があるでしょうし、おそらく新たな構文を必要とします。型付けなし定数の正当な値を記述する何らかの方法が必要となります。`<(T) bool` のサポートすることが、ジェネリック関数が<=も使えるという意味になるのか、そして同様に `+(T) T` のサポートすることが、関数が ++ も使えるという意味にになるかのを検討しなければなりません。演算子トークンがうまく機能させることは可能でしょうが、簡単ではありません。このデザインで用いられている方法は、単純ですし、新たな構文構造（型リスト）と新たな名前（`comparable`）だけに依存します。

破棄されたアイデア：パッケージに対して、なぜ型パラメータを追加しないのか

これに関しては広範囲に調査しました。それは、`list` パッケージを書き、そのパッケージに一つの型要素の `List` を別の型要素の `List` へ変換する `Transform` 関数を書きたい場

合、問題となります。あるパッケージの一つのインスタンス化内の関数が同じパッケージの異なるインスタンス化を要求する型を返すのはとても困難です。

型定義を持つパッケージの境界を混乱させもします。ジェネリック型の利用がパッケージにきちんと分割されると考えるべき特別な理由はありません。時には、そうなるかもしれませんが、ならないかもしれません。

破棄されたアイデア：C++ や Java のように、なぜ `F<T>`構文を使わないのか

`v := F<T>`といった関数内のコードを解析する場合、`<`に遭遇した時点で、型のインスタンス化なのか、`<`演算子を使った式なのかは曖昧です。型情報なしでは解決するのが困難です。

たとえば、次のような文を考えてみてください。

```
a, b = w < x, y > (z)
```

型情報なしでは、代入の右辺が式の一組（`w < x` と `y > (z)`）なのか、ジェネリック関数のインスタンス化でその呼び出し（`(w<x, y>)(z)`）が二つの値を返すのかを決められません。

型情報なしで解析が可能というのが Go の重要なデザイン決定であり、それは角括弧（`<>`）を使うの不可能に思えます。

破棄されたアイデア：なぜ `F(T)` 構文を使わないのか

このデザインの初期のバージョンは、この構文を使っていました。それはうまくいっていましたが、いくつかの解析問題を抱えていました。たとえば、`var f func(x(T))` と書いた場合、これはインスタンス化された型 `x(T)` の一つの名前なしパラメータを持つ関数かのか、`(T)` 型を持つ名前があるパラメータ `x` を持つ関数（たいていは、`func(x T)` と書きますが、これはパラメータ化された型の場合です）なのかが不明瞭です

他の曖昧さもありました。`[]T(v1)` と `[]T(v2){}`に対して、開き括弧の時点で、これが（`v1` の値を `[]T` への）型変換なのか、型リテラル（その型は、インスタンス化された型 `T(v2)` なのか分かりません。`interface { M(T) }`に対しては、メソッド `M` を持つインタフェースなのか、埋め込まれてインスタンス化されたインタフェース `M(T)` を持つインタフェースなのか分かりません。さらに丸括弧を加えることで、これらの曖昧さは解消できますが、不格好です。

また、`func F(T any)(v T)(r1, r2 T)` や `F(int)(1)` などの宣言での括弧の数に頭を悩ます人達もいました。

破棄されたアイデア：なぜ `F « T »` 構文を使わないのか

検討はしましたが、ASCII 以外を必要にする気にはなれませんでした。

破棄されたアイデア：なぜ既存のパッケージに制約を定義しないのか

型リストを書く代わりに、`constraints.Arithmetic` や `constraints.Comparable` といった名前を使う。

型のすべての可能な組み合わせを列挙するととても長くなります。それは、ジェネリックコードの作成者だけではなく、何よりも読み手が覚えなければならない新たな名前の集まりを導入することにもなります。このデザインの目標の一つは、できる限り新たな名前を導入しないことです。このデザインでは、二つの新たな事前宣言名である `comparable` と `any` だけを導入しています。

開発者がそのような名前が役立つと分かれば、`constraints` パッケージを導入できると考えており、そのパッケージは、他の型や関数から使えて、他の制約に埋め込める制約の形式でそのような名前を定義します。それは、標準ライブラリで最も役立つ名前を定義し、一方で、適切な場所では他の型の組み合わせを使うという柔軟性を開発者に与えます。

破棄されたアイデア：型パラメータを型とする値に対して、なぜ型アサーションを許さないのか

このデザインの初期のバージョンでは、型パラメータを型とする値、あるいは、型パラメータに基づく型を型とする値に対して型アサーションや型 switch を使うことを許していました。私達はその機構を取り除きました。なぜなら、任意の型の値は空インタフェース型へ常に変換できて、それからそれに対して型アサーションや型 switch を使えるかれです。また、型リストを持つ制約内で、型アサーションや型 switch が、型引数の基底型ではなく実際の型引数を使うことは、ときには混乱をもたらしていました（違いについては、「問題点：一致した事前宣言型の特定」（45ページ）で説明しています。）

Java との比較

Java のジェネリックスに関するほとんどの不満は、型イレージャ（*type erasure*）に集中しています。このデザインは、型イレージャは使っていません。ジェネリック型に対するリフレクション情報は、完全なコンパイル時の型情報を含んでいます。

Java の型では、ワイルドカード（`List<? extends Number>`や `List<? super Number>`）は、共変性（*covariance*）や反変性（*contravariance*）を実装しています。Go にはそれらの概念はありませんので、ジェネリック型がとても単純になっています。

C++ との比較

（コンセプト（*concept* プロポーザルが採用されなければ）C++ のテンプレート（*template*）は型引数に対して何も制約を強制しません。それは、テンプレートのコードの変更は、誤ってかなり離れたインスタンス化を壊すことを意味します。また、エラーメッセージはインス

