# Summary of Go cycle detection rules

Dawid Lachowicz

January 6, 2025

## 1 Syntax

| | | | |
|---|---|---|---|
| Type | $\tau, \sigma ::=$ | Type name | $t, u$ |
| Type parameter | $\alpha$ | Type parameter | |
| Named type | $t[\overline{\tau}]$ | declaration | $\Phi ::= \alpha\ \gamma$ |
| | | Type constraint | $\gamma ::= t[\overline{\tau}]$ |
| Type Literal | $T ::=$ | | |
| Structure | $\textbf{struct}\ \{\overline{f\ \tau}\}$ | Method signature | $M ::= (\overline{x\ \tau})\ \tau$ |
| Interface | $\textbf{interface}\ \{\overline{S}\}$ | Interface element | $S ::= mM$ |
| Array | $[n]\tau$ | | |
| | | Field name | $f$ |
| Declaration | $D ::=$ | Method name | $m$ |
| Type declaration | $\textbf{type}\ t[\overline{\Phi}]\ T$ | Variable | $x$ |

Program $\quad P ::= \textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{\_ = e\}$

## 2 Typing rules

Well-formed declarations $\boxed{D\ ok}$

$$\frac{\text{T-Type}\quad \overline{\Phi\ ok} \qquad \overline{\Phi} \vdash\ T\ ok \qquad notContains(t,\ T)}{\textbf{type}\ t[\overline{\Phi}]\ T\ ok}$$

Programs $\boxed{P\ ok}$

$$\frac{\text{T-Prog}\quad distinct(tdecls(\overline{D}), \textbf{int}) \qquad distinct(mdecls(\overline{D})) \qquad \overline{D\ ok} \qquad \emptyset;\ \emptyset \vdash e : \tau}{\textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{\_ = e\}\ ok}$$

The remaining type checking rules (e.g. ones used in the examples) are provided in section 5. They are omitted here to focus the discussion on cycle detection.

# 3 Cycle detection rules

## 3.1 Type containment rules

Type declaration containment $\boxed{notContains(\overline{t}, T)}$

$$\frac{}{notContains(\overline{t_r},\ \textbf{interface}\ \{\overline{S}\})}$$

$$\frac{notContains(\overline{t_r}, \tau)}{notContains(\overline{t_r},\ [n]\tau)} \qquad \frac{\forall \tau \in \overline{f\ \tau}.notContains(\overline{t_r}, \tau)}{notContains(\overline{t_r},\ \textbf{struct}\{\overline{f\ \tau}\})}$$

Type declaration containment recursion $\boxed{notContains(\overline{t}, \tau)}$

$$\frac{}{notContains(\overline{t_r}, \textbf{int})} \qquad \frac{}{notContains(\overline{t_r}, \alpha)}$$

$$\frac{(\textbf{type}\ t[\overline{\Phi}]\ T) \in \overline{D} \qquad t \notin \overline{t_r} \qquad \eta = (\overline{\Phi := \tau}) \qquad notContains(\overline{t_r}, t, T[\![\eta]\!])}{notContains(\overline{t_r}, t[\overline{\tau}])}$$

- Type containment checks can be thought of as checking the structure of type literals, applicable to both generic and non-generic code.
- The *notContains* relation can be explained as "asserting a type/type literal doesn't contain any of the *already seen* types where they are not allowed to occur in the type's structure".
- As recursion progresses, the encountered type names $t$ are added to the set of already seen type names ($notContains(\overline{t_r}, t, T[\![\eta]\!])$). This is so we can detect indirect cycles without getting the algorithm into an infinite loop.
- Basic interfaces (and other pointer types) can refer to the type being defined in their structure, and therefore are base cases of the recursion.
- $notContains(\overline{t_r}, \textbf{int})$ can be more widely applied to primitive (predeclared) types in Go. Since primitive types may be redefined in a Go program, we'd need to check whether a type name (e.g. **int**) indeed still points at a primitive type.
- Type parameters shadow type names. Whenever a type parameter is encountered during the containment check, a base case of the recursion is reached.
- The $T[\![\eta]\!]$ notation indicates that the type literal has its type parameters $\alpha$ substituted with type arguments $\tau$.
- Note in particular, that these rules do not recurse on type parameter constraints of the checked types. Issue #65714 demonstrates a family of programs where the type checker sometimes rejects programs based on cycles mixing containment rules and type parameter constraint reference rules (described in the next section), depending on the ordering of type declarations.

## 3.2 Lifting type parameter reference rules

- The Go spec currently states: "Within a type parameter list of a generic type T, a type constraint may not (directly, or indirectly through the type parameter list of another generic type) refer to T." We can in fact life this restriction, i.e. allowing type parameter constraints to refer to the type T being defined.

- The next section contains a few examples of programs that either the the compiler currently struggles with, or are currently disallowed by the compiler due to the above-mentioned rule in the spec, but do not actually cause any problems during type-checking and could be allowed. Both positive (well-typed programs) and negative (badly-typed programs) examples are presented, to demonstrate that the rules presented in this document are sufficient for both cases.

- Some self-referential type parameters may be uninstantiable, however, Go already permits defining non-instantiable types, and defining empty type sets is also allowed. This is fine since types need to be explicitly instantiated, and checks also occur upon type instantiation. Type containment checks are critical however, since values can be instantiated implicitly via zero values, and so the type declaration (or instantiation in the case of generic types) must ensure that valid (non-infinite) zero values can be constructed.

```
type Empty interface {
    int
    string
}
type Uninstantiable[E Empty] struct{ x E }

func main() {}
```

# 4 Examples

## 4.1 Containment check in generic type

```
type Foo[T any] struct {
    x T
}

type Bar struct {
    x Foo[Bar]
}

func main() {
}
```

The definition of `Bar` should be rejected, since it creates a type containment cycle via `Foo`. The compiler correctly reports `Bar` as an invalid recursive type.

```
import "fmt"

type Foo[T Bar] struct {
    x *T
}

type Bar struct {
    x Foo[Bar]
}

func main() {
    x := Foo[Bar]{}
    y := Bar{x: x}
    fmt.Println(x)
    fmt.Println(y)
}
```

Using e.g. a pointer to `T` in the definition of `Foo` would make the program valid. We can even change the type constraint of `T` from `any` to `Bar`, and the type checker should behave the same (since `Bar` is a subtype of `any`). However, as of Go 1.23, such a program is either rejected or accepted by the type checker depending on the order of the struct declarations (similar to Issue #65714).

