
Featherweight Goをつくる

YOSHIMURA Hikaru (吉村 優)

hikaru_yoshimura@r.recruit.co.jp

English バックエンド開発グループ

July 17, 2020 @ Quipper LT

[y-yu/featherweight_go-slide@baade48](#)

目次

- ① 自己紹介
- ② Featherweight Go とは？
- ③ プログラム言語処理系をつくるには？
- ④ Featherweight Go の実装
- ⑤ まとめ

自己紹介



Twitter	@_yyu_
Qiita	yyu
GitHub	y-yu
Facebook	h1karuy

自己紹介



Twitter @_yyu_
Qiita yyu
GitHub y-yu
Facebook h1karuy

- 筑波大学 情報学群情報科学類卒（学士）
 - 正規表現を利用した型システムの研究
- 株式会社リクルートマーケティングパートナーズ（中途）
 - スタディサプリ ENGLISH サーバーサイド（Scala）
- 未踏ターゲット事業（ゲート式量子コンピュータ）
 - 公平な抽選プロトコルの開発・シミュレーター実装
- 暗号・セキュリティー
 - 大学同期とのチームで CTF 活動を 7 年くらいやっている
- \LaTeX 組版
 - 同人誌を 5 年くらい作っている
 - The Rust Programming Language 2nd Edition（日本語）や Erlang in Anger（日本語）の組版をやった
 - このスライドも全部 \LaTeX で作った

Featherweight Go とは

Featherweight Go とは

- Featherweight Go (FG) は [1] で導入された Go 言語から “Go の特徴” を残した最小の部分を取り出した小型モデル言語

Featherweight Go とは

- Featherweight Go (FG) は [1] で導入された Go 言語から “Go の特徴” を残した最小の部分を取り出した小型モデル言語
- Go 言語は実用言語なので、フォーマルな議論などには不要な機能がある（たとえばシンタックスシュガーとか）

Featherweight Go とは

- Featherweight Go (FG) は [1] で導入された Go 言語から “Go の特徴” を残した最小の部分を取り出した小型モデル言語
- Go 言語は実用言語なので、フォーマルな議論などには不要な機能がある（たとえばシンタックスシュガーとか）
- 実用するには不便ではあるが、FG で Go 言語が表現できるなら、FG で型などの議論をすればよく、考えることが減る

Featherweight Go とは

- Featherweight Go (FG) は [1] で導入された Go 言語から “Go の特徴” を残した最小の部分を取り出した小型モデル言語
- Go 言語は実用言語なので、フォーマルな議論などには不要な機能がある（たとえばシンタックスシュガーとか）
- 実用するには不便ではあるが、FG で Go 言語が表現できるなら、FG で型などの議論をすればよく、考えることが減る
- 似たようなコンセプトに *Featherweight Java*[2] や *DOT*[3] がある

Featherweight Go とは

- Featherweight Go (FG) は [1] で導入された Go 言語から “Go の特徴” を残した最小の部分を取り出した小型モデル言語
- Go 言語は実用言語なので、フォーマルな議論などには不要な機能がある（たとえばシンタックスシュガーとか）
- 実用するには不便ではあるが、FG で Go 言語が表現できるなら、FG で型などの議論をすればよく、考えることが減る
- 似たようなコンセプトに *Featherweight Java*[2] や *DOT*[3] がある
- 本来は型システムなどの形式手法を議論するためのものだが、小さいので自作するのが楽ということでやってみた

Featherweight Go とは

- [1] ではさらに *Generics* を入れた “FGG” (Featherweight Go with Generics) を定義している

Featherweight Go とは

- [1] ではさらに *Generics* を入れた “FGG” (Featherweight Go with Generics) を定義している
- そして FGG から FG への変換 (*monomorphisation*) を定義することで、
 - ① Go 言語のコードは FG のコードで (量は増えるけど) 表現できる
 - ② 型が付く FGG のコードを、型がつく FG へ変換できる
 - ③ したがって FG 経由で Go 言語に Generics が入れられる