タンス化のときだけに報告されて、深くネストすることがあり、理解が困難です。このデザインは、必須で明示的な制約を通してそのような問題を防いでいます。

C++ は、テンプレート・メタプログラミング（*template metaprogramming*）をサポートしています。それは、テンプレートを使っていない C++ とは異なる構文を使うコンパイル時に行われる普通のプログラミングと考えられます。このデザインは、類似の機能は持っていません。そのことにより、何らかの力と実行時の効率を失いますが、かなりの複雑さを省いています。

C++ は二段階の名前検索を使っており、名前によってはテンプレート定義の文脈で検索され、そして名前によってはテンプレートインスタンス化の文脈で検索されます。このデザインでは、すべての名前は、書かれた時点で検索されます。

実際のところ、すべての C++ コンパイラは、個々のテンプレートをインスタンス化された時点でコンパイルします。それは、コンパイル時間の低下をもたらすことがあります。このデザインでは、ジェネリック関数のコンパイルの処理方法に関しては、柔軟性を提供しています。

Rust との比較

このデザインで記述されているジェネリックスは、Rust でのジェネリックスに似ています。

一つの違いは、Rust ではトレイト境界（*trait bound*）と型の間の関連は、トレイト境界を定義しているクレイト（*crate*）か、その型を定義しているクレイトのどちらかに明示的に定義しなければならないことです。Go に関しては、これは、型が制約を満足しているかをどこかに宣言しなければならないことを意味します。Go の型は、明示的な宣言なしで Go のインタフェースを満足させられるように、このデザインでは、Go の型引数は明示的な宣言なしで制約を満足できます。

このデザインで型リストを使っている場所で、Rust の標準ライブラリは、比較などの操作のための標準トレイトを定義しています。それらの標準トレイトは、Rust の基本データ型で自動的に実行されており、ユーザ定義の型でも同様に実装できます。Rust、少なくとも 34 個の広範囲なトレイトの一覧を提供している演算子のすべてを網羅しています。

Rust はメソッドに対する型パラメータをサポートしていますが、このデザインはサポートしていません。

第2章

コード例

この章の各節は、このデザインを使ったコード例です。これは、Go にジェネリックスが欠けていることに関して開発者達が報告してきた特定の領域へ対応したものです。

2.1 マップ/リデュース/フィルター

スライスに対して、マップ（*map*）、リデュース（*reduce*）、フィルター（*filter*）の関数の書き方の例です。これらの関数は、Lisp、Python、Java などの同様な関数に対応したものです。

```
// パッケージ slices は、さまざまなスライスアルゴリズムを実装しています。
package slices

// Map は、マッピング関数を使って []T1 を []T2 へ変換します。
// この関数は二つの型パラメータである T1 と T2 を持っています。
// この関数は、任意の型のスライスに対して機能します。
func Map[T1, T2 any](s []T1, f func(T1) T2) []T2 {
    r := make([]T2, len(s))
    for i, v := range s {
        r[i] = f(v)
    }
    return r
}

// Reduce は、リダクション関数を使って []T1 を単一の値へとリデュースします。
func Reduce[T1, T2 any](
    s []T1,
    initializer T2,
    f func(T2, T1) T2,
) T2 {
    r := initializer
    for _, v := range s {
        r = f(r, v)
    }
    return r
}
```

```
// Filter は、フィルター関数を使ってスライスから値をフィルターします。
// f が true を返した s の要素だけを含む新たなスライスを返します。
func Filter[T any](s []T, f func(T) bool) []T {
    var r []T
    for _, v := range s {
        if f(v) {
            r = append(r, v)
        }
    }
    return r
}
```

次は、これらの関数の呼び出し例です。非型引数の型に基づいて型引数を決定するために型推論が使われています。

```
s := []int{1, 2, 3}

floats := slices.Map(s, func(i int) float64 { return float64(i) })
// floats は、[]float64{1.0, 2.0, 3.0}。

sum := slices.Reduce(s, 0, func(i, j int) int { return i + j })
// sum は、6。

evens := slices.Filter(s, func(i int) bool { return i%2 == 0 })
// evens は、[]int{2}。
```

2.2　マップのキー

次は、任意のマップのキーのスライスを得る方法です。

```
// パッケージ mpas は、すべてのマップ型に対して機能する汎用関数を提供しています。

package maps

// Key は、マップ m のキーをスライスで返します。
// キーは予想不可能な順序で返されます。
// この関数は、K と V の二つの型パラメータを持っています。
// マップのキーは比較可能でなければなりませんので、キーは事前宣言された制約
// comparable を持っています。マップの値は何でも可能です。
func Keys[K comparable, V any](m map[K]V) []K {
    r := make([]K, 0, len(m))
    for k := range m {
        r = append(r, k)
    }
```

```
    return r
}
```

典型的な使い方では、マップのキーと値の型は推論されます。

```
    k := maps.Keys(map[int]int{1:2, 2:4})
    // kは、[]int{1, 2}か[]int{2, 1}のどちらかです。
```

