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 3 月 27 日

目次

第1章

プロポーザル

1.1 ステータス

これは、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て、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、関数呼び出しにおいて、型引数（*type arguments*）の省略を多くの場合で許します。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、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、こので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 に似た言語でジェネリックスがどのように機能するかをすでに知っている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、どのような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。

[*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

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て、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主な新たな機能は、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 parameters*）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 arguments*）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です。（後で、**ジェネリック型**（*generic types*）について説明します。）

```
// Print はスライスの要素を表示します。
// どのようなスライス値に対しても、この関数が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**（*constraints*）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、どのような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、どのような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、どのように非型引数からこの型引数が多くの場合で推定できるかを説明します。今のところ、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。引数とは別のリストとしてです。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しょう。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しょう。

```
// 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` はどのような型も可能です。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どのようなメソッドも呼び出せます。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**（*constraints*）と呼びます（他の言語は、型境

界（*type bounds*[*5]）、トレイト境界（*trait bounds*）[*6]、あるいはコンセプト（concepts）[*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、誤って制約を変わってしまう 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、私たちは、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信じています。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どのような型引数も許されます。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
- 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

[*5] 訳注：Java
[*6] 訳注：Rust
[*7] 訳注：C++

- 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 {
        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。

[*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

```
// 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。

```
// 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。

[*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。

```
// 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。

```
// List は、T 型の値のリンクリストです。
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]
}

// 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[*11]。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、イシュの中（46ページ）で行っています。

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:] {
```

[*11] 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。

```
            if v < r { // INVALID
                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] // panics if slice is empty
    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、どのような比較可能な型でも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、次の関数はどのような比較可能な型でも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*）[*12] 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。

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)
}
```

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

```
// 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*）[*13] 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。

- `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。

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

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[*14]。

どのように行われるかを知るために、単純な `Print` 関数[*15] 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[*16]。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[*17]。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。

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[*18]。

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[*19] を決定します[*20]。それか

[*14] 訳注：どういう意味？

[*15] 訳注：`Print` 関数の定義は、`func Print[T any](s []T) { ... }`です（4ページ）。

[*16] 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。

[*17] 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。

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

[*19] 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。

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

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。

```
// 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[*21]。

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。

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

[*21] 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。

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*）[*22]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[*23] 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。

```
// 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
```

[*22] 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。

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

```
}

// 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 を書き直してみましょう。

```
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。それは、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 を見返してみましょう。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) {
    // same as above
}

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。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、注意する必要がある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` をリストしていますが、`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*）[*24] 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[*25]。

```
// 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 {
```

[*24] 訳注：既存の型から作られる型を指します。

[*25] 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.

```
        // 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無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 returns the i'th entry in a slice or the value of a map
// at key i. This is valid as the result of the operator is always 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。
func Entry[T sliceOrMap](c T, i int) int {
    // This is either a slice index operation or a map key lookup.
    // Either way, the index and result types are type int.
    return c[i]
}

// sliceOrFloatMap is a type constraint for a slice or a map.
type sliceOrFloatMap interface {
    type []int, map[float64]int
}

// This function is INVALID.
// In this example the input type of the index operation is either
// int (for a slice) or float64 (for a map), so the operation is
// not permitted.
func FloatEntry[T sliceOrFloatMap](c T) int {
    return c[1.0] // INVALID: input type is either int or float64.
}
```

Imposing this restriction makes it easier to reason about the type of some operation in a generic function. It avoids introducing the notion of a value whose type is the union of a set of types found by applying some operation to each element of a type list.

(Note: with more understanding of how people want to write code, it may be possible to relax this restriction in the future.)

Type parameters in type lists

A type literal in a constraint can refer to type parameters of the constraint. In this example, the generic function ‘Map‘ takes two type parameters. The first type parameter is required to have an underlying type that is a slice of the second type parameter. There are no constraints on the second slice parameter.

```
// SliceConstraint is a type constraint that matches a slice of
// the type parameter.
type SliceConstraint[T any] interface {
```

```
    type []T
}

// Map takes a slice of some element type and a transformation function,
// and returns a slice of the function applied to each element.
// Map returns a slice that is the same type as its slice argument,
// even if that is a defined type.
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 is a simple defined type.
type MySlice []int

// DoubleMySlice takes a value of type MySlice and returns a new
// MySlice value with each element doubled in value.
func DoubleMySlice(s MySlice) MySlice {
    // The type arguments listed explicitly here could be inferred.
    v := Map[MySlice, int](s, func(e int) int { return 2 * e })
    // Here v has type MySlice, not type []int.
    return v
}
```

We showed other examples of this earlier in the discussion of [constraint type inference](#Constraint-type-inference).

Type conversions

In a function with two type parameters ‘From‘ and ‘To‘, a value of type ‘From‘ may be converted to a value of type ‘To‘ if all the types accepted by ‘From‘’s constraint can be converted to all the types accepted by ‘To‘’s constraint. If either type parameter does not have a type list, type conversions are not permitted.

This is a consequence of the general rule that a generic function may use any operation that is permitted by all types listed in the type list.

For example:

```
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
}
```

The type conversions in 'Convert' are permitted because Go permits every integer type to be converted to every other integer type.

Untyped constants

Some functions use untyped constants. An untyped constant is permitted with a value of a type parameter if it is permitted with every type accepted by the type parameter's constraint.

As with type conversions, this is a consequence of the general rule that a generic function may use any operation that is permitted by all types listed in the type list.

```
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 can convert to any integer type
    }
}

// This function is INVALID.
func Add1024[T integer](s []T) {
    for i, v := range s {
        s[i] = v + 1024 // INVALID: 1024 not permitted by int8/uint8
    }
}
```

Type lists in embedded constraints

When a constraint embeds another constraint, the type list of the final constraint is the intersection of all the type lists involved. If there are multiple embedded types, intersection preserves the property that any type argument must satisfy the requirements of all embedded types.