Featherweight Go とは

- [1] ではさらに *Generics* を入れた “FGG” (Featherweight Go with Generics) を定義している
- そして FGG から FG への変換 (*monomorphisation*) を定義することで、
 - ① Go 言語のコードは FG のコードで (量は増えるけど) 表現できる
 - ② 型が付く FGG のコードを、型がつく FG へ変換できる
 - ③ したがって FG 経由で Go 言語に Generics が入れられる
- まだ FGG は実装してない🙄 ので、FG だけについて今回は解説！

Featherweight Go の機能

Featherweight Go の機能

- 機能を列挙すると👉 となる
 - プリミティブ型はなし（！？）
 - `struct`（構造体）と `interface`（インターフェース）の宣言
 - 関数定義
 - 構造体の初期化・フィールド解決
 - メソッドコール

Featherweight Go の機能

- 機能を列挙すると👉 となる
 - プリミティブ型はなし（！？）
 - struct（構造体）と interface（インターフェース）の宣言
 - 関数定義
 - 構造体の初期化・フィールド解決
 - メソッドコール

これじゃ使い物にならんでしょ😡



Featherweight Go の機能

- 機能を列挙すると👉 となる
 - プリミティブ型はなし（！？）
 - struct（構造体）と interface（インターフェース）の宣言
 - 関数定義
 - 構造体の初期化・フィールド解決
 - メソッドコール

これじゃ使い物にならんでしょ😡



- とはいえこれくらいあれば、実はそれなりなものが書ける

Featherweight Go の機能

- 機能を列挙すると👉 となる
 - プリミティブ型はなし（！？）
 - struct（構造体）と interface（インターフェース）の宣言
 - 関数定義
 - 構造体の初期化・フィールド解決
 - メソッドコール

これじゃ使い物にならんでしょ😡



- とはいえこれくらいあれば、実はそれなりなものが書ける
- https://y-yu.github.io/featherweight_go/ 👉 で今すぐ試そう！

Featherweight Go のプログラム例

```
package main;

type Nat interface {
    plus(a Nat) Nat
}

type Zero struct { }
type Succ struct {
    pred Nat
}

func (this Zero) plus(a Nat) Nat {
    return a
}

func (this Succ) plus(a Nat) Nat {
    return Succ{this.pred.plus(a)}
}

func main() {
    _ = Succ{Succ{Zero{}}}.plus(Succ{Succ{Zero{}}})
}
```

Featherweight Go のプログラム例

```
package main;

type Nat interface {
    plus(a Nat) Nat
}

type Zero struct { }
type Succ struct {
    pred Nat
}

func (this Zero) plus(a Nat) Nat {
    return a
}

func (this Succ) plus(a Nat) Nat {
    return Succ{this.pred.plus(a)}
}

func main() {
    _ = Succ{Succ{Zero{}}}.plus(Succ{Succ{Zero{}}})
}
```

Zero 0 を定義

Succ(n) $n + 1$ を定義

Featherweight Go のプログラム例

```
package main;

type Nat interface {
    plus(a Nat) Nat
}

type Zero struct { }
type Succ struct {
    pred Nat
}

func (this Zero) plus(a Nat) Nat {
    return a
}

func (this Succ) plus(a Nat) Nat {
    return Succ{this.pred.plus(a)}
}

func main() {
    _ = Succ{Succ{Zero{}}}.plus(Succ{Succ{Zero{}}})
}
```

Zero 0 を定義

Succ(n) $n + 1$ を定義

自然数の完成！



Featherweight Go のプログラム例

```
package main;

type Nat interface {
    plus(a Nat) Nat
}

type Zero struct { }
type Succ struct {
    pred Nat
}

func (this Zero) plus(a Nat) Nat {
    return a
}

func (this Succ) plus(a Nat) Nat {
    return Succ{this.pred.plus(a)}
}

func main() {
    _ = Succ{Succ{Zero{}}}.plus(Succ{Succ{Zero{}}})
}
```