2.3　セット

多くの開発者がGoの組み込みのマップ型は、セット型をサポートするために拡張、むしろ縮小すべきだと要求してきました。次は、セット型の型安全な実装ですが、[]などの演算子ではなくメソッドを使うものです。

```
// パッケージsetsは、任意の比較可能な型のセットを実装しています。
package sets

// Setは、値のセットです。
type Set[T comparable] map[T]struct{}

// Makeは何らかの要素型のセットを返します。
func Make[T comparable]() Set[T] {
    return make(Set[T])
}

// Addはセットsにvを追加します。
// vがs内にすでにあれば、何も影響を及ぼしません。
func (s Set[T]) Add(v T) {
    s[v] = struct{}{}
}

// Deleteは、セットsからvを取り除きます。
// vがs内になければ、何も影響を及ぼしません。
func (s Set[T]) Delete(v T) {
    delete(s, v)
}

// Containsはvがs内にあるか報告します。
func (s Set[T]) Contains(v T) bool {
    _, ok := s[v]
    return ok
}

// Lenはs内の要素数を報告します。
```

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

// Iterate は、s の各要素に対して f を呼び出します。
// f が Delete メソッドを呼び出すのは問題ありません。
func (s Set[T]) Iterate(f func(T)) {
    for v := range s {
        f(v)
    }
}
```

使用例は次の通りです。

```
// int のセットを生成する。
// 型引数として int を渡します。
// それから、Make は非型パラメータを受け取らないので、() を書きます。
// Make へ明示的な型引数を渡さなければなりません。
// 関数引数型推論は行われません。なぜなら、Make への型引数は
// 結果パラメータ型にだけ使われるからです。
s := sets.Make[int]()

// セット s に値 1 を追加する。
s.Add(1)

// s が値 2 を含んでいないことを検査する。
if s.Contains(2) { panic("unexpected 2") }
```

この例は、このデザインが既存の API に対するコンパイル時型安全なラッパーを提供する方法を示しています。

2.4　ソート

sort.Slice を導入する以前は、よくあった不平は、sort.Sort を使うために決まりきった定義が必要だったことです。このデザインでは、sort パッケージに次のように追加できます。

```
// Ordered は、すべての順序付けされた型と一致する型制約です
//  （順序付けされた型は、< <= >= > 演算子をサポートする型です。）
// 実際には、この型制約は標準ライブラリのパッケージで定義されるでしょう。
type Ordered interface {
    type int, int8, int16, int32, int64,
        uint, uint8, uint16, uint32, uint64, uintptr,
        float32, float64,
```

```
        string
}

// orderedSlice は、sort.Interface を実装している内部の型です。
// Less メソッドは < 演算子を使っています。Ordered 型制約は、
// T が < 演算子を持つことを保証しています。
type orderedSlice[T Ordered] []T

func (s orderedSlice[T]) Len() int {
    return len(s)
}
func (s orderedSlice[T]) Less(i, j int) bool {
    return s[i] < s[j]
}
func (s orderedSlice[T]) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}

// OrderedSlice は、昇順でスライス s をソートします。
// s の要素は < 演算子を使って順序付けできなければなりません。
func OrderedSlice[T Ordered](s []T) {
    // s を orderedSlice[T] 型へ変換します。
    // s は []T なのと、orderedSlice[T] は []T と定義されているので、
    // その変換は許されます。
    // orderedSlice[T] は sort.interface を実装しており、
    // その結果を sort.Sort へ渡せます。
    // 要素は < 演算子を使ってソートされます。
    sort.Sort(orderedSlice[T](s))
}
```

これで、次のように書けます。

```
s1 := []int32{3, 5, 2}
sort.OrderedSlice(s1)
// s1 は、[]int32{2, 3, 5}

s2 := []string{"a", "c", "b"})
sort.OrderedSlice(s2)
// s2 は、[]string{"a", "b", "c"}
```

同じように、sort.Slice に似た、比較関数を使ってソート用関数を追加できますが、その関数はスライスのインデックスではなく値を取ります。

```
// sliceFn は、sort.Interface を実装している内部の型です。
// Less メソッドは cmp フィールドを呼び出します。
type sliceFn[T any] struct {
```

```
    s   []T
    cmp func(T, T) bool
}

func (s sliceFn[T]) Len() int {
    return len(s.s)
}
func (s sliceFn[T]) Less(i, j int) bool {
    return s.cmp(s.s[i], s.s[j])
}
func (s sliceFn[T]) Swap(i, j int) {
    s.s[i], s.s[j] = s.s[j], s.s[i]
}

// SliceFn は、関数 cmp に従ってスライス s をソートします。
func SliceFn[T any](s []T, cmp func(T, T) bool) {
    sort.Sort(sliceFn[T]{s, cmp})
}
```

この関数の呼び出し例は次のようになります。

```
var s []*Person
// ...
sort.SliceFn(s, func(p1, p2 *Person) bool {
    return p1.Name < p2.Name
})
```

2.5　チャネル

多くの単純な汎用チャネル関数が書かれていません。なぜなら、それらはリフレクションを使って書かなければなりませんし、呼び出しもとは結果を型アサーションしなければなりません。このデザインでは、そのような関数を書くのが簡単になります。

```
// パッケージ chans はさまざまなチャネルアルゴリズムを実装しています。
package chans

import "runtime"

// Drain は、チャネルから残りの要素を読み捨てます。
func Drain[T any](c <-chan T) {
    for range c {
    }
}
```

```
// Merge は、何らかの要素型の二つのチャネルを単一のチャネルへ統合します。
func Merge[T any](c1, c2 <-chan T) <-chan T {
    r := make(chan T)
    go func(c1, c2 <-chan T, r chan<- T) {
        defer close(r)
        for c1 != nil || c2 != nil {
            select {
            case v1, ok := <-c1:
                if ok {
                    r <- v1
                } else {
                    c1 = nil
                }
            case v2, ok := <-c2:
                if ok {
                    r <- v2
                } else {
                    c2 = nil
                }
            }
        }
    }(c1, c2, r)
    return r
}