```
// Addable is types that support the + operator.
type Addable interface {
    type int, int8, int16, int32, int64,
        uint, uint8, uint16, uint32, uint64, uintptr,
        float32, float64, complex64, complex128,
        string
}

// Byteseq is a byte sequence: either string or []byte.
type Byteseq interface {
    type string, []byte
}

// AddableByteseq is a byte sequence that supports +.
// This is every type is that is both Addable and Byteseq.
// In other words, just the type string.
type AddableByteseq interface {
    Addable
    Byteseq
}
```

General notes on type lists

It may seem awkward to explicitly list types in a constraint, but it is clear both as to which type arguments are permitted at the call site, and which operations are permitted by the generic function.

If the language later changes to support operator methods (there are no such plans at present), then constraints will handle them as they do any other kind of method.

There will always be a limited number of predeclared types, and a limited number of operators that those types support. Future language changes will not fundamentally change those facts, so this approach will continue to be useful.

This approach does not attempt to handle every possible operator. The expectation is that composite types will normally be handled using composite types in generic function and type declarations, rather than putting composite types in a type list. For example, we expect functions that want to index into a slice to be parameterized on the slice element type ‘T‘, and to use parameters or variables of type ‘[]T‘.

As shown in the ‘DoubleMySlice‘ example above, this approach makes it awkward to declare generic functions that accept and return a composite type and want to return the same result type as their argument type. Defined composite types are not common, but they do arise. This awkwardness is a weakness of this approach. Constraint type inference can help at the call site.

1.6.16 Reflection

We do not propose to change the reflect package in any way. When a type or function is instantiated, all of the type parameters will become ordinary non-generic types. The ‘String‘ method of a ‘reflect.Type‘ value of an instantiated type will return the name with the type arguments in square brackets. For example, ‘List[int]‘.

It’s impossible for non-generic code to refer to generic code without instantiating it, so there is no reflection information for uninstantiated generic types or functions.

1.6.17 Implementation

Russ Cox [famously observed](https://research.swtch.com/generic) that generics require choosing among slow programmers, slow compilers, or slow execution times.

We believe that this design permits different implementation choices. Code may be compiled separately for each set of type arguments, or it may be compiled as though each type argument is handled similarly to an interface type with method calls, or there may be some combination of the two.

In other words, this design permits people to stop choosing slow programmers, and permits the implementation to decide between slow compilers (compile each set of type arguments separately) or slow execution times (use method calls for each operation on a value of a type argument).

1.6.18 Summary

While this document is long and detailed, the actual design reduces to a few major points.

* Functions and types can have type parameters, which are defined using constraints, which are interface types. * Constraints describe the methods required and the types permitted for a type argument. * Constraints describe the methods and operations permitted for a type parameter. * Type inference will often permit omitting type arguments when calling functions with type parameters.

This design is completely backward compatible.

We believe that this design addresses people’s needs for generic programming in Go, without making the language any more complex than necessary.

We can’t truly know the impact on the language without years of experience with this design. That said, here are some speculations.

Complexity

One of the great aspects of Go is its simplicity. Clearly this design makes the language more complex.

We believe that the increased complexity is small for people reading well written generic code, rather than writing it. Naturally people must learn the new syntax for declaring type parameters. This new syntax, and the new support for type lists in interfaces, are the only new syntactic constructs in this design. The code within a generic function reads like ordinary Go code, as can be seen in the examples below. It is an easy shift to go from '[]int' to '[]T'. Type parameter constraints serve effectively as documentation, describing the type.

We expect that most packages will not define generic types or functions, but many packages are likely to use generic types or functions defined elsewhere. In the common case, generic functions work exactly like non-generic functions: you simply call them. Type inference means that you do not have to write out the type arguments explicitly. The type inference rules are designed to be unsurprising: either the type arguments are deduced correctly, or the call fails and requires explicit type parameters. Type inference uses type equivalence, with no attempt to resolve two types that are similar but not equivalent, which removes significant complexity.

Packages using generic types will have to pass explicit type arguments. The syntax for this is straightforward. The only change is passing arguments to types rather than only to functions.

In general, we have tried to avoid surprises in the design. Only time will tell whether we succeeded.

Pervasiveness

We expect that a few new packages will be added to the standard library. A new 'slices' packages will be similar to the existing bytes and strings packages, operating on slices of any element type. New 'maps' and 'chans' packages will provide algorithms that are currently duplicated for each element type. A 'sets' package may be added.

A new 'constraints' package will provide standard constraints, such as constraints that permit all integer types or all numeric types.

Packages like 'container/list' and 'container/ring', and types like 'sync.Map' and 'sync/atomic.Value', will be updated to be compile-time type-safe, either using new names or new versions of the packages.

The 'math' package will be extended to provide a set of simple standard algorithms for all numeric types, such as the ever popular 'Min' and 'Max' functions.

We may add generic variants to the 'sort' package.

It is likely that new special purpose compile-time type-safe container types will be developed.

We do not expect approaches like the C++ STL iterator types to become widely used. In Go that sort of idea is more naturally expressed using an interface type. In C++ terms, using an interface type for an iterator can be seen as carrying an abstraction penalty, in that run-time efficiency will be less than C++ approaches that in effect inline all code; we believe that Go programmers will continue to find that sort of penalty to be acceptable.

As we get more container types, we may develop a standard 'Iterator' interface. That may in turn lead to pressure to modify the language to add some mechanism for using an 'Iterator' with the 'range' clause. That is very speculative, though.

Efficiency

It is not clear what sort of efficiency people expect from generic code.

Generic functions, rather than generic types, can probably be compiled using an interface-based approach. That will optimize compile time, in that the function is only compiled once, but there will be some run time cost.

Generic types may most naturally be compiled multiple times for each set of type arguments. This will clearly carry a compile time cost, but there shouldn't be any run time cost. Compilers can also choose to implement generic types similarly to interface types, using special purpose methods to access each element that depends on a type parameter.

Only experience will show what people expect in this area.

Omissions

We believe that this design covers the basic requirements for generic programming. However, there are a number of programming constructs that are not supported.

* No specialization. There is no way to write multiple versions of a generic function that are designed to work with specific type arguments. * No metaprogramming. There is no way to write code that is executed at compile time to generate code to be executed at run time. * No higher level abstraction. There is no way to use a function with type arguments other than to call it or instantiate it. There is no way to use a generic type other than to instantiate it. * No general type description. In order to use operators in a generic function, constraints list specific types, rather than describing the characteristics that a type must have. This is easy to understand but may be limiting at times. * No covariance or contravariance of function parameters. * No operator methods. You can write a generic container that is compile-time type-safe, but you can only access it with ordinary methods, not with syntax like 'c[k]'. * No currying. There is no way to partially instantiate a generic function or type, other than by using a helper function or a wrapper type. All type arguments must be either explicitly passed or inferred at instantiation time. * No variadic type

parameters. There is no support for variadic type parameters, which would permit writing a single generic function that takes different numbers of both type parameters and regular parameters. * No adaptors. There is no way for a constraint to define adaptors that could be used to support type arguments that do not already implement the constraint, such as, for example, defining an '==' operator in terms of an 'Equal' method, or vice-versa. * No parameterization on non-type values such as constants. This arises most obviously for arrays, where it might sometimes be convenient to write 'type Matrix[n int] [n][n]float64'. It might also sometimes be useful to specify significant values for a container type, such as a default value for elements.

Issues

There are some issues with this design that deserve a more detailed discussion. We think these issues are relatively minor compared to the design as a whole, but they still deserve a complete hearing and discussion.

The zero value

This design has no simple expression for the zero value of a type parameter. For example, consider this implementation of optional values that uses pointers:

```
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
}
```

In the case where 'o.p == nil', we want to return the zero value of 'T', but we have no way to write that. It would be nice to be able to write 'return nil', but that wouldn't work if 'T' is, say, 'int'; in that case we would have to write 'return 0'. And, of course, there is no way to write a constraint to support either 'return nil' or 'return 0'.

Some approaches to this are:

* Use 'var zero T', as above, which works with the existing design but requires an extra statement. * Use '*new(T)', which is cryptic but works with the existing design. * For results only, name the result parameter, and use a naked 'return' statement to return the zero value. * Extend the design to permit using 'nil' as the zero value of any generic type (but see [issue 22729](https://golang.org/issue/22729)). * Extend the design to permit

using ‘T’, where ‘T‘ is a type parameter, to indicate the zero value of the type. * Change the language to permit using _ on the right hand of an assignment (including ‘return‘ or a function call) as proposed in [issue 19642](https://golang.org/issue/19642). * Change the language to permit ‘return ...‘ to return zero values of the result types, as proposed in [issue 21182](https://golang.org/issue/21182).

We feel that more experience with this design is needed before deciding what, if anything, to do here.

Identifying the matched predeclared type

The design doesn’t provide any way to test the underlying type matched by a type argument. Code can test the actual type argument through the somewhat awkward approach of converting to an empty interface type and using a type assertion or a type switch. But that lets code test the actual type argument, which is not the same as the underlying type.

Here is an example that shows the difference.

```
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))
    }
    // Code omitted.
}

type MyFloat float32

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

This code will panic when initializing ‘G‘, because the type of ‘v‘ in the ‘NewtonSqrt‘ function will be ‘MyFloat‘, not ‘float32‘ or ‘float64‘. What this function actually wants to test is not the type of ‘v‘, but the type that ‘v‘ matched in the constraint.

One way to handle this would be to permit type switches on the type ‘T‘, with the proviso that the type ‘T‘ would always match a type defined in the constraint. This kind of type switch would only be permitted if the constraint lists explicit types, and only types listed

in the constraint would be permitted as cases.

No way to express convertibility

The design has no way to express convertibility between two different type parameters. For example, there is no way to write this function:

```
// Copy copies values from src to dst, converting them as they go.
// It returns the number of items copied, which is the minimum of
// the lengths of dst and src.
// This implementation is INVALID.
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) // INVALID
    }
    return len(src)
}
```

The conversion from type ‘T2‘ to type ‘T1‘ is invalid, as there is no constraint on either type that permits the conversion. Worse, there is no way to write such a constraint in general. In the particular case where both ‘T1‘ and ‘T2‘ can require some type list, then this function can be written as described earlier when discussing [type conversions using type lists](#Type-conversions). But, for example, there is no way to write a constraint for the case in which ‘T1‘ is an interface type and ‘T2‘ is a type that implements that interface.

It’s worth noting that if ‘T1‘ is an interface type then this can be written using a conversion to the empty interface type and a type assertion, but this is, of course, not compile-time type-safe.

```
// Copy copies values from src to dst, converting them as they go.
// It returns the number of items copied, which is the minimum of
// the lengths of dst and src.
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)
}
```

No parameterized methods

This design does not permit methods to declare type parameters that are specific to the method. The receiver may have type parameters, but the method may not add any type parameters.

In Go, one of the main roles of methods is to permit types to implement interfaces. It is not clear whether it is reasonably possible to permit parameterized methods to implement interfaces. For example, consider this code, which uses the obvious syntax for parameterized methods. This code uses multiple packages to make the problem clearer.

```
package p1

// S is a type with a parameterized method Identity.
type S struct{}

// Identity is a simple identity method that works for any type.
func (S) Identity[T any](v T) T { return v }

package p2

// HasIdentity is an interface that matches any type with a
// parameterized Identity method.
type HasIdentity interface {
    Identity[T any](T) T
}

package p3

import "p2"

// CheckIdentity checks the Identity method if it exists.
// Note that although this function calls a parameterized method,
// this function is not itself parameterized.
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"
```

```
    "p3"
)

// CheckSIdentity passes an S value to CheckIdentity.
func CheckSIdentity() {
    p3.CheckIdentity(p1.S{})
}
```

In this example, we have a type ‘p1.S‘ with a parameterized method and a type ‘p2.HasIdentity‘ that also has a parameterized method. ‘p1.S‘ implements ‘p2.HasIdentity‘. Therefore, the function ‘p3.CheckIdentity‘ can call ‘vi.Identity‘ with an ‘int‘ argument, which in the call from ‘p4.CheckSIdentity‘ will be a call to ‘p1.S.Identity[int]‘. But package p3 does not know anything about the type ‘p1.S‘. There may be no other call to ‘p1.S.Identity‘ elsewhere in the program. We need to instantiate ‘p1.S.Identity[int]‘ somewhere, but how?

We could instantiate it at link time, but in the general case that requires the linker to traverse the complete call graph of the program to determine the set of types that might be passed to ‘CheckIdentity‘. And even that traversal is not sufficient in the general case when type reflection gets involved, as reflection might look up methods based on strings input by the user. So in general instantiating parameterized methods in the linker might require instantiating every parameterized method for every possible type argument, which seems untenable.

Or, we could instantiate it at run time. In general this means using some sort of JIT, or compiling the code to use some sort of reflection based approach. Either approach would be very complex to implement, and would be surprisingly slow at run time.

Or, we could decide that parameterized methods do not, in fact, implement interfaces, but then it’s much less clear why we need methods at all. If we disregard interfaces, any parameterized method can be implemented as a parameterized function.

So while parameterized methods seem clearly useful at first glance, we would have to decide what they mean and how to implement that.