### 4.1.1 Type checking derivation with contradiction

Below, we derive a contradiction while type checking the above program on the left, showing that the *notContains* rule is sufficient for detecting structural cycles in generic types.

$$D_0 = \textbf{type}\ \text{any}\ \textbf{interface}\ \{\}$$
$$D_1 = \textbf{type}\ \text{Foo[T any]}\ \textbf{struct}\ \{x\ \text{T}\}$$
$$D_2 = \textbf{type}\ \text{Bar}\ \textbf{struct}\ \{x\ \text{Foo[Bar]}\}$$
$$\overline{D} = \{D_0, D_1, D_2\}$$

$$\frac{\dfrac{}{\emptyset \vdash \textbf{interface}\ \{\}\ ok}\ \textsc{T-Interface} \qquad \dfrac{}{notContains(\text{any}, \textbf{interface}\ \{\})}}{D_0\ ok}\ \textsc{T-Type}$$

$$\frac{\dfrac{}{D_0 \in \overline{\overline{D}}}}{(\text{T}\ \ \text{any}) \vdash \text{any}\ \ ok}\ \textsc{T-Named} \qquad \frac{\dfrac{}{distinct(\text{T})} \qquad (\text{T}\ \ \text{any}) \vdash \text{any}\ \ ok}{(\text{T}\ \ \text{any})\ ok}\ \textsc{T-Formal}$$

$$\frac{\dfrac{}{distinct(f)} \qquad \dfrac{\dfrac{}{(\text{T}\ \ \text{any}) \in (\text{T}\ \ \text{any})}}{(\text{T}\ \ \text{any}) \vdash \text{T}\ \ ok}\ \textsc{T-Param}}{(\text{T}\ \ \text{any}) \vdash \textbf{struct}\ \{x\ \ \text{T}\}\ ok}\ \textsc{T-Struct}$$

$$\frac{\dfrac{}{notContains(\text{Foo}, \text{T})}}{notContains(\text{Foo}, \textbf{struct}\ \{x\ \ \text{T}\})}$$

$$\frac{\dfrac{(\text{T}\ \ \text{any})\ ok}{(\text{T}\ \ \text{any}) \vdash \textbf{struct}\ \{x\ \ \text{T}\}\ ok} \qquad notContains(\text{Foo}, \textbf{struct}\ \{x\ \ \text{T}\})}{D_1\ ok}\ \textsc{T-Type}$$

$$\frac{D_2 \in \overline{D}}{\emptyset \vdash \text{Bar}\ \ ok}\ \textsc{T-Named} \qquad \frac{\eta = (\text{T} := \text{Bar})}{\eta = ((\text{T}\ \ \text{any}) := \text{Bar})}$$

$$\frac{\dfrac{}{methods_\emptyset(\text{Bar}) \supseteq methods_\emptyset(\text{any})}}{\emptyset \vdash (\text{Bar} <: \text{any})}\ <:_{\text{I}} \qquad \frac{\eta = ((\text{T}\ \ \text{any}) := \text{Bar}) \qquad \dfrac{\emptyset \vdash (\text{Bar} <: \text{any})}{\emptyset \vdash (\text{T} <: \text{any})[\eta]}}{\eta = ((\text{T}\ \ \text{any}) :=_\emptyset \text{Bar})}$$

$$\frac{\emptyset \vdash \text{Bar}\ \ ok \qquad (\textbf{type}\,\text{Foo}[\text{T}\ \ \text{any}]) \in \overline{D} \qquad \eta = ((\text{T}\ \ \text{any}) :=_\emptyset \text{Bar})}{\emptyset \vdash \text{Foo}[\text{Bar}]\ \ ok}\ \textsc{T-Named}$$

$$\frac{\dfrac{}{distinct(x)} \qquad \emptyset \vdash \text{Foo}[\text{Bar}]\ \ ok}{\emptyset \vdash \textbf{struct}\ \{x\ \ \text{Foo}[\text{Bar}]\}\ ok}\ \textsc{T-Struct}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\bot}{\text{Bar} \notin \{\text{Bar}, \text{Foo}\}}}{\mathit{notContains}(\text{Bar}, \text{Foo}, \text{Bar})}}{\mathit{notContains}(\text{Bar}, \text{Foo}, \textbf{struct}\ \{x\ \ \text{Bar}\})}}{\mathit{notContains}(\text{Bar}, \text{Foo}, \textbf{struct}\ \{x\ \ \text{T}\}[\eta])}$$

$$\dfrac{\text{Foo} \notin \{\text{Bar}\} \qquad \eta = ((\text{T}\ \ \text{any}) := \text{Bar}) \qquad \begin{array}{c}(\textbf{type}\ \ \text{Foo[T}\ \ \text{any]}\ \ \textbf{struct}\ \{x\ \ \text{T}\}) \in \overline{D} \\ \mathit{notContains}(\text{Bar}, \text{Foo}, \textbf{struct}\ \{x\ \ \text{T}\}[\eta])\end{array}}{\mathit{notContains}(\text{Bar}, \text{Foo[Bar]})}$$

$$\dfrac{\mathit{notContains}(\text{Bar}, \text{Foo[Bar]})}{\mathit{notContains}(\text{Bar}, \textbf{struct}\ \{x\ \ \text{Foo[Bar]}\})}$$

$$\dfrac{\emptyset \vdash \textbf{struct}\ \{x\ \ \text{Foo[Bar]}\}\ \mathit{ok} \qquad \mathit{notContains}(\text{Bar}, \textbf{struct}\ \{x\ \ \text{Foo[Bar]}\})}{D_2\ \mathit{ok}}\ \text{T-TYPE}$$

## 4.2 Issue #51244

Issue #51244 demonstrates another case where the compiler's cycle detection accepts or rejects the program depending on the order of type declarations. According to the above defined rules, the program should be accepted. One of the participants of the issue (findleyr) also suggested removing the restriction of cycles through type parameter constraints.