Zero 0 を定義

Succ(n) $n + 1$ を定義

自然数の完成！



- Succ{Succ{Zero{}}}}は
 $1 + 1 + 0 = 2$

Featherweight Go のプログラム例

```
package main;

type Nat interface {
    plus(a Nat) Nat
}

type Zero struct { }
type Succ struct {
    pred Nat
}

func (this Zero) plus(a Nat) Nat {
    return a
}

func (this Succ) plus(a Nat) Nat {
    return Succ{this.pred.plus(a)}
}

func main() {
    _ = Succ{Succ{Zero{}}}.plus(Succ{Succ{Zero{}}})
}
```

Zero 0 を定義

Succ(n) $n + 1$ を定義

自然数の完成！



- Succ{Succ{Zero{}}} は $1 + 1 + 0 = 2$
- $2 + 2$ となり結果は 4 を表わす👉となる

Succ{Succ{Succ{Succ{Zero{}}}}}

プログラム言語処理系をつくるには？

プログラム言語処理系をつくるには？

- 言語処理系はだいたい次の流れになる
 1. パーズ 文字列レベルのコード（具象構文）を**抽象構文木**へ変換する
 2. 型チェック 抽象構文木に型が付くかを検査する
 3. 実行 抽象構文木に基づいてプログラムを実行する

プログラム言語処理系をつくるには？

- 言語処理系はだいたい次の流れになる
 1. パーズ 文字列レベルのコード（具象構文）を**抽象構文木**へ変換する
 2. 型チェック 抽象構文木に型が付くかを検査する
 3. 実行 抽象構文木に基づいてプログラムを実行する

この余白はそれを書くには狭すぎる



プログラム言語処理系をつくるには？

- 言語処理系はだいたい次の流れになる
 1. パーズ 文字列レベルのコード（具象構文）を**抽象構文木**へ変換する
 2. 型チェック 抽象構文木に型が付くかを検査する
 3. 実行 抽象構文木に基づいてプログラムを実行する

この余白はそれを書くには狭すぎる




すごい雑に説明するか！



パーザーの作成

パーザーの作成

 ! ! ! 気合 ! ! ! 

パーザーの作成

 **！！！！気合！！！！** 


- 具象構文から抽象構文にするのが辛すぎる……😓
 - いちおう正規表現エンジンを自作した経験があるけど、それでもつらい

パーザーの作成

 **！！！！気合！！！！** 

- 具象構文から抽象構文にするのが辛すぎる……😓
 - いちおう正規表現エンジンを自作した経験があるけど、それでもつらい
- 詳細はとにかく面倒なので**全て割愛**します！

パーザーの作成

 **!!! 気合!!!** 

- 具象構文から抽象構文にするのが辛すぎる……😓
 - いちおう正規表現エンジンを自作した経験があるけど、それでもつらい
- 詳細はとにかく面倒なので**全て割愛**します！
- さっきの3つの中でパーザーが最も面倒だと思う
 - このスライド作ってる間にもバグを見つけた……

パーザーの作成

 **！！！！気合！！！！** 

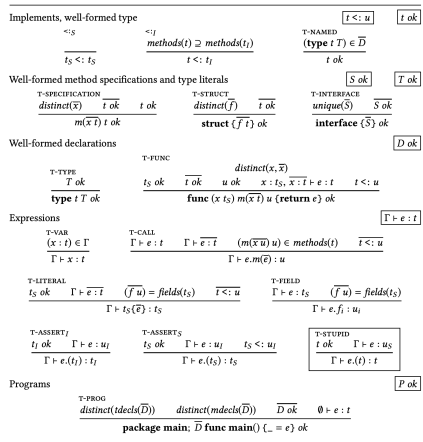
- 具象構文から抽象構文にするのが辛すぎる……😓
 - いちおう正規表現エンジンを自作した経験があるけど、それでもつらい
- 詳細はとにかく面倒なので**全て割愛**します！
- さっきの3つの中でパーザーが最も面倒だと思う
 - このスライド作ってる間にもバグを見つけた……
- プログラム言語を作りたいけどパーザーに興味はないなら、具象構文としてJSONなどを使ってしまうという手もある……😓