No way to require pointer methods

In some cases a parameterized function is naturally written such that it always invokes methods on addressable values. For example, this happens when calling a method on each element of a slice. In such a case, the function only requires that the method be in the slice element type’s pointer method set. The type constraints described in this design have no way to write that requirement.

For example, consider a variant of the ‘Stringify‘ example we [showed earlier](#Using-a-constraint).

```
// Stringify2 calls the String method on each element of s,
// and returns the results.
func Stringify2[T Stringer](s []T) (ret []string) {
    for i := range s {
        ret = append(ret, s[i].String())
    }
    return ret
}
```

Suppose we have a ‘[]bytes.Buffer‘ and we want to convert it into a ‘[]string‘. The ‘Stringify2‘ function here won’t help us. We want to write ‘Stringify2[bytes.Buffer]‘, but we can’t, because ‘bytes.Buffer‘ doesn’t have a ‘String‘ method. The type that has a ‘String‘ method is ‘*bytes.Buffer‘. Writing ‘Stringify2[*bytes.Buffer]‘ doesn’t help because that function expects a ‘[]*bytes.Buffer‘, but we have a ‘[]bytes.Buffer‘.

We discussed a similar case in the [pointer method example](#Pointer-method-example) above. There we used constraint type inference to help simplify the problem. Here that doesn’t help, because ‘Stringify2‘ doesn’t really care about calling a pointer method. It just wants a type that has a ‘String‘ method, and it’s OK if the method is only in the pointer method set, not the value method set. But we also want to accept the case where the method is in the value method set, for example if we really do have a ‘[]*bytes.Buffer‘.

What we need is a way to say that the type constraint applies to either the pointer method set or the value method set. The body of the function would be required to only call the method on addressable values of the type.

It’s not clear how often this problem comes up in practice.

No association between float and complex

Constraint type inference lets us give a name to the element of a slice type, and to apply other similar type decompositions. However, there is no way to associate a float type and a complex type. For example, there is no way to write the predeclared ‘real‘, ‘imag‘, or ‘complex‘ functions with this design There is no way to say ”if the argument type is ‘complex64‘, then the result type is ‘float32‘.”

One possible approach here would be to permit ‘real(T)‘ as a type constraint meaning ”the float type associated with the complex type ‘T‘”. Similarly, ‘complex(T)‘ would mean ”the complex type associated with the floating point type ‘T‘”. Constraint type inference would simplify the call site. However, that would be unlike other type constraints.

Discarded ideas

This design is not perfect, and there may be ways to improve it. That said, there are many ideas that we’ve already considered in detail. This section lists some of those ideas

in the hopes that it will help to reduce repetitive discussion. The ideas are presented in the form of a FAQ.

What happened to contracts?

An earlier draft design of generics implemented constraints using a new language construct called contracts. Type lists appeared only in contracts, rather than on interface types. However, many people had a hard time understanding the difference between contracts and interface types. It also turned out that contracts could be represented as a set of corresponding interfaces; there was no loss in expressive power without contracts. We decided to simplify the approach to use only interface types.

Why not use methods instead of type lists?

Type lists are weird. Why not write methods for all operators?

It is possible to permit operator tokens as method names, leading to methods such as '+(T) T'. Unfortunately, that is not sufficient. We would need some mechanism to describe a type that matches any integer type, for operations such as shifts '¡¡(integer) T' and indexing '[](integer) T' which are not restricted to a single int type. We would need an untyped boolean type for operations such as '==(T) untyped bool'. We would need to introduce new notation for operations such as conversions, or to express that one may range over a type, which would likely require some new syntax. We would need some mechanism to describe valid values of untyped constants. We would have to consider whether support for '¡(T) bool' means that a generic function can also use '¡=', and similarly whether support for '+(T) T' means that a function can also use '++'. It might be possible to make this approach work but it's not straightforward. The approach used in this design seems simpler and relies on only one new syntactic construct (type lists) and one new name ('comparable').

Why not put type parameters on packages?

We investigated this extensively. It becomes problematic when you want to write a 'list' package, and you want that package to include a 'Transform' function that converts a 'List' of one element type to a 'List' of another element type. It's very awkward for a function in one instantiation of a package to return a type that requires a different instantiation of the same package.

It also confuses package boundaries with type definitions. There is no particular reason to think that the uses of generic types will break down neatly into packages. Sometimes they will, sometimes they won't.

Why not use the syntax ‘F¡T¿‘ like C++ and Java?

When parsing code within a function, such as ‘v := F¡T¿‘, at the point of seeing the ‘¡‘ it’s ambiguous whether we are seeing a type instantiation or an expression using the ‘¡‘ operator. This is very difficult to resolve without type information.

For example, consider a statement like

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

Without type information, it is impossible to decide whether the right hand side of the assignment is a pair of expressions (‘w ¡ x‘ and ‘y ¿ (z)‘), or whether it is a generic function instantiation and call that returns two result values (‘(w¡x, y¿)(z)‘).

It is a key design decision of Go that parsing be possible without type information, which seems impossible when using angle brackets for generics.

Why not use the syntax ‘F(T)‘?

An earlier version of this design used that syntax. It was workable but it introduced several parsing ambiguities. For example, when writing ‘var f func(x(T))‘ it wasn’t clear whether this the type was a function with a single unnamed parameter of the instantiated type ‘x(T)‘ or whether it was a function with a parameter named ‘x‘ with type ‘(T)‘ (more usually written as ‘func(x T)‘, but in this case with a parenthesized type).

There were other ambiguities as well. For ‘[]T(v1)‘ and ‘[]T(v2)‘, at the point of the open parentheses we don’t know whether this is a type conversion (of the value ‘v1‘ to the type ‘[]T‘) or a type literal (whose type is the instantiated type ‘T(v2)‘). For ‘interface M(T) ‘ we don’t know whether this an interface with a method ‘M‘ or an interface with an embedded instantiated interface ‘M(T)‘. These ambiguities are solvable, by adding more parentheses, but awkward.

Also some people were troubled by the number of parenthesized lists involved in declarations like ‘func F(T any)(v T)(r1, r2 T)‘ or in calls like ‘F(int)(1)‘.

Why not use ‘F « T » ‘?

We considered it but we couldn’t bring ourselves to require non-ASCII.

Why not define constraints in a builtin package?

Instead of writing out type lists, use names like ‘constraints.Arithmetic‘ and ‘constraints.Comparable‘.