// Ranger は、受信側が値の読み出しを止めた場合、
// 値を送信しているゴルーチンを終了させる便利な方法を提供しています。
//
// Ranger は Sender と Receiver を返します。Receiver は、値を取り出すための
// Next メソッドを提供しています。Sender は、値を送信するための Send メソッド
// と、値の送信を止めるための Close メソッドを提供しています。Next メソッドは、
// Sender が閉じられたことを示し、Send メソッドは Receiver が解放されたことを
// 示します。
func Ranger[T any]() (*Sender[T], *Receiver[T]) {
    c := make(chan T)
    d := make(chan bool)
    s := &Sender[T]{values: c, done: d}
    r := &Receiver[T]{values: c, done: d}
    // レシーバーに対するファイナライザは、
    // レシーバーが受信を止めたかを送信側に伝えます。
    runtime.SetFinalizer(r, r.finalize)
    return s, r
}

// Sender は、Receiver へ値を送信するために使います。
type Sender[T any] struct {
    values chan<- T
```

```
    done   <-chan bool
}

// Send は受信側へ値を送信します。さらに値を送信できるかを報告します。
// つまり、値が送信できなかったら false を返します。
func (s *Sender[T]) Send(v T) bool {
    select {
    case s.values <- v:
        return true
    case <-s.done:
        // レシーバが受信を止めた。
        return false
    }
}

// Close は、受信側にこれ以上値が送られないことを伝えます。
// Close が呼び出された後は、Sender はそれ以上使えません。
func (s *Sender[T]) Close() {
    close(s.values)
}

// Receiver は、Sender から値を受信します。
type Receiver[T any] struct {
    values <-chan T
    done  chan<- bool
}

// Next は、チャネルからの次の値を返します。bool の結果は、値が有効かを示しま
// す。値が有効でなければ、Sender は閉じられて、それ以上の値は受信されません。
func (r *Receiver[T]) Next() (T, bool) {
    v, ok := <-r.values
    return v, ok
}

// finalize は受信側用のファイナライザです。
// 受信側が受信を止めたことを送信側に伝えます。
func (r *Receiver[T]) finalize() {
    close(r.done)
}
```

次の節に、この関数を使うコード例があります。

2.6　コンテナ

Go でのジェネリックスに対する度重なる要求の一つは、コンパイル時型安全なコンテナを書けることです。このデザインは、既存のコンテナに対して、コンパイル時型安全なラッ

パーを書くことを容易にしています。しかし、それに対するコード例は示しません。このデザインは、ボクシングを使わないンパイル時型安全なコンテナを書くことも容易にしています。

次は、二分木として実装された順序付けされたマップの例です。それがどのように機能するかの詳細は重要ではありません。重要な点は、次の通りです。

- コードは、必要とされる場所でキー型と値型を使って自然な Go のスタイルで書かれています。
- キーと値は直接ツリーのノードに直接保存されており、ポインタは使っておらず、インタフェース値としてボクシングされていません。

```
// パッケージ orderedmaps は、二分木として実装された順序付けされたマップを
// 提供しています。
package orderedmaps

import "chans"

// Map は順序付けされたマップです。
type Map[K, V any] struct {
    root    *node[K, V]
    compare func(K, K) int
}

// node は、二分木のノードの型です。
type node[K, V any] struct {
    k           K
    v           V
    left, right *node[K, V]
}

// New は新たなマップを返します。
// 型パラメータ V は結果にだけ使われており、型推論は働きませんし、
// New への呼び出しでは必ず明示的な型パラメータを渡さなければなりません。
func New[K, V any](compare func(K, K) int) *Map[K, V] {
    return &Map[K, V]{compare: compare}
}

// find はマップ内で k を検索し、k を保持する node へのポインタか、
// k を保持すべきノードの場所へのポインタを返します。
func (m *Map[K, V]) find(k K) **node[K, V] {
    pn := &m.root
    for *pn != nil {
        switch cmp := m.compare(k, (*pn).k); {
        case cmp < 0:
            pn = &(*pn).left
```

```
        case cmp > 0:
            pn = &(*pn).right
        default:
            return pn
        }
    }
    return pn
}

// Insert は新たな key/value をマップへ挿入します。
// そのキーがすでに存在していれば、値が置換されます。
// これが新たなキーかを報告します。
func (m *Map[K, V]) Insert(k K, v V) bool {
    pn := m.find(k)
    if *pn != nil {
        (*pn).v = v
        return false
    }
    *pn = &node[K, V]{k: k, v: v}
    return true
}

// Find はキーに関連付けされた値を返します。キーが存在しなければ、
// ゼロ値を返します。bool の結果は、キーが見つかったか報告します。
func (m *Map[K, V]) Find(k K) (V, bool) {
    pn := m.find(k)
    if *pn == nil {
        var zero V // 前述のゼロ値の説明を参照
        return zero, false
    }
    return (*pn).v, true
}

// keyValue は、イテレートする際に使われるキーと値の組です。
type keyValue[K, V any] struct {
    k K
    v V
}

// InOrder は、マップと間順走査（in-order traversal）するイテレータを
// 返します。
func (m *Map[K, V]) InOrder() *Iterator[K, V] {
    type kv = keyValue[K, V] // convenient shorthand
    sender, receiver := chans.Ranger[kv]()
    var f func(*node[K, V]) bool
    f = func(n *node[K, V]) bool {
        if n == nil {
```

```
                return true
            }
            // sender.Send が false を返したら、受信側で誰も受信していない
            / ことを意味するので、値の送信を止める。
            return f(n.left) &&
                sender.Send(kv{n.k, n.v}) &&
                f(n.right)
        }
        go func() {
            f(m.root)
            sender.Close()
        }()
        return &Iterator[K, V]{receiver}
}