型検査器の作成

Implements, well-formed type			$\boxed{t <: u}$	$\boxed{t \text{ ok}}$
$\frac{}{t_S <: t_S}$	$\frac{}{methods(t) \supseteq methods(t_I)}$	$\frac{}{t <: t_I}$	$\frac{}{t \text{ ok}}$	
Well-formed method specifications and type literals			$\boxed{S \text{ ok}}$	$\boxed{T \text{ ok}}$
$\frac{}{distinct(\bar{x})}$	$\frac{}{distinct(\bar{f})}$	$\frac{}{unique(\bar{S})}$	$\frac{}{S \text{ ok}}$	$\frac{}{T \text{ ok}}$
$\frac{}{m(\bar{x} \bar{t}) \text{ ok}}$	$\frac{}{struct \{ \bar{f} \bar{t} \} \text{ ok}}$	$\frac{}{interface \{ \bar{S} \} \text{ ok}}$		
Well-formed declarations			$\boxed{D \text{ ok}}$	
$\frac{}{T \text{ ok}}$	$\frac{}{t_S \text{ ok}}$	$\frac{}{u \text{ ok}}$	$\frac{}{x : t_S, \bar{x} : \bar{t} \vdash e : t}$	$\frac{}{t <: u}$
$\frac{}{type \ t \ T \text{ ok}}$	$\frac{}{func \ (x \ t_S) \ m(\bar{x} \bar{t}) \ u \ (return \ e) \text{ ok}}$			
Expressions			$\boxed{\Gamma \vdash e : t}$	
$\frac{}{\Gamma \vdash x : t}$	$\frac{}{\Gamma \vdash e : t}$	$\frac{}{\Gamma \vdash \bar{e} : \bar{t}}$	$\frac{}{(m(\bar{x} \bar{u}) \ u) \in methods(t)}$	$\frac{}{t <: u}$
$\frac{}{t_S \text{ ok}}$	$\frac{}{\Gamma \vdash \bar{e} : \bar{t}}$	$\frac{}{(\bar{f} \ \bar{u}) = fields(t_S)}$	$\frac{}{\bar{t} <: \bar{u}}$	
$\frac{}{\Gamma \vdash t_S \{ \bar{e} \} : t_S}$	$\frac{}{\Gamma \vdash e : t_S}$	$\frac{}{(\bar{f} \ \bar{u}) = fields(t_S)}$	$\frac{}{\Gamma \vdash e.f_i : u_i}$	
$\frac{}{t_I \text{ ok}}$	$\frac{}{t_S \text{ ok}}$	$\frac{}{\Gamma \vdash e : u_I}$	$\frac{}{t_S <: u_I}$	
$\frac{}{\Gamma \vdash e.(t_I) : t_I}$	$\frac{}{\Gamma \vdash e : u_I}$	$\frac{}{t_S <: u_I}$	$\frac{}{\Gamma \vdash e.(t_S) : t_S}$	
Programs			$\boxed{P \text{ ok}}$	
$\frac{}{distinct(tdecls(\bar{D}))}$	$\frac{}{distinct(mdecls(\bar{D}))}$	$\frac{}{\bar{D} \text{ ok}}$	$\frac{}{\emptyset \vdash e : t}$	
$\text{package main; } \bar{D} \text{ func main() } \{ _ = e \} \text{ ok}$				