### 4.2.1 Accepting type checking derivation

```
type A interface{ M(B[T]) T }
type B[a A] struct{}

type T struct{}

func (t T) M(b B[T]) T { return t }

func main() {}
```

Below is a derivation of the program in the issue, with minor adjustments to fit a Featherweight Go-like syntax used in this document.

$$\begin{aligned}
D_0 &= \textbf{type}\ \ \text{A}\ \ \textbf{interface}\ \{\,\text{M}(b\ \ \text{B[T]})\ \ \text{T}\,\} \\
D_1 &= \textbf{type}\ \ \text{B}[a\ \ \text{A}]\ \ \textbf{struct}\ \{\} \\
D_2 &= \textbf{type}\ \ \text{T}\ \ \textbf{struct}\ \{\} \\
D_3 &= \textbf{func}\ (t\ \ \text{T})\ \ \text{M}(b\ \ \text{B[T]})\ \ \text{T}\ \ \{\ \textbf{return}\ t\ \} \\
\overline{D} &= \{D_0, D_1, D_2, D_3\}
\end{aligned}$$

$$\frac{\overline{\mathit{unique}(\mathrm{M}(b\ \mathrm{B}[\mathrm{T}])\ \mathrm{T})} \qquad \emptyset \vdash \mathrm{M}(b\ \mathrm{B}[\mathrm{T}])\ \mathrm{T}\ \mathit{ok}}{\emptyset \vdash \textbf{interface}\ \{\ \mathrm{M}(b\ \mathrm{B}[\mathrm{T}])\ \mathrm{T}\ \}\ \mathit{ok}}\ \textsc{T-Interface}$$

$$\frac{\eta = (a := \mathrm{T}) \qquad \dfrac{\dfrac{\overline{\{\ \mathrm{M}(b\ \mathrm{B}[\mathrm{T}])\ \mathrm{T}\ \} \supseteq \{\ \mathrm{M}(b\ \mathrm{B}[\mathrm{T}])\ \mathrm{T}\ \}}}{\mathit{methods}_\emptyset(\mathrm{T}) \supseteq \mathit{methods}_\emptyset(\mathrm{A})}\ <:_I}{\emptyset \vdash T <: A}}{\eta = ((a\ \mathrm{A}) := \mathrm{T}) \qquad \emptyset \vdash (a <: A)[\![\eta]\!]}}{\eta = ((a\ \mathrm{A}) :=_\emptyset \mathrm{T})}$$

$$\frac{\emptyset \vdash \mathrm{T}\ \mathit{ok} \qquad (\textbf{type}\ \mathrm{B}[a\ \mathrm{A}]\ \textbf{struct}\ \{\}) \in \overline{D} \qquad \eta = ((a\ \mathrm{A}) :=_\emptyset \mathrm{T})}{\emptyset \vdash \mathrm{B}[\mathrm{T}]\ \mathit{ok}}\ \textsc{T-Named}$$

$$\frac{\overline{\mathit{distinct}(b)} \qquad \emptyset \vdash \mathrm{B}[\mathrm{T}]\ \mathit{ok} \qquad \emptyset \vdash \mathrm{T}\ \mathit{ok}}{\emptyset \vdash \mathrm{M}(b\ \mathrm{B}[\mathrm{T}])\ \mathrm{T}\ \mathit{ok}}\ \textsc{T-Specification}$$

$$\frac{\emptyset \vdash \textbf{interface}\ \{\ \mathrm{M}(b\ \mathrm{B}[\mathrm{T}])\ \mathrm{T}\ \}\ \mathit{ok} \qquad \overline{\mathit{notContains}(\mathrm{A},\ \textbf{interface}\ \{\ \mathrm{M}(b\ \mathrm{B}[\mathrm{T}])\ \mathrm{T}\ \})}}{D_0\ \mathit{ok}}\ \textsc{T-Type}$$

$$\frac{\dfrac{\overline{\mathit{distinct}(a)} \qquad \dfrac{\overline{(\textbf{type}\ \mathrm{A}\ \textbf{interface}\ \{\ \mathrm{M}(b\ \mathrm{B}[\mathrm{T}])\ \mathrm{T}\ \}) \in \overline{D}}}{(a\ \mathrm{A}) \vdash \mathrm{A}\ \mathit{ok}}\ \textsc{T-Named}}{(a\ \mathrm{A})\ \mathit{ok}}\ \textsc{T-Formal}$$

$$\frac{(a\ \mathrm{A})\ \mathit{ok} \qquad \dfrac{\overline{(a\ \mathrm{A}) \vdash \textbf{struct}\ \{\}\ \mathit{ok}}\ \textsc{T-Struct} \qquad \overline{\mathit{notContains}(\mathrm{B},\ \textbf{struct}\ \{\})}}{D_1\ \mathit{ok}}\ \textsc{T-Type}$$

$$\frac{\dfrac{\overline{\emptyset \vdash \textbf{struct}\ \{\}\ \mathit{ok}}\ \textsc{T-Struct} \qquad \overline{\mathit{notContains}(\mathrm{T},\ \textbf{struct}\ \{\})}}{D_2\ \mathit{ok}}\ \textsc{T-Type}$$