// Iterator は、マップをいてレートするために使われます。
type Iterator[K, V any] struct {
        r *chans.Receiver[keyValue[K, V]]
}

// Next は、次のキーと値の組を返します。bool の結果は、キーと値が有効か
// 示します。有効でなかれば、最後に達したことを意味します。
func (it *Iterator[K, V]) Next() (K, V, bool) {
        kv, ok := it.r.Next()
        return kv.k, kv.v, ok
}
```

このパッケージを使うコードは次のようになります。

```
import "container/orderedmaps"

// 比較関数として strings.Compare を使って、
// m を string から string への順序付けされたマップに設定する
var m = orderedmaps.New[string, string](strings.Compare)

// Add は、a と b の組を m に追加します。
func Add(a, b string) {
        m.Insert(a, b)
}
```

2.7　アペンド

事前宣言された append 関数は、スライスを拡張するのに必要な決まりきったコードの代わりのために存在しています。Go 言語に append が追加される前には、bytes パッケージ

には、次のような Add 関数がありました。

```
// Add は、t の内容を s の終わりに追加して、その結果を返します。
// s が十分な容量を保てば、そのまま追加されます。そうでなければ、
// 新たな配列が割り当てられて返されます。
func Add(s, t []byte) []byte
```

Add は、二つの []byte 値を一緒にして、新たなスライスを返します。これは、この関数は、[]byte に対しては問題ありませんが、他の型のスライスを持っていたら、値を追加するために基本的に同じコードを書かなければなりませんでした。この（ジェネリックスの）デザインがその時にあれば、おそらく言語に append を追加しかなったでしょう。代わりに次のようなコードを書くことができました。

```
// パッケージ slices は、さまざまなスライスアルゴリズムを実装しています。
package slices

// Append は、t の内容を s の終わりに追加して、その結果を返します。
// s が十分な容量を保てば、そのまま追加されます。そうでなければ、
// 新たな配列が割り当てられて返されます。
func Append[T any](s []T, t ...T) []T {
    lens := len(s)
    tot := lens + len(t)
    if tot < 0 {
        panic("Append: cap out of range")
    }
    if tot > cap(s) {
        news := make([]T, tot, tot + tot/2)
        copy(news, s)
        s = news
    }
    s = s[:tot]
    copy(s[lens:], t)
    return s
}
```

このコード例は、事前宣言された copy 関数を使っていますが、問題ありません。それも書けます。

```
// Copy は t から値を s へコピーし、どちらかのスライスが終わりに達したら
// コピーを止めて、コピーされた値の数を返します。
func Copy[T any](s, t []T) int {
    i := 0
    for ; i < len(s) && i < len(t); i++ {
        s[i] = t[i]
```

```
    }
    return i
}
```

これらの関数は、次のように使えます。

```
    s := slices.Append([]int{1, 2, 3}, 4, 5, 6)
    // s は、[]int{1, 2, 3, 4, 5, 6}です。
    slices.Copy(s[3:], []int{7, 8, 9})
    // s は、[]int{1, 2, 3, 7, 8, 9}です。
```

このコードは、`string` を `[]byte` へ追加したりコピーする特別な場合を実装していません。そして、それは、事前宣言関数の実装と同じように効率的にはならないでしょう。それでも、このコード例は、このデザインを使って、追加の特別な言語の機能を必要とせずに、`append` と `copy` を一般的に一度書くだけでよいことを示しています。

2.8　メトリクス

Go Experience Report[*1] で、Sameer Ajmani はメトリクス実装を説明しています。個々のメトリクは一つの値を一個以上のフィールドを持ちます。フィールドは異なる型です。メトリクを定義するには、フィールドの型を指定する必要があります。`Add` メソッドは引数としてフィールドの型を受け取り、フィールドの集まりのインスタンスを記録します。C++ による実装は、可変個引数のテンプレートを使っています。Java による実装は、型名にフィールドの個数を含めています。C++ と Java による両方の実装は、コンパイル時型安全な `Add` メソッドを提供してます。

次は、コンパイル時型安全な `Add` メソッドでもって、Go で同じ機能を提供するためのこのデザインがどのように使えるかを示したものです。型引数の可変個引数はサポートしておらず、Java と同様に引数の数が異なるごとに別々の名前を使わなければなりません。この実装は比較可能な型に対してのみ機能します。複雑な実装では、任意の型を扱える比較関数を受け付けるでしょう。

```
// パッケージ metrics は、さまざまな値のメトリクスを蓄積するための
// 一般的な機構を提供します。
package metrics

import "sync"

// Metric1 は、単一の値のメトリクスを蓄積します。
type Metric1[T comparable] struct {
```

[*1] https://medium.com/@sameer_74231/go-experience-report-for-generics-google-metrics-api-b019d597aaa4

```
    mu sync.Mutex
    m  map[T]int
}

// Add は、一つの値のインスタンスを追加します。
func (m *Metric1[T]) Add(v T) {
    m.mu.Lock()
    defer m.mu.Unlock()
    if m.m == nil {
        m.m = make(map[T]int)
    }
    m.m[v]++
}

// key2 は、Metric2 で使われる内部の型です。
type key2[T1, T2 comparable] struct {
    f1 T1
    f2 T2
}

// Metric2 は、一組の値のメトリクスを蓄積します。
type Metric2[T1, T2 comparable] struct {
    mu sync.Mutex
    m  map[key2[T1, T2]]int
}

// Add は値の組のインスタンスを追加します。
func (m *Metric2[T1, T2]) Add(v1 T1, v2 T2) {
    m.mu.Lock()
    defer m.mu.Unlock()
    if m.m == nil {
        m.m = make(map[key2[T1, T2]]int)
    }
    m.m[key2[T1, T2]{v1, v2}]++
}