Figure: [1] より

型検査器の作成



- ➡ 型ルールの図に基づいて、ひたすら実装

Figure: [1] より

型検査器の作成

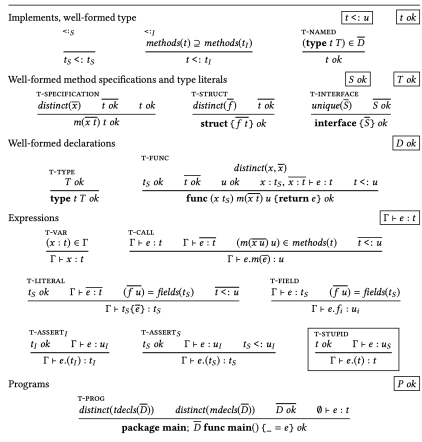
<p>Implements, well-formed type</p> $\frac{<_S}{t_S <: t_S} \quad \frac{<_I \quad \text{methods}(t) \supseteq \text{methods}(t_I)}{t <: t_I} \quad \frac{\text{T-NAMED} \quad (\text{type } t \ T) \in \bar{D}}{t \text{ ok}} \quad \boxed{t <: u} \quad \boxed{t \text{ ok}}$		
<p>Well-formed method specifications and type literals</p> $\frac{\text{T-SPECIFICATION} \quad \text{distinct}(\bar{x}) \quad \bar{t} \text{ ok} \quad t \text{ ok}}{m(\bar{x} \ \bar{t}) \ t \text{ ok}} \quad \frac{\text{T-STRUCT} \quad \text{distinct}(\bar{f}) \quad \bar{t} \text{ ok}}{\text{struct } \{\bar{f} \ \bar{t}\} \text{ ok}} \quad \frac{\text{T-INTERFACE} \quad \text{unique}(\bar{S}) \quad \bar{S} \text{ ok}}{\text{interface } \{\bar{S}\} \text{ ok}} \quad \boxed{S \text{ ok}} \quad \boxed{T \text{ ok}}$		
<p>Well-formed declarations</p> $\frac{\text{T-TYPE} \quad T \text{ ok}}{\text{type } t \ T \text{ ok}} \quad \frac{\text{T-FUNC} \quad \text{distinct}(x, \bar{x}) \quad t_S \text{ ok} \quad \bar{t} \text{ ok} \quad u \text{ ok} \quad x : t_S, \bar{x} : \bar{t} \vdash e : t \quad t <: u}{\text{func } (x \ t_S) \ m(\bar{x} \ \bar{t}) \ u \ \{\text{return } e\} \text{ ok}} \quad \boxed{D \text{ ok}}$		
<p>Expressions</p> $\frac{\text{T-VAR} \quad (x : t) \in \Gamma \quad \Gamma \vdash x : t}{\Gamma \vdash e : t} \quad \frac{\text{T-CALL} \quad \Gamma \vdash \bar{e} : \bar{t} \quad (m(\bar{x} \ \bar{u}) \in \text{methods}(t)) \quad \bar{t} <: u}{\Gamma \vdash e.m(\bar{e}) : u} \quad \boxed{\Gamma \vdash e : t}$		
$\frac{\text{T-LITERAL} \quad t_S \text{ ok} \quad \Gamma \vdash \bar{e} : \bar{t} \quad (\bar{f} \ \bar{u}) = \text{fields}(t_S) \quad \bar{t} <: u}{\Gamma \vdash t_S\{\bar{e}\} : t_S} \quad \frac{\text{T-FIELD} \quad \Gamma \vdash e : t_S \quad (\bar{f} \ \bar{u}) = \text{fields}(t_S)}{\Gamma \vdash e.f_i : u_i}$		
$\frac{\text{T-ASSERT}_I \quad t_I \text{ ok} \quad \Gamma \vdash e : u_I}{\Gamma \vdash e.(t_I) : t_I} \quad \frac{\text{T-ASSERT}_S \quad t_S \text{ ok} \quad \Gamma \vdash e : u_I \quad t_S <: u_I}{\Gamma \vdash e.(t_S) : t_S} \quad \boxed{\text{T-STUPID} \quad t \text{ ok} \quad \Gamma \vdash e : u_S \quad \Gamma \vdash e.(t) : t} \quad \boxed{P \text{ ok}}$		
<p>Programs</p> $\frac{\text{T-PROG} \quad \text{distinct}(t\text{decls}(\bar{D})) \quad \text{distinct}(m\text{decls}(\bar{D})) \quad \bar{D} \text{ ok} \quad \emptyset \vdash e : t}{\text{package main; } \bar{D} \text{ func main() } \{ _ = e \} \text{ ok}} \quad \boxed{P \text{ ok}}$		

