Featherweight Go

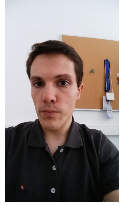
Chalmers FP, 8 June 2020

Robert Griesemer, Google

Ian Lance Taylor, Google

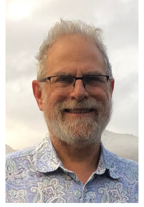
Raymond Hu, Hertfordshire

Bernardo Toninho, NOVA



Wen Kokke, Edinburgh

Philip Wadler, Edinburgh



Julien Lange, Kent

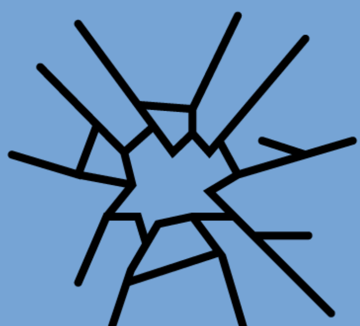
Nobuko Yoshida, Imperial

We interrupt this program …

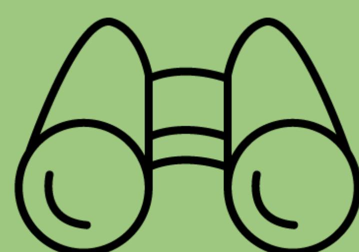
CAMPAIGN ZERO

WE CAN LIVE IN A WORLD WHERE THE POLICE DON'T KILL PEOPLE
BY LIMITING POLICE INTERVENTIONS, IMPROVING COMMUNITY INTERACTIONS
AND ENSURING ACCOUNTABILITY.

1 END BROKEN WINDOWS POLICING



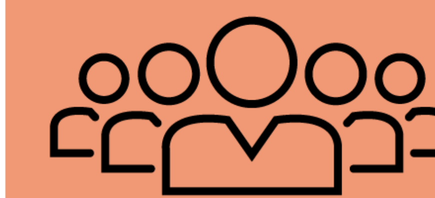
2 COMMUNITY OVERSIGHT



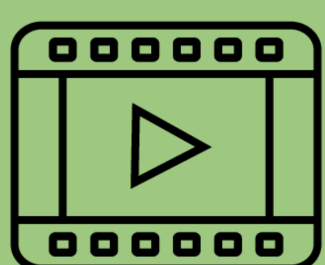
3 LIMIT USE OF FORCE

4 INDEPENDENTLY INVESTIGATE & PROSECUTE

5 COMMUNITY REPRESENTATION



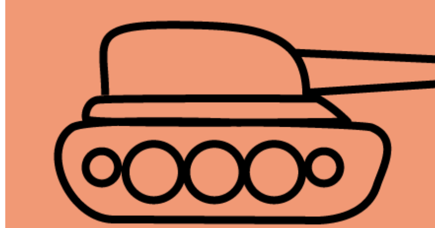
6 BODY CAMS / FILM THE POLICE



7 TRAINING

8 END FOR-PROFIT POLICING

9 DEMILITARIZATION



10 FAIR POLICE UNION CONTRACTS

WE CAN LIVE IN A WORLD WHERE SYSTEMS AND STRUCTURES DO GOOD, NOT HARM.

JOINCAMPAIGNZERO.ORG

Double blind

vs

Lightweight double blind

… We now return to our regularly scheduled programming

Any questions?

Would you be interested in helping us get polymorphism right (and/or figuring out what “right” means) for some future version of Go? — Rob Pike

Featherweight Java: A Minimal Core Calculus for Java and GJ

ATSUSHI IGARASHI
University of Tokyo
BENJAMIN C. PIERCE
University of Pennsylvania
and
PHILIP WADLER
Avaya Labs

Several recent studies have introduced lightweight versions of Java: reduced languages in which complex features like threads and reflection are dropped to enable rigorous arguments about key properties such as type safety. We carry this process a step further, omitting almost all features of the full language (including interfaces and even assignment) to obtain a small calculus, Featherweight Java, for which rigorous proofs are not only possible but easy. Featherweight Java bears a similar relation to Java as the lambda-calculus does to languages such as ML and Haskell. It offers a similar computational "feel," providing classes, methods, fields, inheritance, and dynamic typecasts with a semantics closely following Java's. A proof of type safety for Featherweight Java thus illustrates many of the interesting features of a safety proof for the full language, while remaining pleasingly compact. The minimal syntax, typing rules, and operational semantics of Featherweight Java make it a handy tool for studying the consequences of extensions and variations. As an illustration of its utility in this regard, we extend Featherweight Java with *generic classes* in the style of GJ (Bracha, Odersky, Stoutamire, and Wadler) and give a detailed proof of type safety. The extended system formalizes for the first time some of the key features of GJ.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects*;