// key3 は、Metric3 で使われる内部の型です。
type key3[T1, T2, T3 comparable] struct {
    f1 T1
    f2 T2
    f3 T3
}

// Metric3 は、三つの値のメトリクスを蓄積します。
type Metric3[T1, T2, T3 comparable] struct {
    mu sync.Mutex
    m  map[key3[T1, T2, T3]]int
```

```
}

// Add は三つの値のインスタンスを追加します。
func (m *Metric3[T1, T2, T3]) Add(v1 T1, v2 T2, v3 T3) {
    m.mu.Lock()
    defer m.mu.Unlock()
    if m.m == nil {
        m.m = make(map[key3[T1, T2, T3]]int)
    }
    m.m[key3[T1, T2, T3]{v1, v2, v3}]++
}

// 許される最大引数まで繰り返す。
```

このパッケージを使う例は、次の通りです。

```
import "metrics"

var m = metrics.Metric2[string, int]{}

func F(s string, i int) {
    m.Add(s, i) // この呼出はコンパイル時に型検査されます。
}
```

可変個の型パラメータをサポートしていないため、この実装はある程度の繰り返しを含んでいます。しかし、このパッケージを使うのは簡単で型安全です。

2.9　リスト変換

スライスは効率的で使うのが簡単ですが、ときにはリンクリストが適切な場合があります。同じジェネリック型の異なるインスタンス化を用いた例として、次の例は、主に一つの型のリンクリストを別の型のリンクリストへの変換を示しています。

```
// パッケージ lists は、任意の型のリンクリストを提供しています。
package lists

// List は、リンクリストです。
type List[T any] struct {
    head, tail *element[T]
}

// element は、リンクリスト内のエントリです。
type element[T any] struct {
    next *element[T]
```

```
    val  T
}

// Push は、リストの終端に要素を追加します。
func (lst *List[T]) Push(v T) {
    if lst.tail == nil {
        lst.head = &element[T]{val: v}
        lst.tail = lst.head
    } else {
        lst.tail.next = &element[T]{val: v }
        lst.tail = lst.tail.next
    }
}

// Iterator は、リストをイテレートします。
type Iterator[T any] struct {
    next **element[T]
}

// Range は、リストの先頭から開始する Iterator を返します。
func (lst *List[T]) Range() *Iterator[T] {
    return &Iterator[T]{next: &lst.head}
}

// Next はイテレータを一つ進めます。
// さらに要素があるかを報告します。
func (it *Iterator[T]) Next() bool {
    if *it.next == nil {
        return false
    }
    it.next = &(*it.next).next
    return true
}

// Val は、現在の要素の値を返します。
// bool の結果は、その値が有効か報告します。
func (it *Iterator[T]) Val() (T, bool) {
    if *it.next == nil {
        var zero T
        return zero, false
    }
    return (*it.next).val, true
}

// Transform は、リストに対して変換関数を適用して、新たなリストを返します。
func Transform[T1, T2 any](
    lst *List[T1],
```

```
    f func(T1) T2,
) *List[T2] {
    ret := &List[T2]{}
    it := lst.Range()
    for {
        if v, ok := it.Val(); ok {
            ret.Push(f(v))
        }
        if !it.Next() {
            break
        }
    }
    return ret
}
```

2.10　ドット積

一般的なドット積（*dot product*）[*2] の実装は、任意の数値型のスライスに対して機能します。

```
// Numeric は、任意の数値型と一致する制約です。
// これは、標準ライブラリの constraints パッケージに含まれるでしょう。
type Numeric interface {
    type int, int8, int16, int32, int64,
        uint, uint8, uint16, uint32, uint64, uintptr,
        float32, float64,
        complex64, complex128
}

// DotProduct は、二つのスライスのドット積を返します。
// 二つのスライスが同じ長さでなければパニックします。
func DotProduct[T Numeric](s1, s2 []T) T {
    if len(s1) != len(s2) {
        panic("DotProduct: slices of unequal length")
    }
    var r T
    for i := range s1 {
        r += s1[i] * s2[i]
    }
    return r
}
```

[*2] 訳注：数学あるいは物理学においてドット積（ドットせき、英: dot product）あるいは点乗積（てんじょうせき）とは、ベクトル演算の一種で、二つの同じ長さの数列から一つの数値を返す演算。（出典：ウィキペディア）

（注：ジェネリックスによる実装方法は、DotProduct が FMA [*3] を使うかにより、浮動小数点型を使う場合の正確な結果に影響を与えるかもしれません。その問題がどれほどの程度なのか、あるいは修正方法があるのかははっきりとしていません。）

2.11　絶対差

Abs メソッドを使って、二つの数値の間の絶対差（*absolute difference*）を計算します。これは、前の例で定義された同じ Numeric を使います。

絶対差を計算する単純な場合に適した方法を使います。メソッドの正確な定義は使う型に依存して変わりますが、これは、アルゴリズムの共通部分をメソッドに分割する方法を示すことを意図しています[*4]。

```
// NumericAbs は、Abs メソッドを持つ数値型と一致します。
type NumericAbs[T any] interface {
    type int, int8, int16, int32, int64,
        uint, uint8, uint16, uint32, uint64, uintptr,
        float32, float64,
        complex64, complex128
    Abs() T
}

// AbsDifference は、a と b の差の絶対値を計算し、
// 絶対値は Abs メソッドで決定されます。
func AbsDifference[T NumericAbs](a, b T) T {
    d := a - b
    return d.Abs()
}
```

さまざまな数値型に対して適切な Abs メソッドを定義できます。