- 型ルールの図に基づいて、ひたすら実装
なにこれ？



Figure: [1] より

型検査器の作成



- 型ルールの図に基づいて、ひたすら実装
なにこれ？



- 読めるようになると、英語（論文）が読めなくてもある程度なんとかなる（？）

Figure: [1] より

型検査器の作成

<p>Implements, well-formed type</p> $\frac{<_S}{t_S <: t_S} \quad \frac{<_I \quad \text{methods}(t) \supseteq \text{methods}(t_I)}{t <: t_I} \quad \frac{\text{T-NAMED} \quad (\text{type } t \ T) \in \bar{D}}{t \text{ ok}} \quad \boxed{t <: u} \quad \boxed{t \text{ ok}}$		
<p>Well-formed method specifications and type literals</p> $\frac{\text{T-SPECIFICATION} \quad \text{distinct}(\bar{x}) \quad \overline{t \text{ ok}} \quad t \text{ ok}}{m(\bar{x} \ T) \ t \text{ ok}} \quad \frac{\text{T-STRUCT} \quad \text{distinct}(\bar{f}) \quad \overline{t \text{ ok}}}{\text{struct } \{\bar{f} \ T\} \text{ ok}} \quad \frac{\text{T-INTERFACE} \quad \text{unique}(\bar{S}) \quad \overline{S \text{ ok}}}{\text{interface } \{\bar{S}\} \text{ ok}} \quad \boxed{S \text{ ok}} \quad \boxed{T \text{ ok}}$		
<p>Well-formed declarations</p> $\frac{\text{T-TYPE} \quad \overline{T \text{ ok}}}{\text{type } t \ T \text{ ok}} \quad \frac{\text{T-FUNC} \quad \text{distinct}(x, \bar{x}) \quad \overline{t_S \text{ ok}} \quad \overline{t \text{ ok}} \quad \overline{u \text{ ok}} \quad x : t_S, \bar{x} : \bar{T} \vdash e : t \quad t <: u}{\text{func } (x \ t_S) \ m(\bar{x} \ T) \ u \ (\text{return } e) \text{ ok}} \quad \boxed{D \text{ ok}}$		
<p>Expressions</p> $\frac{\text{T-VAR} \quad (x : t) \in \Gamma}{\Gamma \vdash x : t} \quad \frac{\text{T-CALL} \quad \Gamma \vdash e : t \quad \Gamma \vdash \bar{e} : \bar{t} \quad (m(\bar{x} \ T) \ u) \in \text{methods}(t) \quad \overline{t <: u}}{\Gamma \vdash e.m(\bar{e}) : u} \quad \boxed{\Gamma \vdash e : t}$		
<p>T-LITERAL</p> $\frac{t_S \text{ ok} \quad \Gamma \vdash \bar{e} : \bar{t} \quad (\bar{f} \ u) = \text{fields}(t_S) \quad \overline{\bar{t} <: u}}{\Gamma \vdash t_S\{\bar{e}\} : t_S} \quad \frac{\text{T-FIELD} \quad \Gamma \vdash e : t_S \quad (\bar{f} \ u) = \text{fields}(t_S)}{\Gamma \vdash e.f_i : u_i}$		
<p>T-ASSERT_I</p> $\frac{t_I \text{ ok} \quad \Gamma \vdash e : u_I}{\Gamma \vdash e.(t_I) : t_I} \quad \frac{\text{T-ASSERT}_S \quad t_S \text{ ok} \quad \Gamma \vdash e : u_I \quad t_S <: u_I}{\Gamma \vdash e.(t_S) : t_S} \quad \boxed{\text{T-STUPID} \quad \overline{t \text{ ok}} \quad \overline{\Gamma \vdash e : u_S}} \quad \boxed{\Gamma \vdash e.(t) : t}$		
<p>Programs</p> $\frac{\text{T-PROG} \quad \text{distinct}(t\text{decls}(\bar{D})) \quad \text{distinct}(m\text{decls}(\bar{D})) \quad \overline{\bar{D} \text{ ok}} \quad \emptyset \vdash e : t}{\text{package main; } \bar{D} \text{ func main() } \{ _ = e \} \text{ ok}} \quad \boxed{P \text{ ok}}$		