OOPSLA, 1999; TOPLAS 2001

Java vs Go

Java	**Go**
Classes and Interfaces	Structs and Interfaces
Nominal	Structural
Closed set of super types	Open set of super types
Erasure	Monomorphisation

Java vs Go

Haskell

Datatypes and Type Classes
Structural
Closed set of super types
(*Monad* and *Functor*)

Java vs Go

Java	**Go**
Classes and Interfaces	Structs and Interfaces
Nominal	Structural
Closed set of super types	Open set of super types
Erasure	Monomorphisation

Functions in Featherweight Go

```go
type Any interface {}

type Function interface {
  Apply(x Any) Any
}
```

```
type incr struct { n int }
func (this incr) Apply(x Any) Any {
  return x.(int) + this.n
}

type pos struct {}
func (this pos) Apply(x Any) Any {
  return x.(int) > 0
}
```

```
type compose struct {
  f Function
  g Function
}
func (this compose) Apply(x Any) Any {
  return this.g.Apply(this.f.Apply(x))
}
```

```go
func main() {
  var h Function =
    compose{incr{-5},pos{}}
  var b bool = h.Apply(3).(bool)
}
```

Functions in Featherweight Generic Go

```
type Any interface {}

type Function(type a, b Any) interface {
  Apply(x a) b
}
```

```go
type incr struct { n int }
func (this incr) Apply(x int) int {
  return x + this.n
}

type pos struct {}
func (this pos) Apply(x int) bool {
  return x > 0
}
```

```
type compose(type a, b, c Any) struct {
  f Function(a, b)
  g Function(b, c)
}
func (this compose(type a, b, c Any))
    Apply(x a) c {
  return this.g.Apply(this.f.Apply(x))
}
```

```
func main() {
  var h Function(int,bool) =
    compose(int,int,bool){incr{-5},pos{}}
  var b bool = h.Apply(3)
}
```

List and Map

```
type List(type a Any) interface {
  Map(type b Any)(f Function(a, b)) List(b)
}
type Nil(type a Any) struct {}
type Cons(type a Any) struct {
  head a
  tail List(a)
}
```

```
func (xs Nil(type a Any))
    Map(type b Any)(f Function(a,b)) List(b) {
  return Nil(b){}
}
func (xs Cons(type a Any))
    Map(type b Any)(f Function(a,b)) List(b) {
  return Cons(b)
    {f.Apply(xs.head), xs.tail.Map(b)(f)}
}
```

```
func main() {
  var xs List(int)  =
    Cons(int){3, Cons(int){6, Nil(int){}}}
  var ys List(int)  = xs.Map(int)(incr{-5})
    // Cons{-2, Cons{1, Nil{}}}
  var zs List(bool) = ys.Map(bool)(pos{})
    // Cons{false, Cons{true, Nil{}}}
```

Equal

```
type Eq(type a Eq(a)) interface {
  Equal(that a) bool
}

type Int int
func (this Int) Equal(that Int) bool {
  return this == that
}

type Bool bool
func (this Bool) Equal(that Bool) bool {
  return this == that
}
```

```
type Pair(type a, b Any) struct {
  first a
  second b
}

func (this Pair(type a Eq(a), b Eq(b)))
    Equal(that Pair(a, b)) bool {
  return this.first.Equal(that.first) &&
         this.second.Equal(that.second)
}
```

Expression Problem

The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the data type, *without recompiling* existing code, and while retaining *static type safety*. — Philip Wadler, 1998

One can think of cases as rows and functions as columns in a table. In a *functional language*, the rows are fixed (cases in a datatype declaration) but it is easy to add new columns (functions). In an *object-oriented language*, the columns are fixed (methods in a class declaration) but it is easy to add new rows (subclasses). We want to make it easy to add either rows or columns.