```
// OrderedNumeric は、< 演算子をサポートしている数値型に一致します。
type OrderedNumeric interface {
    type int, int8, int16, int32, int64,
        uint, uint8, uint16, uint32, uint64, uintptr,
        float32, float64
}
```

[*3] 訳注：積和の演算式において、途中の積算 $b \times c$ の演算結果を浮動小数点数の値としていったん丸めてしまうと、最終演算結果に大きな誤差が発生する。途中の積算を丸めず、積和演算を 1 命令で行なってしまうことで、最終演算結果の誤差を小さくするのが融合積和演算（fused multiply-add, FMA/FMAD）である。FMA は IEEE 754 規格の 2008 年改訂版（IEEE 754-2008）で標準化されている。（出典：ウィキペディア）

[*4] 訳注：AbsDifference の実装は明らかに正しくないです。T が uint8 で、a が 1 で b が 2 だと差は 1 ですが、AbsDifference の実装では 255 になってしまいます。

```
// Complex は、 < 演算子を持たない二つの複素数型に一致します。
type Complex interface {
    type complex64, complex128
}

// OrderedAbs は、順序付けされた数値型に対して
// Abs メソッドを定義しているヘルパー型です。
type OrderedAbs[T OrderedNumeric] T

func (a OrderedAbs[T]) Abs() OrderedAbs[T] {
    if a < 0 {
        return -a
    }
    return a
}

// ComplexAbs は、複素数型に対して Abs メソッドを定義しているヘルパー型です。
type ComplexAbs[T Complex] T

func (a ComplexAbs[T]) Abs() ComplexAbs[T] {
    d := math.Hypot(float64(real(a)), float64(imag(a)))
    return ComplexAbs[T](complex(d, 0))
}
```

次に、定義したばかりの型へと型からの変更を行うことで呼び出しもとに対して処理を行える関数を定義できます。

```
// OrderedAbsDifference は、a と b の差の絶対値を返します。
// ここで、a と b が順序付けされた型です。
func OrderedAbsDifference[T OrderedNumeric](a, b T) T {
    return T(AbsDifference(OrderedAbs[T](a), OrderedAbs[T](b)))
}

// ComplexAbsDifference は、a と b の差の絶対値を返します。
// ここで、a と b は複素数型です。
func ComplexAbsDifference[T Complex](a, b T) T {
    return T(AbsDifference(ComplexAbs[T](a), ComplexAbs[T](b)))
}
```

このデザインは、次のようなコードを書けるほど強力ではないことに注意してください。

```
// This function is INVALID.
func GeneralAbsDifference[T Numeric](a, b T) T {
    switch (interface{})(a).(type) {
    case int, int8, int16, int32, int64,
```

```
            uint, uint8, uint16, uint32, uint64, uintptr,
            float32, float64:
            return OrderedAbsDifference(a, b) // 不正
        case complex64, complex128:
            return ComplexAbsDifference(a, b) // 不正
        }
    }
```

OrderedAbsDifference と ComplexAbsDifference の呼び出しは不正です。なぜなら、Numeric を実装しているすべての型が OrderedNumeric 制約か Complex 制約を実装できるわけではないからです。この型 switch は、概念的にはこのコードは実行時に機能すべきということを意味しますが、コンパイル時にこのコードを書くためのサポートはありません。これは、上記の省略項目の一つを別の方法で表現してもので、このデザインは特化をサポートしていません。

謝辞

Go チームの多くのメンバー、Go イシュ・トラッカーに貢献してくれた多くの人々、そして、初期のドラフトデザインに対してアイデアやフィードバックを共有してくれたすべての人々に感謝します。私達はそのすべてを読みましたし、感謝しています。

For this version of the proposal in particular we received detailed feedback from Josh Bleecher-Snyder, Jon Bodner, Dave Cheney, Jaana Dogan, Kevin Gillette, Mitchell Hashimoto, Chris Hines, Bill Kennedy, Ayke van Laethem, Daniel Martí, Elena Morozova, Roger Peppe, and Ronna Steinberg.

このバージョンのプロポーザルに関しては、詳細なフィードバックを Josh Bleecher-Snyder、Jon Bodner、Dave Cheney、JaanaDogan、Kevin Gillette、Mitchell Hashimoto、Chris Hines, Bill Kennedy、Ayke van Laethem、Daniel Martí、Elena Morozova、Roger Peppe、Ronna Steinberg から受け取りました。

付録 A

詳細

この付録では、本文で取り上げるほど重要ではないと思われるデザインのさまざまな詳細を説明します。

A.1 ジェネリック型エイリアス

型エイリアスはジェネリック型を参照できますが、型エイリアスは独自のパラメータを持てません。この制限があるのは、制約のある型パラメータを持つ型エイリアスをどのように取り扱うかが不明だからです。

```
type VectorAlias = Vector
```

この場合、型エイリアスを使うには、エイリアスされたジェネリック型に対して適切な型引数を提供する必要があります。

```
var v VectorAlias[int]
```

型エイリアスは、インスタンス化された型も参照できます。

```
type VectorInt = Vector[int]
```

A.2 関数のインスタンス化

Go では通常、引数を渡さずに関数を参照できて、関数型の値を生成します。型パラメータを持つ関数では、それを行うことができません。すべての型引数は、コンパイル時に分かっていなければなりません。とは言え、型引数を渡すことで関数をインスタンス化できますが、そのインスタンス化されたものを呼び出す必要はありません。これにより、型パラメータを持たない関数値が生成されます。