$$\frac{\overline{(\textbf{type } T \textbf{ struct } \{\}) \in \overline{D}}}{\emptyset = \mathit{typeParams}(T)} \qquad \frac{\overline{(t : \mathrm{T}) \in (t : \mathrm{T}, b : \mathrm{B[T]})}}{\emptyset; t : \mathrm{T}, b : \mathrm{B[T]} \vdash t : \mathrm{T}}\ \text{T-VAR} \qquad \frac{}{\emptyset \vdash T <: T}\ <:_V$$

$$\frac{\emptyset \vdash \mathrm{M}(b\ \mathrm{B[T]})\ \mathrm{T}\ \mathit{ok} \qquad \overline{\mathit{distinct}(t, b)} \qquad \emptyset; t : \mathrm{T}, b : \mathrm{B[T]} \vdash t : \mathrm{T} \qquad \emptyset = \mathit{typeParams}(T) \qquad \emptyset \vdash T <: T}{D_3\ \mathit{ok}}\ \text{T-FUNC}$$

## 4.3 Self-referential type parameter constraints

The following two examples show the rules applying to types where the type parameter constraints refer to the type being defined (currently disallowed by spec). While the type parameters are recursive, the type checking algorithms do not recurse infinitely. The intuition behind this is that while type checking some declaration $D$, lookups in the global $\overline{D}$ do not depend on the types in $\overline{D}$ to have already been type checked. Similarly, when type checking some type parameter constraints $\overline{\Phi}$, lookups in the same $\overline{\Phi}$ do not require the type parameter constraints in $\overline{\Phi}$ to have already been type checked. This is crucial in order to allow for backward and forward references, and also self-references (recursion).

### 4.3.1 Negative example

In this example, the type `E` is not well-typed, since it "flips" the bounds between `Foo` and `Bar`. A contradiction is reached during the type checking derivation.

```
type Foo[F Foo[F]] interface{ m(f F) F }
type Bar[B Bar[B]] interface{ m(b B) B }

type E[F Foo[B], B Bar[F]] struct{}

func main() {}
```

$$D_0 = \textbf{type Foo[F Foo[F]] interface } \{\ m(f\ \mathrm{F})\ \mathrm{F}\ \}$$
$$D_1 = \textbf{type Bar[B Bar[B]] interface } \{\ m(b\ \mathrm{B})\ \mathrm{B}\ \}$$
$$D_2 = \textbf{type E[F Foo[B], B Bar[F]] struct } \{\}$$
$$\overline{D} = \{D_0, D_1, D_2\}$$

$$\frac{\overline{(\mathrm{F} : \mathrm{Foo[F]}) \in (\mathrm{F}\ \ \mathrm{Foo[F]})}}{(\mathrm{F}\ \ \mathrm{Foo[F]}) \vdash \mathrm{F}\ \ ok}\ \textsc{T-Param}$$

$$\frac{\eta_0 = (\mathrm{F} := \mathrm{F}) \qquad \dfrac{\dfrac{\dfrac{\overline{(\mathrm{F} : \mathrm{Foo[F]}) \in (\mathrm{F}\ \ \mathrm{Foo[F]})}}{methods_{(\mathrm{F}\ \ \mathrm{Foo[F]})}(\mathrm{F}) = methods_{(\mathrm{F}\ \ \mathrm{Foo[F]})}(\mathrm{Foo[F]})}}{methods_{(\mathrm{F}\ \ \mathrm{Foo[F]})}(\mathrm{F}) \supseteq methods_{(\mathrm{F}\ \ \mathrm{Foo[F]})}(\mathrm{Foo[F]})}}{(\mathrm{F}\ \ \mathrm{Foo[F]}) \vdash (\mathrm{F} <: \mathrm{Foo[F]})}\ <:_I}{\eta_2 = ((\mathrm{F}\ \mathrm{Foo[F]}) := \mathrm{F}) \qquad (\mathrm{F}\ \ \mathrm{Foo[F]}) \vdash (\mathrm{F} <: \mathrm{Foo[F]})[\![\eta_0]\!]}}{\eta_0 = ((\mathrm{F}\ \ \mathrm{Foo[F]}) :=_{(\mathrm{F}\ \ \mathrm{Foo[F]})} \mathrm{F})}$$

$$\frac{(\mathrm{F}\ \ \mathrm{Foo[F]}) \vdash \mathrm{F}\ \ ok \qquad \begin{array}{l}\overline{(\textbf{type}\ \ \mathrm{Foo[F}\ \ \mathrm{Foo[F]]}\ \textbf{interface}\ \{\ m(f\ \ \mathrm{F})\ \ \mathrm{F}\ \ \})} \in \overline{\overline{D}} \\ \eta_0 = ((\mathrm{F}\ \ \mathrm{Foo[F]}) :=_{(\mathrm{F}\ \ \mathrm{Foo[F]})} \mathrm{F})\end{array}}{(\mathrm{F}\ \ \mathrm{Foo[F]}) \vdash \mathrm{Foo[F]}\ \ ok}\ \textsc{T-Named}$$

$$\frac{\overline{distinct(F)} \qquad (\mathrm{F}\ \ \mathrm{Foo[F]}) \vdash \mathrm{Foo[F]}\ \ ok}{(\mathrm{F}\ \ \mathrm{Foo[F]})\ \ ok}\ \textsc{T-Formal}$$

$$\frac{\overline{distinct(f)} \qquad \dfrac{\overline{(\mathrm{F} : \mathrm{Foo[F]}) \in (\mathrm{F}\ \ \mathrm{Foo[F]})}}{(\mathrm{F}\ \ \mathrm{Foo[F]}) \vdash \mathrm{F}\ \ ok}\ \textsc{T-Param}}{(\mathrm{F}\ \ \mathrm{Foo[F]}) \vdash m(f\ \ \mathrm{F})\ \ \mathrm{F}\ \ ok}\ \textsc{T-Specification}$$