Listing all the possible combinations of types gets rather lengthy. It also introduces a new set of names that not only the writer of generic code, but, more importantly, the reader, must remember. One of the driving goals of this design is to introduce as few new names as possible. In this design we introduce only two new predeclared names, ‘comparable‘

and ‘any‘.

We expect that if people find such names useful, we can introduce a package ‘constraints‘ that defines those names in the form of constraints that can be used by other types and functions and embedded in other constraints. That will define the most useful names in the standard library while giving programmers the flexibility to use other combinations of types where appropriate.

Why not permit type assertions on values whose type is a type parameter?

In an earlier version of this design, we permitted using type assertions and type switches on variables whose type was a type parameter, or whose type was based on a type parameter. We removed this facility because it is always possible to convert a value of any type to the empty interface type, and then use a type assertion or type switch on that. Also, it was sometimes confusing that in a constraint with a type list, a type assertion or type switch would use the actual type argument, not the underlying type of the type argument (the difference is explained in the section on [identifying the matched predeclared type](#Identifying-the-matched-predeclared-type)).

Comparison with Java

Most complaints about Java generics center around type erasure. This design does not have type erasure. The reflection information for a generic type will include the full compile-time type information.

In Java type wildcards (‘List¡? extends Number¿‘, ‘List¡? super Number¿‘) implement covariance and contravariance. These concepts are missing from Go, which makes generic types much simpler.

Comparison with C++

C++ templates do not enforce any constraints on the type arguments (unless the concept proposal is adopted). This means that changing template code can accidentally break far-off instantiations. It also means that error messages are reported only at instantiation time, and can be deeply nested and difficult to understand. This design avoids these problems through mandatory and explicit constraints.

C++ supports template metaprogramming, which can be thought of as ordinary programming done at compile time using a syntax that is completely different than that of non-template C++. This design has no similar feature. This saves considerable complexity while losing some power and run time efficiency.

C++ uses two-phase name lookup, in which some names are looked up in the context of the template definition, and some names are looked up in the context of the template instantiation. In this design all names are looked up at the point where they are written.

In practice, all C++ compilers compile each template at the point where it is instantiated. This can slow down compilation time. This design offers flexibility as to how to handle the compilation of generic functions.

Comparison with Rust

The generics described in this design are similar to generics in Rust.

One difference is that in Rust the association between a trait bound and a type must be defined explicitly, either in the crate that defines the trait bound or the crate that defines the type. In Go terms, this would mean that we would have to declare somewhere whether a type satisfied a constraint. Just as Go types can satisfy Go interfaces without an explicit declaration, in this design Go type arguments can satisfy a constraint without an explicit declaration.

Where this design uses type lists, the Rust standard library defines standard traits for operations like comparison. These standard traits are automatically implemented by Rust's primitive types, and can be implemented by user defined types as well. Rust provides a fairly extensive list of traits, at least 34, covering all of the operators.

Rust supports type parameters on methods, which this design does not.

第2章

コード例

The following sections are examples of how this design could be used. This is intended to address specific areas where people have created user experience reports concerned with Go's lack of generics.

2.1 Map/Reduce/Filter

Here is an example of how to write map, reduce, and filter functions for slices. These functions are intended to correspond to the similar functions in Lisp, Python, Java, and so forth.

```
// Package slices implements various slice algorithms.
package slices

// Map turns a []T1 to a []T2 using a mapping function.
// This function has two type parameters, T1 and T2.
// This works with slices of any type.
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 reduces a []T1 to a single value using a reduction function.
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 filters values from a slice using a filter function.
// It returns a new slice with only the elements of s
```

```
// for which f returned true.
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
}
```

Here are some example calls of these functions. Type inference is used to determine the type arguments based on the types of the non-type arguments.

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

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

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

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

2.2　Map keys

Here is how to get a slice of the keys of any map.

```
// Package maps provides general functions that work for all map types.
package maps

// Keys returns the keys of the map m in a slice.
// The keys will be returned in an unpredictable order.
// This function has two type parameters, K and V.
// Map keys must be comparable, so key has the predeclared
// constraint comparable. Map values can be any type.
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
}
```

In typical use the map key and val types will be inferred.

```
    k := maps.Keys(map[int]int{1:2, 2:4})
    // Now k is either []int{1, 2} or []int{2, 1}.
```

2.3 Sets

Many people have asked for Go's builtin map type to be extended, or rather reduced, to support a set type. Here is a type-safe implementation of a set type, albeit one that uses methods rather than operators like '[]'.

```
// Package sets implements sets of any comparable type.
package sets

// Set is a set of values.
type Set[T comparable] map[T]struct{}

// Make returns a set of some element type.
func Make[T comparable]() Set[T] {
    return make(Set[T])
}

// Add adds v to the set s.
// If v is already in s this has no effect.
func (s Set[T]) Add(v T) {
    s[v] = struct{}{}
}

// Delete removes v from the set s.
// If v is not in s this has no effect.
func (s Set[T]) Delete(v T) {
    delete(s, v)
}

// Contains reports whether v is in s.
func (s Set[T]) Contains(v T) bool {
    _, ok := s[v]
    return ok
}

// Len reports the number of elements in s.
func (s Set[T]) Len() int {
    return len(s)
}
```

```
// Iterate invokes f on each element of s.
// It's OK for f to call the Delete method.
func (s Set[T]) Iterate(f func(T)) {
    for v := range s {
        f(v)
    }
}
```

Example use:

```
    // Create a set of ints.
    // We pass int as a type argument.
    // Then we write () because Make does not take any non-type arguments.
    // We have to pass an explicit type argument to Make.
    // Function argument type inference doesn't work because the
    // type argument to Make is only used for a result parameter type.
    s := sets.Make[int]()

    // Add the value 1 to the set s.
    s.Add(1)

    // Check that s does not contain the value 2.
    if s.Contains(2) { panic("unexpected 2") }
```

This example shows how to use this design to provide a compile-time type-safe wrapper around an existing API.