```
// PrintInts は、func([]int) 型です。
var PrintInts = Print[int]
```

A.3　埋め込み型パラメータ

ジェネリック型が構造体で、その構造体のフィールドとして型パラメータが埋め込まれている場合、フィールドの名前は、型パラメータの名前です。

```
// Lockable は、Get メソッドと Set メソッドにより
// 複数のゴルーチンから同時に安全にアクセスできる値です。
type Lockable[T any] struct {
    T
    mu sync.Mutex
}

// Get は、Lockable に保存された値を返します。
func (l *Lockable[T]) Get() T {
    l.mu.Lock()
    defer l.mu.Unlock()
    return l.T
}

// Set は、Lockable に値を設定します。
func (l *Lockable[T]) Set(v T) {
    l.mu.Lock()
    defer l.mu.Unlock()
    l.T = v
}
```

A.4　埋め込み型パラメータメソッド

ジェネリック型が構造体で、その構造体のフィールドとして型パラメータが埋め込まれている場合、その型パラメータの制約のメソッドは、その構造体のメソッドに格上げされます。（セレクター解決（*selector resolution*）[*1] のために、これらのメソッドは、型パラメータの深さ 0 にあるとして扱われます。たとえ、実際の型引数内で、メソッドが埋め込み型から格上げされていたとしてもです。）

```
// NamedInt は、名前付けされた int です。
// Name は、String メソッドを持つ任意の型です。
type NamedInt[Name fmt.Stringer] struct {
    Name
    val int
}
```

[*1] https://golang.org/ref/spec#Selectors

```
// Name は、NamedInt の名前を返します。
func (ni NamedInt[Name]) Name() string {
    // String メソッドは、埋め込み Name から格上げされています。
    return ni.String()
}
```

A.5　埋め込みインスタンス化された型

インスタンス化された型を埋め込む場合、そのフィールドの名前は、型引数なしの型の名前です。

```
type S struct {
    T[int] // フィールド名は T
}

func F(v S) int {
    return v.T // not v.T[int]
}
```

A.6　型 switch の case でのジェネリック型

ジェネリック型は、型アサーションでの型として、あるいは、型 switch での case として使えます。

次は、明らかな例です。

```
func Assertion[T any](v interface{}) (T, bool) {
    t, ok := v.(T)
    return t, ok
}

func Switch[T any](v interface{}) (T, bool) {
    switch v := v.(type) {
    case T:
        return v, true
    default:
        var zero T
        return zero, false
    }
}
```

型 switch では、ジェネリック型が型 switch の他のケースと重複していも問題ありません。

最初に一致したケースが選択されます。

```
func Switch2[T any](v interface{}) int {
    switch v.(type) {
    case T:
        return 0
    case string:
        return 1
    default:
        return 2
    }
}

// S2a には、0 が設定されます。
var S2a = Switch2[string]("a string")

// S2b には、1 が設定されます。
var S2b = Switch2[int]("another string")
```

A.7　コンポジットリテラルに対する型推論

これは、現在提案していない機能ですが、将来の Go のバージョンでは検討する可能性があります。

ジェネリック型のコンポジットリテラルに対して型推論のサポートすることも考えられます。

```
type Pair[T any] struct { f1, f2 T }
var V = Pair{1, 2} // Pair(int){1, 2}と推論される
```

実際のコードで、この問題がどのくらいの頻度で発生するかは不明です。

A.8　ジェネリック関数引数に対する型推論

これは、現在提案していない機能ですが、将来の Go のバージョンでは検討する可能性があります。

次のコード例では、`FindClose` 内の `Find` の呼び出しを考えてみてください。型推論は、`Find` への型引数を `T4` と決定し、そのことから、最終的な引数の型は `func(T4, T4) bool` でなければならないことが分かります。そして、それにより、`IsClose` への型引数も `T4` でなければならないと導けます。しかし、前述した型推論アルゴリズムはそれを行えないので、明示的に `IsClose[T4]` と書かなければなりません。

最初は難解に思えるかもしれませんが、ジェネリックの `Map` 関数と `Filter` 関数へジェ

ネリック関数を渡す場合に発生します。

```
// Differ は、値との差を返す Diff メソッドを持っています。
type Differ[T1 any] interface {
    Diff(T1) int
}

// IsClose は、Diff に基づいて、a と b が近いか返します。
func IsClose[T2 Differ[T2]](a, b T2) bool {
    return a.Diff(b) < 2
}

// Find は、cmp 関数に基づいて、e に一致する s 内の最初の要素のインデックス
// を返します。一致する要素がない場合、-1 を返します。
func Find[T3 any](s []T3, e T3, cmp func(a, b T3) bool) int {
    for i, v := range s {
        if cmp(v, e) {
            return i
        }
    }
    return -1
}

// FindClose は、IsClose に基づいて、e に近い s 内の最初の要素のインデックス
// を返します。
func FindClose[T4 Differ[T4]](s []T4, e T4) int {
    // 現在の型推論アルゴリズムでは、ここでは明示的に IsClose[T4] と書かなけ
    // ればなりません。それが、ここで使える唯一の型引数であってもです。
    return Find(s, e, IsClose[T4])
}
```

A.9　型引数に対するリフレクション

reflect パッケージの変更は提案していませんが、reflect.Type に二つの新たなメソッドの追加を検討する可能性はあります。NumTypeArgument() int は型に対する型引数を返し、TypeArgument(i) Type は i 番目の型引数を返します。NumTypeArgument は、インスタンス化されたジェネリック型に対してはゼロではない値を返します。同様なメソッドが reflect.Value に追加され、その NumTypeArgument はインスタンス化されたジェネリック関数に対してはゼロではない値を返します。このような情報を扱うプログラムが書かれるかもしれません。