$$\frac{\overline{unique(m(f\ \ \mathrm{F})\ \ \mathrm{F})} \qquad (\mathrm{F}\ \ \mathrm{Foo[F]}) \vdash m(f\ \ \mathrm{F})\ \ \mathrm{F}\ \ ok}{(\mathrm{F}\ \ \mathrm{Foo[F]}) \vdash \textbf{interface}\ \{\ m(f\ \ \mathrm{F})\ \ \mathrm{F}\ \ \}\ ok}\ \textsc{T-Interface}$$

$$\frac{}{\overline{notContains(Foo,\ \textbf{interface}\ \{\ m(f\ \ \mathrm{F})\ \ \mathrm{F}\ \ \})}}$$

$$\frac{(\mathrm{F}\ \ \mathrm{Foo[F]})\ \ ok \qquad (\mathrm{F}\ \ \mathrm{Foo[F]}) \vdash \textbf{interface}\ \{\ m(f\ \ \mathrm{F})\ \ \mathrm{F}\ \ \}\ ok \qquad notContains(Foo,\ \textbf{interface}\ \{\ m(f\ \ \mathrm{F})\ \ \mathrm{F}\ \ \})}{D_0\ \ ok}\ \textsc{T-Type}$$

An analogous tree can be derived for $D_1$ (`Bar` is isomorphic to `Foo`).

$$\frac{\overline{(\mathrm{B}\ \ \mathrm{Bar[F]}) \in (\mathrm{F}\ \ \mathrm{Foo[B]},\ \ \mathrm{B}\ \ \mathrm{Bar[F]})}}{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \ \mathrm{B}\ \ \mathrm{Bar[F]}) \vdash \mathrm{B}\ \ ok}\ \textsc{T-Param}$$

$$\frac{}{(\textbf{type}\ \ \mathrm{Foo[F}\ \ \mathrm{Foo[F]]}\ \textbf{interface}\ \{\ m(f\ \ \mathrm{F})\ \ \mathrm{F}\ \ \}) \in \overline{\overline{D}}}$$

$$\frac{(\mathrm{B} : \mathrm{Bar[F]}) \in (\mathrm{F}\ \ \mathrm{Foo[B]},\ \ \mathrm{B}\ \ \mathrm{Bar[F]})}{\mathit{methods}_{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \ \mathrm{B}\ \ \mathrm{Bar[F]})}(\mathrm{B}) = \mathit{methods}_{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \ \mathrm{B}\ \ \mathrm{Bar[F]})}(\mathrm{Bar[F]})}$$

$$\frac{\dfrac{\dfrac{}{(\textbf{type}\ \ \mathrm{Bar[B}\ \ \mathrm{Bar[B]]}\ \textbf{interface}\ \{\ m(b\ \ \mathrm{B})\ \ \mathrm{B}\ \ \}) \in \overline{\overline{D}}} \qquad \dfrac{\eta_3 = (\mathrm{B} := \mathrm{F})}{\eta_3 = (\mathrm{B}\ \ \mathrm{Bar[B]} := \mathrm{F})}}{\mathit{methods}_{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \ \mathrm{B}\ \ \mathrm{Bar[F]})}(\mathrm{Bar[F]}) = \{\ m(b\ \ \mathrm{B})\ \ \mathrm{B}\ \}[\![\eta_3]\!]}}{\mathit{methods}_{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \ \mathrm{B}\ \ \mathrm{Bar[F]})}(\mathrm{Bar[F]}) = \{\ m(b\ \ \mathrm{F})\ \ \mathrm{F}\ \ \}}$$

$$\frac{\dfrac{\dfrac{}{\textbf{type}\ \ \mathrm{Foo[F}\ \ \mathrm{Foo[F]]}\ \textbf{interface}\ \{\ m(f\ \ \mathrm{F})\ \ \mathrm{F}\ \ \}} \qquad \dfrac{\eta_2 = (\mathrm{F} := \mathrm{B})}{\eta_2 = ((\mathrm{F}\ \ \mathrm{Foo[F]}) := \mathrm{B})}}{\mathit{methods}_{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \ \mathrm{B}\ \ \mathrm{Bar[F]})}(\mathrm{Foo[B]}) = \{\ m(f\ \ \mathrm{F})\ \ \mathrm{F}\ \ \}\ [\![\eta_2]\!]}}{\mathit{methods}_{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \ \mathrm{B}\ \ \mathrm{Bar[F]})}(\mathrm{Foo[B]}) = \{\ m(f\ \ \mathrm{B})\ \ \mathrm{B}\ \}}$$

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\bot}{\{\ m(b\ \ \mathrm{F})\ \ \mathrm{F}\ \ \} \supseteq \{\ m(f\ \ \mathrm{B})\ \ \mathrm{B}\ \}}}{\mathit{methods}_{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \ \mathrm{B}\ \ \mathrm{Bar[F]})}(\mathrm{Bar[F]}) \supseteq \mathit{methods}_{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \ \mathrm{B}\ \ \mathrm{Bar[F]})}(\mathrm{Foo[B]})}}{\mathit{methods}_{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \ \mathrm{B}\ \ \mathrm{Bar[F]})}(\mathrm{B}) \supseteq \mathit{methods}_{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \ \mathrm{B}\ \ \mathrm{Bar[F]})}(\mathrm{Foo[B]})}}{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \ \mathrm{B}\ \ \mathrm{Bar[F]}) \vdash (\mathrm{B} <: \mathrm{Foo[B]})}\ <:_I}{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \ \mathrm{B}\ \ \mathrm{Bar[F]}) \vdash (\mathrm{F} <: \mathrm{Foo[B]})[\![\eta_2]\!]}$$