2.4　Sort

Before the introduction of ‘sort.Slice‘, a common complaint was the need for boilerplate definitions in order to use ‘sort.Sort‘. With this design, we can add to the sort package as follows:

```
// Ordered is a type constraint that matches all ordered types.
// (An ordered type is one that supports the < <= >= > operators.)
// In practice this type constraint would likely be defined in
// a standard library package.
type Ordered interface {
    type int, int8, int16, int32, int64,
        uint, uint8, uint16, uint32, uint64, uintptr,
        float32, float64,
        string
}
```

```
// orderedSlice is an internal type that implements sort.Interface.
// The Less method uses the < operator. The Ordered type constraint
// ensures that T has a < operator.
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 sorts the slice s in ascending order.
// The elements of s must be ordered using the < operator.
func OrderedSlice[T Ordered](s []T) {
    // Convert s to the type orderedSlice[T].
    // As s is []T, and orderedSlice[T] is defined as []T,
    // this conversion is permitted.
    // orderedSlice[T] implements sort.Interface,
    // so can pass the result to sort.Sort.
    // The elements will be sorted using the < operator.
    sort.Sort(orderedSlice[T](s))
}
```

Now we can write:

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

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

Along the same lines, we can add a function for sorting using a comparison function, similar to ‘sort.Slice‘ but writing the function to take values rather than slice indexes.

```
// sliceFn is an internal type that implements sort.Interface.
// The Less method calls the cmp field.
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 sorts the slice s according to the function cmp.
```

```
func SliceFn[T any](s []T, cmp func(T, T) bool) {
    sort.Sort(sliceFn[E]{s, cmp})
}
```

An example of calling this might be:

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

2.5　Channels

Many simple general purpose channel functions are never written, because they must be written using reflection and the caller must type assert the results. With this design they become straightforward to write.

```
// Package chans implements various channel algorithms.
package chans

import "runtime"

// Drain drains any elements remaining on the channel.
func Drain[T any](c <-chan T) {
    for range c {
    }
}

// Merge merges two channels of some element type into a single channel.
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 provides a convenient way to exit a goroutine sending values
// when the receiver stops reading them.
//
// Ranger returns a Sender and a Receiver. The Receiver provides a
// Next method to retrieve values. The Sender provides a Send method
// to send values and a Close method to stop sending values. The Next
// method indicates when the Sender has been closed, and the Send
// method indicates when the Receiver has been freed.
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}
    // The finalizer on the receiver will tell the sender
    // if the receiver stops listening.
    runtime.SetFinalizer(r, r.finalize)
    return s, r
}

// A Sender is used to send values to a Receiver.
type Sender[T any] struct {
    values chan<- T
    done   <-chan bool
}

// Send sends a value to the receiver. It reports whether any more
// values may be sent; if it returns false the value was not sent.
func (s *Sender[T]) Send(v T) bool {
    select {
    case s.values <- v:
        return true
    case <-s.done:
        // The receiver has stopped listening.
        return false
    }
}

// Close tells the receiver that no more values will arrive.
// After Close is called, the Sender may no longer be used.
```

```
func (s *Sender[T]) Close() {
    close(s.values)
}

// A Receiver receives values from a Sender.
type Receiver[T any] struct {
    values <-chan T
    done  chan<- bool
}

// Next returns the next value from the channel. The bool result
// reports whether the value is valid. If the value is not valid, the
// Sender has been closed and no more values will be received.
func (r *Receiver[T]) Next() (T, bool) {
    v, ok := <-r.values
    return v, ok
}

// finalize is a finalizer for the receiver.
// It tells the sender that the receiver has stopped listening.
func (r *Receiver[T]) finalize() {
    close(r.done)
}
```

There is an example of using this function in the next section.

2.6 Containers

One of the frequent requests for generics in Go is the ability to write compile-time type-safe containers. This design makes it easy to write a compile-time type-safe wrapper around an existing container; we won't write out an example for that. This design also makes it easy to write a compile-time type-safe container that does not use boxing.

Here is an example of an ordered map implemented as a binary tree. The details of how it works are not too important. The important points are:

* The code is written in a natural Go style, using the key and value types where needed.
* The keys and values are stored directly in the nodes of the tree, not using pointers and not boxed as interface values.

```
// Package orderedmaps provides an ordered map, implemented as a binary tree.
package orderedmaps

import "chans"

// Map is an ordered map.
```

```
type Map[K, V any] struct {
    root    *node[K, V]
    compare func(K, K) int
}

// node is the type of a node in the binary tree.
type node[K, V any] struct {
    k           K
    v           V
    left, right *node[K, V]
}

// New returns a new map.
// Since the type parameter V is only used for the result,
// type inference does not work, and calls to New must always
// pass explicit type arguments.
func New[K, V any](compare func(K, K) int) *Map[K, V] {
    return &Map[K, V]{compare: compare}
}

// find looks up k in the map, and returns either a pointer
// to the node holding k, or a pointer to the location where
// such a node would go.
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 inserts a new key/value into the map.
// If the key is already present, the value is replaced.
// Reports whether this is a new key.
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 returns the value associated with a key, or zero if not present.
// The bool result reports whether the key was found.
func (m *Map[K, V]) Find(k K) (V, bool) {
    pn := m.find(k)
    if *pn == nil {
        var zero V // see the discussion of zero values, above
        return zero, false
    }
    return (*pn).v, true
}

// keyValue is a pair of key and value used when iterating.
type keyValue[K, V any] struct {
    k K
    v V
}

// InOrder returns an iterator that does an in-order traversal of the map.
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
        }
        // Stop sending values if sender.Send returns false,
        // meaning that nothing is listening at the receiver end.
        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 is used to iterate over the map.
type Iterator[K, V any] struct {
    r *chans.Receiver[keyValue[K, V]]
}
```

```
// Next returns the next key and value pair. The bool result reports
// whether the values are valid. If the values are not valid, we have
// reached the end.
func (it *Iterator[K, V]) Next() (K, V, bool) {
    kv, ok := it.r.Next()
    return kv.k, kv.v, ok
}
```

This is what it looks like to use this package:

```
import "container/orderedmaps"

// Set m to an ordered map from string to string,
// using strings.Compare as the comparison function.
var m = orderedmaps.New[string, string](strings.Compare)

// Add adds the pair a, b to m.
func Add(a, b string) {
    m.Insert(a, b)
}
```

2.7 Append

The predeclared ‘append‘ function exists to replace the boilerplate otherwise required to grow a slice. Before ‘append‘ was added to the language, there was a function ‘Add‘ in the bytes package:

```
// Add appends the contents of t to the end of s and returns the result.
// If s has enough capacity, it is extended in place; otherwise a
// new array is allocated and returned.
func Add(s, t []byte) []byte
```