Figure: [1] より

- 型ルールの図に基づいて、ひたすら実装
なにこれ？



- 読めるようになると、英語（論文）が読めなくてもある程度なんとかなる（？）

日本語で学べます！



評価器の作成

評価器の作成

- 👉 評価ルール（意味論）の図に基づいて、ひたすら実装

Evaluation context		Value	$v ::= t_S\{\bar{v}\}$	
Hole		$E ::=$		
Method call receiver		$E.m(\bar{e})$	Structure	$t_S\{\bar{v}, E, \bar{e}\}$
Method call arguments		$v.m(\bar{v}, E, \bar{e})$	Select	$E.f$
			Type assertion	$E.(t)$
Reduction				$d \longrightarrow e$
$\frac{\text{R-FIELD}}{(f \ t) = \text{fields}(t_S)}$		$\frac{\text{R-CALL}}{(x : t_S, \bar{x} : \bar{t}).e = \text{body}(\text{type}(v).m)}$		$\frac{\text{R-ASSERT}}{\text{type}(v) <: t}$
$\frac{}{t_S\{\bar{v}\}.f_i \longrightarrow v_i}$		$\frac{}{v.m(\bar{v}) \longrightarrow e[x := v, \bar{x} := \bar{v}]}$		$\frac{\text{R-CONTEXT}}{d \longrightarrow e}$
				$\frac{}{E[d] \longrightarrow E[e]}$

どうして？



Figure: [1] より

評価器の作成

- 👉 評価ルール（意味論）の図に基づいて、ひたすら実装

Evaluation context		Value	$v ::= t_S\{\bar{v}\}$	
Hole		$E ::=$		
Method call receiver		$E.m(\bar{e})$	Structure	$t_S\{\bar{v}, E, \bar{e}\}$
Method call arguments		$v.m(\bar{v}, E, \bar{e})$	Select	$E.f$
			Type assertion	$E.(t)$
Reduction				$d \longrightarrow e$
R-FIELD		R-CALL		R-ASSERT
$\frac{(f \ t) = fields(t_S)}{t_S\{\bar{v}\}.f_i \longrightarrow v_i}$		$\frac{(x : t_S, \bar{x} : \bar{t}).e = body(type(v).m)}{v.m(\bar{v}) \longrightarrow e[x := v, \bar{x} := \bar{v}]}$		$\frac{type(v) <: t}{v.(t) \longrightarrow v}$
				R-CONTEXT
				$\frac{d \longrightarrow e}{E[d] \longrightarrow E[e]}$

Figure: [1] より

どうして？



日本語で学べます！



Web アプリ

Web アプリ

Scala.js[†]で簡単に
Web アプリにならないかな 🤔



Web アプリ

Scala.js[†]で簡単に
Web アプリにならないかな🤔



できそう



- 📌 というようなノリで *Scala.js* で Web アプリとなった! 🎉
- Web UI の作り方が分からなさすぎたけど、@nanashiki, @kazuma1989, @nobuoka の助けでどうにか完成🙏