$$\frac{\dfrac{\eta_2 = (\mathrm{F} := \mathrm{B})}{\eta_2 = ((\mathrm{F}\ \ \mathrm{Foo[F]}) := \mathrm{B})} \qquad (\mathrm{F}\ \ \mathrm{Foo[B]},\ \mathrm{B}\ \ \mathrm{Bar[F]}) \vdash (\mathrm{B} <: \mathrm{Bar[F]})[\![\eta_2]\!]}{\eta_2 = ((\mathrm{F}\ \ \mathrm{Foo[F]}) :=_{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \mathrm{B}\ \ \mathrm{Bar[F]})} \mathrm{B})} \qquad (\mathrm{F}\ \ \mathrm{Foo[B]},\ \mathrm{B}\ \ \mathrm{Bar[F]}) \vdash (\mathrm{F} <: \mathrm{Foo[B]})[\![\eta_2]\!]$$

$$\frac{\begin{array}{c}(\mathrm{F}\ \ \mathrm{Foo[B]},\ \mathrm{B}\ \ \mathrm{Bar[F]}) \vdash \mathrm{B}\ \ ok \\ (\textbf{type}\ \ \mathrm{Foo[F}\ \ \mathrm{Foo[F]]}\ \textbf{interface}\ \{\ m(f\ \ \mathrm{F})\ \ \mathrm{F}\ \ \}) \in \overline{D} \\ \eta_2 = ((\mathrm{F}\ \ \mathrm{Foo[F]}) :=_{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \mathrm{B}\ \ \mathrm{Bar[F]})} \mathrm{B})\end{array}}{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \mathrm{B}\ \ \mathrm{Bar[F]}) \vdash \mathrm{Foo[B]}\ \ ok}\ \text{T-Named}$$

$$\frac{\overline{\mathit{distinct}(\mathrm{F}, \mathrm{B})} \qquad (\mathrm{F}\ \ \mathrm{Foo[B]},\ \mathrm{B}\ \ \mathrm{Bar[F]}) \vdash \mathrm{Foo[B]}\ \ ok \qquad (\mathrm{F}\ \ \mathrm{Foo[B]},\ \mathrm{B}\ \ \mathrm{Bar[F]}) \vdash \mathrm{Bar[F]}\ \ ok}{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \mathrm{B}\ \ \mathrm{Bar[F]})\ \ ok}\ \text{T-Formal}$$

$$\frac{}{(\mathrm{F}\ \ \mathrm{Foo[B]},\ \mathrm{B}\ \ \mathrm{Bar[F]}) \vdash \textbf{struct}\ \{\}\ \ ok}$$

$$\frac{\overline{\mathit{notContains}(\mathrm{E},\ \textbf{struct}\ \{\})} \qquad (\mathrm{F}\ \ \mathrm{Foo[B]},\ \mathrm{B}\ \ \mathrm{Bar[F]})\ \ ok}{D_2\ \ ok}\ \text{T-Type}$$

#### 4.3.2 Positive example

```
type Foo[F Foo[F]] interface{ m(f F) F }
type Bar[B Bar[B]] interface{ m(b B) B }

type E[F Foo[F], B Bar[B]] struct{}

type T struct{}

func (t T) m(t2 T) T { return t }

func main() {
    _ = E[T, T]{}
}
```

$$
\begin{aligned}
D_0 &= \textbf{type}\ \ \text{Foo[F}\ \ \text{Foo[F]]}\ \textbf{interface}\ \{\ m(f\ \ \text{F})\ \ \text{F}\ \ \}\\
D_1 &= \textbf{type}\ \ \text{Bar[B}\ \ \text{Bar[B]]}\ \textbf{interface}\ \{\ m(b\ \ \text{B})\ \ \text{B}\ \ \}\\
D_2 &= \textbf{type}\ \ \text{E[F}\ \ \text{Foo[F]},\ \ \text{B}\ \ \text{Bar[B]]}\ \textbf{struct}\ \{\}\\
D_3 &= \textbf{type}\ \ \text{T}\ \ \textbf{struct}\ \{\}\\
D_4 &= \textbf{func}\ (t\ \ \text{T})\ m(t2\ \ \text{T})\ \ \text{T}\ \ \{\ \textbf{return}\ t\ \}\\
\overline{D} &= \{D_0, D_1, D_2, D_3, D_4\}\\
e &= \text{E[T},\ \ \text{T]}\{\}
\end{aligned}
$$

The derivations for $D_0$, $D_1$, and $D_3$ were shown in previous examples.

$$
\frac{\overline{\mathit{distinct}(t2)} \qquad \dfrac{(\textbf{type}\ \ \text{T}\ \ \textbf{struct}\ \{\}) \in \overline{D}}{\emptyset \vdash \text{T}\ \ \mathit{ok}}\ \text{T-Named}}{\emptyset \vdash m(t2\ \ \text{T})\ \ \text{T}\ \ \mathit{ok}}\ \text{T-Specification}
$$

$$
\frac{\overline{(t : \text{T}) \in (t : \text{T}, t2 : \text{T})}}{\emptyset; t : \text{T}, t2 : \text{T} \vdash t : \text{T}}\ \text{T-Var}
$$

$$
\frac{\overline{\mathit{distinct}(t, t2)} \qquad \overline{\emptyset = \mathit{typeParams}(\text{T})} \qquad \emptyset \vdash m(t2\ \ \text{T})\ \ \text{T}\ \ \mathit{ok} \qquad \emptyset; t : \text{T}, t2 : \text{T} \vdash t : \text{T} \qquad \overline{\text{T} <: \text{T}}^{\ <:_V}}{D_4\ \ \mathit{ok}}\ \text{T-Func}
$$

(F Foo[F], B Bar[B]) ⊢ Foo[F] *ok* can be derived analogously to (F Foo[F]) ⊢ Foo[F] *ok* in the previous example. The same applies for (F Foo[F], B Bar[B]) ⊢ Bar[B] *ok*.