‘Add‘ appended two ‘[]byte‘ values together, returning a new slice. That was fine for ‘[]byte‘, but if you had a slice of some other type, you had to write essentially the same code to append more values. If this design were available back then, perhaps we would not have added ‘append‘ to the language. Instead, we could write something like this:

```
// Package slices implements various slice algorithms.
package slices

// Append appends the contents of t to the end of s and returns the result.
// If s has enough capacity, it is extended in place; otherwise a
```

```
// new array is allocated and returned.
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
}
```

That example uses the predeclared ‘copy‘ function, but that’s OK, we can write that one too:

```
// Copy copies values from t to s, stopping when either slice is
// full, returning the number of values copied.
func Copy[T any](s, t []T) int {
    i := 0
    for ; i < len(s) && i < len(t); i++ {
        s[i] = t[i]
    }
    return i
}
```

These functions can be used as one would expect:

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

This code doesn’t implement the special case of appending or copying a ‘string‘ to a ‘[]byte‘, and it’s unlikely to be as efficient as the implementation of the predeclared function. Still, this example shows that using this design would permit ‘append‘ and ‘copy‘ to be written generically, once, without requiring any additional special language features.

2.8 Metrics

In a [Go experience report](https://medium.com/@sameer_74231/go-experience-report-for-generics-google-metrics-api-b019d597aaa4) Sameer Ajmani describes a metrics implementation. Each metric has a value and one or more fields. The fields have different types. Defining a metric requires specifying the types of the fields. The 'Add' method takes the field types as arguments, and records an instance of that set of fields. The C++ implementation uses a variadic template. The Java implementation includes the number of fields in the name of the type. Both the C++ and Java implementations provide compile-time type-safe Add methods.

Here is how to use this design to provide similar functionality in Go with a compile-time type-safe 'Add' method. Because there is no support for a variadic number of type arguments, we must use different names for a different number of arguments, as in Java. This implementation only works for comparable types. A more complex implementation could accept a comparison function to work with arbitrary types.

```
// Package metrics provides a general mechanism for accumulating
// metrics of different values.
package metrics

import "sync"

// Metric1 accumulates metrics of a single value.
type Metric1[T comparable] struct {
    mu sync.Mutex
    m  map[T]int
}

// Add adds an instance of a value.
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 is an internal type used by Metric2.
type key2[T1, T2 comparable] struct {
    f1 T1
    f2 T2
```

```
}

// Metric2 accumulates metrics of pairs of values.
type Metric2[T1, T2 comparable] struct {
    mu sync.Mutex
    m  map[key2[T1, T2]]int
}

// Add adds an instance of a value pair.
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 is an internal type used by Metric3.
type key3[T1, T2, T3 comparable] struct {
    f1 T1
    f2 T2
    f3 T3
}

// Metric3 accumulates metrics of triples of values.
type Metric3[T1, T2, T3 comparable] struct {
    mu sync.Mutex
    m  map[key3[T1, T2, T3]]int
}

// Add adds an instance of a value triplet.
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}]++
}

// Repeat for the maximum number of permitted arguments.
```

Using this package looks like this:

```
import "metrics"

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

func F(s string, i int) {
    m.Add(s, i) // this call is type checked at compile time
}
```

This implementation has a certain amount of repetition due to the lack of support for variadic type parameters. Using the package, though, is easy and type safe.

2.9 List transform

While slices are efficient and easy to use, there are occasional cases where a linked list is appropriate. This example primarily shows transforming a linked list of one type to another type, as an example of using different instantiations of the same generic type.

```
// Package lists provides a linked list of any type.
package lists

// List is a linked list.
type List[T any] struct {
    head, tail *element[T]
}

// An element is an entry in a linked list.
type element[T any] struct {
    next *element[T]
    val  T
}

// Push pushes an element to the end of the list.
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 ranges over a list.
type Iterator[T any] struct {
    next **element[T]
```

```
}

// Range returns an Iterator starting at the head of the list.
func (lst *List[T]) Range() *Iterator[T] {
    return Iterator[T]{next: &lst.head}
}

// Next advances the iterator.
// It reports whether there are more elements.
func (it *Iterator[T]) Next() bool {
    if *it.next == nil {
        return false
    }
    it.next = &(*it.next).next
    return true
}

// Val returns the value of the current element.
// The bool result reports whether the value is valid.
func (it *Iterator[T]) Val() (T, bool) {
    if *it.next == nil {
        var zero T
        return zero, false
    }
    return (*it.next).val, true
}

// Transform runs a transform function on a list returning a new list.
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　Dot product

A generic dot product implementation that works for slices of any numeric type.

```
// Numeric is a constraint that matches any numeric type.
// It would likely be in a constraints package in the standard library.
type Numeric interface {
    type int, int8, int16, int32, int64,
        uint, uint8, uint16, uint32, uint64, uintptr,
        float32, float64,
        complex64, complex128
}

// DotProduct returns the dot product of two slices.
// This panics if the two slices are not the same length.
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
}
```

(Note: the generics implementation approach may affect whether 'DotProduct' uses FMA, and thus what the exact results are when using floating point types. It's not clear how much of a problem this is, or whether there is any way to fix it.)

2.11 Absolute difference

Compute the absolute difference between two numeric values, by using an 'Abs' method. This uses the same 'Numeric' constraint defined in the last example.

This example uses more machinery than is appropriate for the simple case of computing the absolute difference. It is intended to show how the common part of algorithms can be factored into code that uses methods, where the exact definition of the methods can vary based on the kind of type being used.

```
// NumericAbs matches numeric types with an Abs method.
type NumericAbs[T any] interface {
    type int, int8, int16, int32, int64,
        uint, uint8, uint16, uint32, uint64, uintptr,
        float32, float64,
        complex64, complex128
    Abs() T
}
```

```
// AbsDifference computes the absolute value of the difference of
// a and b, where the absolute value is determined by the Abs method.
func AbsDifference[T NumericAbs](a, b T) T {
    d := a - b
    return d.Abs()
}
```

We can define an ‘Abs‘ method appropriate for different numeric types.

```
// OrderedNumeric matches numeric types that support the < operator.
type OrderedNumeric interface {
    type int, int8, int16, int32, int64,
        uint, uint8, uint16, uint32, uint64, uintptr,
        float32, float64
}

// Complex matches the two complex types, which do not have a < operator.
type Complex interface {
    type complex64, complex128
}

// OrderedAbs is a helper type that defines an Abs method for
// ordered numeric types.
type OrderedAbs[T OrderedNumeric] T

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

// ComplexAbs is a helper type that defines an Abs method for
// complex types.
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))
}
```

We can then define functions that do the work for the caller by converting to and from the types we just defined.