<https://www.scala-js.org/>

Web アプリ

Scala.js[†]で簡単に
Web アプリにならないかな🤔



できそう



- 👉 というようなノリで Scala.js で Web アプリとなった！ 🎉
- Web UI の作り方が分からなさすぎたけど、@nanashiki, @kazuma1989, @nobuoka の助けでどうにか完成🙏

<https://www.scala-js.org/>

あなたと Featherweight Go、いますぐダウンロード (?)

https://y-yu.github.io/featherweight_go/

まとめ

まとめ

- こんな感じで Featherweight Go ができた (?)
 - https://github.com/y-yu/featherweight_go

まとめ

- こんな感じで Featherweight Go ができた (?)
 - https://github.com/y-yu/featherweight_go
- (ほぼ全て割愛したけど) 実はパーザーを除いて、他はすごくシンプルにできた

まとめ

- こんな感じで Featherweight Go ができた (?)
 - https://github.com/y-yu/featherweight_go
- (ほぼ全て割愛したけど) 実はパーザーを除いて、他はすごくシンプルにできた
- 教科書に載っていない言語をちょっと作ってみたくなったときにおすすめ

まとめ

- こんな感じで Featherweight Go ができた (?)
 - https://github.com/y-yu/featherweight_go
- (ほぼ全て割愛したけど) 実はパーザーを除いて、他はすごくシンプルにできた
- 教科書に載っていない言語をちょっと作ってみたくなったときにおすすめ
- FGG もそのうち作りたい

まとめ

- こんな感じで Featherweight Go ができた (?)
 - https://github.com/y-yu/featherweight_go
- (ほぼ全て割愛したけど) 実はパーザーを除いて、他はすごくシンプルにできた
- 教科書に載っていない言語をちょっと作ってみたくなったときにおすすめ
- FGG もそのうち作りたい

Go 言語に Generics を入れるとコンパイルが遅くなるのでは……？



まとめ

- こんな感じで Featherweight Go ができた (?)
 - https://github.com/y-yu/featherweight_go
- (ほぼ全て割愛したけど) 実はパーザーを除いて、他はすごくシンプルにできた
- 教科書に載っていない言語をちょっと作ってみたくなったときにおすすめ
- FGG もそのうち作りたい

Go 言語に Generics を入れるとコンパイルが遅くなるのでは……？



- そういうこと👉を思ったら、ぜひFGGを実装してみましょう！

まとめ

- こんな感じで Featherweight Go ができた (?)
 - https://github.com/y-yu/featherweight_go
- (ほぼ全て割愛したけど) 実はパーザーを除いて、他はすごくシンプルにできた
- 教科書に載っていない言語をちょっと作ってみたくなったときにおすすめ
- FGG もそのうち作りたい

Go 言語に Generics を入れるとコンパイルが遅くなるのでは……？



- そういうこと👉を思ったら、ぜひFGGを実装してみましょう！
- だいたいのはことはTaPL (日本語) [4] に載っています

参考文献 I

- [1] Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida.
Featherweight Go, 2020.
- [2] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler.
Featherweight Java: A Minimal Core Calculus for Java and GJ.
ACM Trans. Program. Lang. Syst., Vol. 23, No. 3, pp. 369–450, May 2001.
- [3] Amin, Nada and Grütter, Karl Samuel and Odersky, Martin and Rompf, Tiark and Stucki, Sandro.
The Essence of Dependent Object Types.
A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, pp. 249–272, 2016.

参考文献 II

- [4] Benjamin C. Pierce, 英二郎住井, 侑介遠藤, 政裕酒井, 敬吾今井, 裕介黒木, 宜洋今井, 隆文才川, 健男今井.
型システム入門: プログラミング言語と型の理論.
オーム社, 2013.

Thank you for your attention!