$$\frac{\dfrac{}{\mathit{distinct}(\mathrm{F}, \mathrm{B})}}{\dfrac{(\mathrm{F}\ \ \mathrm{Foo[F]},\ \ \mathrm{B}\ \ \mathrm{Bar[B]}) \vdash \mathrm{Foo[F]}\ \ \mathit{ok} \qquad (\mathrm{F}\ \ \mathrm{Foo[F]},\ \ \mathrm{B}\ \ \mathrm{Bar[B]}) \vdash \mathrm{Bar[B]}\ \ \mathit{ok}}{(\mathrm{F}\ \ \mathrm{Foo[F]},\ \ \mathrm{B}\ \ \mathrm{Bar[B]})\ \ \mathit{ok}}}\ \text{T-Formal}$$

$$(\mathrm{F}\ \ \mathrm{Foo[F]},\ \ \mathrm{B}\ \ \mathrm{Bar[B]})\ \ \mathit{ok}$$

$$\frac{\dfrac{}{\mathrm{F}\ \ \mathrm{Foo[F]},\ \ \mathrm{B}\ \ \mathrm{Bar[B]} \vdash \mathbf{struct}\{\}\ \ \mathit{ok}} \qquad \dfrac{}{\mathit{notContains}(\mathrm{E},\ \mathbf{struct}\{\})}}{D_2\ \ \mathit{ok}}\ \text{T-Type}$$

$$\frac{\dfrac{}{\emptyset = \mathit{typeParams}(\mathrm{T})}}{\mathit{methods}_{\emptyset}(T) = \{\ m(t2\ \ \mathrm{T})\ \ \mathrm{T}\ \}}$$

$$\frac{(\mathbf{type}\ \ \mathrm{Foo[F}\ \ \mathrm{Foo[F]]}\ \mathbf{interface}\ \{\ m(f\ \ \mathrm{F})\ \ \mathrm{F}\ \}) \in \overline{D} \qquad \dfrac{\dfrac{\eta_5 = (\mathrm{F} := \mathrm{T})}{\eta_5 = (\mathrm{F}\ \ \mathrm{Foo[F]} := \mathrm{T})}}{\mathit{methods}_{\emptyset}(\mathrm{Foo[T]}) = \{\ m(f\ \ \mathrm{F})\ \ \mathrm{F}\ \}[\![\eta_5]\!]}}{\mathit{methods}_{\emptyset}(\mathrm{Foo[T]}) = \{\ m(f\ \ \mathrm{T})\ \ \mathrm{T}\ \}}$$

$\emptyset \vdash (\mathrm{T} <: \mathrm{Bar[T]})$ can be derived analogously to $\emptyset \vdash (\mathrm{T} <: \mathrm{Foo[T]})$.

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\ m(t2\ \ \mathrm{T})\ \ \mathrm{T}\ \} \supseteq \{\ m(f\ \ \mathrm{T})\ \ \mathrm{T}\ \}}}{\mathit{methods}_{\emptyset}(T) \supseteq \mathit{methods}_{\emptyset}(\mathrm{Foo[T]})}}{\emptyset \vdash (\mathrm{T} <: \mathrm{Foo[T]})}\ <:_I}{\emptyset \vdash (\mathrm{F} <: \mathrm{Foo[F]})[\eta_4]} \qquad \frac{\emptyset \vdash (\mathrm{T} <: \mathrm{Bar[T]})}{\emptyset \vdash (\mathrm{B} <: \mathrm{Bar[B]})[\eta_4]}$$

$$\frac{\dfrac{\eta_4 = (\mathrm{F} := \mathrm{T},\ \ \mathrm{B} := \mathrm{T})}{\eta_4 = (\mathrm{F}\ \ \mathrm{Foo[F]} := \mathrm{T},\ \ \mathrm{B}\ \ \mathrm{Bar[B]} := \mathrm{T})} \qquad \emptyset \vdash (\mathrm{F} <: \mathrm{Foo[F]})[\eta_4] \qquad \emptyset \vdash (\mathrm{B} <: \mathrm{Bar[B]})[\eta_4]}{\eta_4 = (\mathrm{F}\ \ \mathrm{Foo[F]} :=_{\emptyset} \mathrm{T},\ \ \mathrm{B}\ \ \mathrm{Bar[B]} :=_{\emptyset} \mathrm{T})}$$

$$\frac{\emptyset \vdash \mathrm{T}\ \ \mathit{ok} \qquad \dfrac{(\mathbf{type}\ \ \mathrm{E[F}\ \ \mathrm{Foo[F]},\ \ \mathrm{B}\ \ \mathrm{Bar[B]]}\ \mathbf{struct}\ \{\}) \in \overline{D}}{\eta_4 = (\mathrm{F}\ \ \mathrm{Foo[F]} :=_{\emptyset} \mathrm{T},\ \ \mathrm{B}\ \ \mathrm{Bar[B]} :=_{\emptyset} \mathrm{T})}}{\emptyset \vdash \mathrm{E[T, T]}\ \ \mathit{ok}}\ \text{T-Named}$$

$$\frac{\emptyset \vdash \mathrm{E[T, T]}\ \ \mathit{ok}}{\emptyset; \emptyset \vdash \mathrm{E[T, T]}\{\} : \mathrm{E[T, T]}}\ \text{T-Struct-Literal}$$

# 5 Remaining rules

## 5.1 Syntax

| | | | |
|---|---|---|---|
| Structure type name | $t_S, u_S$ | Structure type | $\tau_S, \sigma_S ::= t_S[\overline{\tau}]$ |
| Interface type name | $t_I, u_I$ | Interface type | $\tau_I, \sigma_I ::= t_I[\overline{\tau}]$ |
| Array type name | $t_A, u_A$ | Array type | $\tau_A, \sigma_A ::= t_A[\overline{\tau}]$ |
| Value type name | $t_V, u_V ::= t_S \mid t_A$ | Value type | $\tau_V, \sigma_V ::= \tau_S \mid \tau_A$ |
| Type name | $t, u ::= t_V \mid t_I$ | | |
| Declaration | $D ::=$ | | |
| Type declaration | $\textbf{type}\ t[\overline{\Phi}]\ T$ | | |
| Method declaration | $\textbf{func}\ (x\ t_V[\overline{\alpha}])\ mM\ \{\textbf{return}\ e\}$ | | |