```
// OrderedAbsDifference returns the absolute value of the difference
// between a and b, where a and b are of an ordered type.
func OrderedAbsDifference[T OrderedNumeric](a, b T) T {
    return T(AbsDifference(OrderedAbs[T](a), OrderedAbs[T](b)))
}

// ComplexAbsDifference returns the absolute value of the difference
// between a and b, where a and b are of a complex type.
func ComplexAbsDifference[T Complex](a, b T) T {
    return T(AbsDifference(ComplexAbs[T](a), ComplexAbs[T](b)))
}
```

It's worth noting that this design is not powerful enough to write code like the following:

```
// 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) // INVALID
    case complex64, complex128:
        return ComplexAbsDifference(a, b) // INVALID
    }
}
```

The calls to 'OrderedAbsDifference' and 'ComplexAbsDifference' are invalid, because not all the types that implement the 'Numeric' constraint can implement the 'OrderedNumeric' or 'Complex' constraints. Although the type switch means that this code would conceptually work at run time, there is no support for writing this code at compile time. This is another way of expressing one of the omissions listed above: this design does not provide for specialization.

謝辞

We'd like to thank many people on the Go team, many contributors to the Go issue tracker, and all the people who have shared their ideas and their feedback on earlier design drafts. We read all of it, and we're grateful.

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.

付録 A

細かな詳細

This appendix covers various details of the design that don't seem significant enough to cover in earlier sections.

A.1 Generic type aliases

A type alias may refer to a generic type, but the type alias may not have its own parameters. This restriction exists because it is unclear how to handle a type alias with type parameters that have constraints.

```
type VectorAlias = Vector
```

In this case uses of the type alias will have to provide type arguments appropriate for the generic type being aliased.

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

Type aliases may also refer to instantiated types.

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

A.2 Instantiating a function

Go normally permits you to refer to a function without passing any arguments, producing a value of function type. You may not do this with a function that has type parameters; all type arguments must be known at compile time. That said, you can instantiate the function, by passing type arguments, but you don't have to call the instantiation. This will produce a function value with no type parameters.

```
// PrintInts is type func([]int).
var PrintInts = Print[int]
```

A.3 Embedded type parameter

When a generic type is a struct, and the type parameter is embedded as a field in the struct, the name of the field is the name of the type parameter.

```
// A Lockable is a value that may be safely simultaneously accessed
// from multiple goroutines via the Get and Set methods.
type Lockable[T any] struct {
    T
    mu sync.Mutex
}

// Get returns the value stored in a Lockable.
func (l *Lockable[T]) Get() T {
    l.mu.Lock()
    defer l.mu.Unlock()
    return l.T
}

// Set sets the value in a Lockable.
func (l *Lockable[T]) Set(v T) {
    l.mu.Lock()
    defer l.mu.Unlock()
    l.T = v
}
```

A.4 Embedded type parameter methods

When a generic type is a struct, and the type parameter is embedded as a field in the struct, any methods of the type parameter's constraint are promoted to be methods of the struct. (For purposes of [selector resolution](https://golang.org/ref/spec#Selectors), these methods are treated as being at depth 0 of the type parameter, even if in the actual type argument the methods were themselves promoted from an embedded type.)

```
// NamedInt is an int with a name. The name can be any type with
// a String method.
type NamedInt[Name fmt.Stringer] struct {
    Name
    val int
}

// Name returns the name of a NamedInt.
```

```
func (ni NamedInt[Name]) Name() string {
    // The String method is promoted from the embedded Name.
    return ni.String()
}
```

A.5 Embedded instantiated type

When embedding an instantiated type, the name of the field is the name of type without the type arguments.

```
type S struct {
    T[int] // field name is T
}

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

A.6 Generic types as type switch cases

A generic type may be used as the type in a type assertion or as a case in a type switch. Here are some trivial examples:

```
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
    }
}
```

In a type switch, it's OK if a generic type turns out to duplicate some other case in the type switch. The first matching case is chosen.

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

// S2a will be set to 0.
var S2a = Switch2[string]("a string")

// S2b will be set to 1.
var S2b = Switch2[int]("another string")
```

A.7　Type inference for composite literals

This is a feature we are not suggesting now, but could consider for later versions of the language.

We could also consider supporting type inference for composite literals of generic types.

```
type Pair[T any] struct { f1, f2 T }
var V = Pair{1, 2} // inferred as Pair(int){1, 2}
```

It's not clear how often this will arise in real code.

A.8　Type inference for generic function arguments

This is a feature we are not suggesting now, but could consider for later versions of the language.

In the following example, consider the call to 'Find' in 'FindClose'. Type inference can determine that the type argument to 'Find' is 'T4', and from that we know that the type of the final argument must be 'func(T4, T4) bool', and from that we could deduce that the type argument to 'IsClose' must also be 'T4'. However, the type inference algorithm described earlier cannot do that, so we must explicitly write 'IsClose[T4]'.

This may seem esoteric at first, but it comes up when passing generic functions to generic 'Map' and 'Filter' functions.

```
// Differ has a Diff method that returns how different a value is.
type Differ[T1 any] interface {
    Diff(T1) int
}

// IsClose returns whether a and b are close together, based on Diff.
func IsClose[T2 Differ](a, b T2) bool {
    return a.Diff(b) < 2
}

// Find returns the index of the first element in s that matches e,
// based on the cmp function. It returns -1 if no element matches.
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 returns the index of the first element in s that is
// close to e, based on IsClose.
func FindClose[T4 Differ](s []T4, e T4) int {
    // With the current type inference algorithm we have to
    // explicitly write IsClose[T4] here, although it
    // is the only type argument we could possibly use.
    return Find(s, e, IsClose[T4])
}
```

A.9 Reflection on type arguments

Although we don't suggest changing the reflect package, one possibility to consider for the future would be to add two new methods to 'reflect.Type': 'NumTypeArgument() int' would return the number of type arguments to a type, and 'TypeArgument(i) Type' would return the i'th type argument. 'NumTypeArgument' would return non-zero for an instantiated generic type. Similar methods could be defined for 'reflect.Value', for which 'NumTypeArgument' would return non-zero for an instantiated generic function. There might be programs that care about this information.