Well-formed type formals $\boxed{\overline{\Phi}\ ok}$

$$\frac{(\overline{\alpha\ \gamma}) = \overline{\Phi} \qquad \mathit{distinct}(\overline{\alpha}) \qquad \overline{\Phi} \vdash \overline{\gamma\ ok}}{\overline{\Phi}\ ok}\ \text{T-FORMAL}$$

Well-formed type $\boxed{\Delta \vdash \tau\ ok}$

$$\frac{(\alpha : \gamma) \in \Delta}{\Delta \vdash \alpha\ ok}\ \text{T-PARAM} \qquad \frac{\Delta \vdash \overline{\tau\ ok} \qquad (\textbf{type}\ t[\overline{\Phi}]\ T) \in \overline{D} \qquad \eta = (\overline{\Phi :=_\Delta \tau})}{\Delta \vdash t[\overline{\tau}]\ ok}\ \text{T-NAMED}$$

$$\frac{(\overline{\alpha\ \gamma}) = \overline{\Phi} \qquad \eta = (\overline{\alpha := \tau})}{(\overline{\Phi := \tau}) = \eta} \qquad \frac{(\overline{\alpha\ \gamma}) = \overline{\Phi} \qquad \eta = (\overline{\Phi := \tau}) \qquad \Delta \vdash (\overline{\alpha <: \gamma})[\![\eta]\!]}{(\overline{\Phi :=_\Delta \tau}) = \eta}$$

Well-formed method specifications and type literals $\boxed{\overline{\Phi} \vdash S\ ok}$ $\boxed{\overline{\Phi} \vdash T\ ok}$

$$\frac{\mathit{distinct}(\overline{x}) \qquad \overline{\Phi} \vdash \overline{\tau\ ok} \qquad \overline{\Phi} \vdash \tau\ ok}{\overline{\Phi} \vdash m(\overline{x\ \tau})\ \tau\ ok}\ \text{T-SPECIFICATION}$$

$$\frac{\mathit{distinct}(\overline{f}) \qquad \overline{\Phi} \vdash \overline{\tau\ ok}}{\overline{\Phi} \vdash \textbf{struct}\ \{\overline{f\ \tau}\}\ ok}\ \text{T-STRUCT}$$

$$\frac{\mathit{unique}(\overline{S}) \qquad \overline{\Phi} \vdash \overline{S\ ok}}{\overline{\Phi} \vdash \textbf{interface}\ \{\overline{S}\}}\ \text{T-INTERFACE}$$

$$\frac{\overline{\Phi} \vdash \tau\ ok}{\overline{\Phi} \vdash\ [n]\tau\ ok}\ \text{T-ARRAY}$$

Well-formed method declarations $\boxed{D\ ok}$

$$\frac{\begin{array}{ccc} & \mathit{distinct}(x, \overline{x}) & \overline{\Phi} = \mathit{typeParams}(t_V) \\ (\overline{\alpha\ \gamma}) = \overline{\Phi} & \overline{\Phi} \vdash m(\overline{x\ \tau})\ \sigma\ ok & \overline{\Phi}; x : t_V[\overline{\alpha}], \overline{x : \tau} \vdash e : \tau \qquad \overline{\Phi} \vdash \tau <: \sigma \end{array}}{\textbf{func}\ (x\ t_V[\overline{\alpha}])\ m(\overline{x\ \tau})\ \sigma\ \{\textbf{return}\ e\}\ ok}\ \text{T-FUNC}$$

Implements $\boxed{\Delta \vdash \tau <: \sigma}$

$$\frac{}{\Delta \vdash \alpha <: \alpha}\ <:_{\text{PARAM}} \qquad \frac{}{\Delta \vdash \tau_V <: \tau_V}\ <:_V \qquad \frac{methods_\Delta(\tau) \supseteq methods_\Delta(\tau_I)}{\Delta \vdash \tau <: \tau_I}\ <:_I$$

$$\frac{\eta = (\overline{\Phi := \tau}) \qquad \overline{\Phi} = typeParams(t_V)}{methods_\Delta(t_V[\overline{\tau}]) = \{(mM)[\![\eta]\!] \mid (\textbf{func}\ (x\ t_V[\overline{\alpha}])\ mM\ \{\textbf{return}\ e\}) \in \overline{D}\}}$$

$$\frac{\textbf{type}\ t_I[\overline{\Phi}]\ \textbf{interface}\ \{\overline{S}\} \in \overline{D} \qquad \eta = (\overline{\Phi := \tau})}{methods_\Delta(t_I[\overline{\tau}]) = \overline{S}[\![\eta]\!]} \qquad \frac{(\alpha : \gamma) \in \Delta}{methods_\Delta(\alpha) = methods_\Delta(\gamma)}$$

$$\frac{(\textbf{type}\ t[\overline{\Phi}]\ T) \in \overline{D}}{\overline{\Phi} = typeParams(t)}$$

Expressions $\boxed{\Delta;\ \Gamma \vdash e : \tau}$

$$\frac{(x : \tau) \in \Gamma}{\Delta;\ \Gamma \vdash x : \tau}\ \text{T-VAR} \qquad \frac{\Delta \vdash \tau_S\ ok \qquad \Delta;\ \Gamma \vdash \overline{e : \tau} \qquad (\overline{f\ \sigma}) = fields(\tau_S) \qquad \Delta \vdash \overline{\tau <: \sigma}}{\Delta;\ \Gamma \vdash \tau_S\{\overline{e}\} : \tau_S}\ \text{T-STRUCT-LITERAL}$$