	Eval	String
Num	1	3
Plus	2	4

Eval on Num

```
type Evaler interface {
  Eval() int
}
type Num struct {
  value int
}
func (e Num) Eval() int {
  return e.value
}
```

Eval on Plus

```
type Plus(type a Any) struct {
    left a
    right a
}
func (e Plus(type a Evaler)) Eval() int {
    return e.left.Eval() + e.right.Eval()
}
```

String on Num

```
type Stringer interface {
  String() string
}
func (e Num) String() string {
  return fmt.Sprintf("%d", e.value)
}
```

String on Plus

```
func (e Plus(type a Stringer))
     String() string {
  return fmt.Sprintf("(%s+%s)",
     e.left.String(), e.right.String())
}
```

Tie it all together

```
type Expr interface {
  Evaler
  Stringer
}
func main() {
  var e Expr = Plus(Expr){Num{1}, {Num{2}}
  var v int = e.Eval() // 3
  var s string = e.String() // "(1+2)"
}
```

Monomorphisation

```
type Top struct {}
type Function<int,int> interface {
  Apply<0> Top
  Apply(x int) int
}
type incr struct { n int }
func (this incr) Apply<0> Top {
  return Top{}
}
func (this incr) Apply(x int) int {
  return x + this.n
}
```

```
type Function<int,bool> interface {
  Apply<1> Top
  Apply(x int) bool
}
type pos struct {}
func (this pos) Apply<1> Top {
  return Top{}
}
func (this pos) Apply(x int) bool {
  return x > 0
}
```

```
type List<int> interface {
  Map<2>() Top
  Map<int>(f Function<int,int>) List<int>
  Map<bool>(f Function<int,bool>) List<bool>
}
type Nil<int> struct {}
type Cons<int> struct {
  head int
  tail List<int>
}
func (xs Nil<int>) Map<2>() Top {
  return Top{}
}
func (xs Cons<int>) Map<2>() Top {
  return Top{}
}
```

```
func (xs Nil<int>)
    Map<int>(f Function<int,int>) List<int> {
  return Nil<int>{}
}
func (xs Cons<int>)
   Map<int>(f Function<int,int>) List<int> {
 return Cons<int>
   {f.Apply(xs.head), xs.tail.Map<int>(f)}
}
func (xs Nil<int>)
   Map<bool>(f Function<int,bool>) List<bool> {
 return Nil<bool>{}
}
func (xs Cons<int>)
   Map<bool>(f Function<int,bool>) List<bool> {
 return Cons<bool>
   {f.Apply(xs.head), xs.tail.Map<bool>(f)}
}
```

```
type List<bool> interface {
  Map<3>() Top
}
type Nil<bool> struct {}
type Cons<bool> struct {
 head bool
 tail List<bool>
}
func (xs Nil<bool>) Map<3>() Top {
 return Top{}
}
func (xs Cons<bool>) Map<3>() Top {
 return Top{}
}
```

```
func main() {
 var xs List<int> =
   Cons<int>{3, Cons<int>{6, Nil<int>}}
 var ys List<int> = xs.Map<int>(incr{-5})
 var zs List<bool> = ys.Map<bool>(pos{})
}
```

FG

Featherweight Go

Formalised

Field name	f	Expression	$d, e ::=$
Method name	m	Variable	x
Variable name	x	Method call	$e.m(\overline{e})$
Structure type name	t_S, u_S	Structure literal	$t_S\{\overline{e}\}$
Interface type name	t_I, u_I	Select	$e.f$
Type name	$t, u ::= t_S \mid t_I$	Type assertion	$e.(t)$
Method signature	$M ::= (\overline{x\ t})\ t$		
Method specification	$S ::= mM$		
Type Literal	$T ::=$		
Structure	$\textbf{struct}\ \{\overline{f\ t}\}$		
Interface	$\textbf{interface}\ \{\overline{S}\}$		
Declaration	$D ::=$		
Type declaration	$\textbf{type}\ t\ T$		
Method declaration	$\textbf{func}\ (x\ t_S)\ mM\ \{\textbf{return}\ e\}$		
Program	$P ::= \textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{\_ = e\}$		

Expressions $\boxed{\Gamma \vdash e : t}$

$$\text{T-VAR}\quad \frac{(x : t) \in \Gamma}{\Gamma \vdash x : t}$$

$$\text{T-CALL}\quad \frac{\Gamma \vdash e : t \qquad \Gamma \vdash \overline{e : t} \qquad (m(\overline{x\ u})\ u) \in \mathit{methods}(t) \qquad \overline{t <: u}}{\Gamma \vdash e.m(\overline{e}) : u}$$

$$\text{T-LITERAL}\quad \frac{t_S\ \mathit{ok} \qquad \Gamma \vdash \overline{e : t} \qquad (\overline{f\ u}) = \mathit{fields}(t_S) \qquad \overline{t <: u}}{\Gamma \vdash t_S\{\overline{e}\} : t_S}$$

$$\text{T-FIELD}\quad \frac{\Gamma \vdash e : t_S \qquad (\overline{f\ u}) = \mathit{fields}(t_S)}{\Gamma \vdash e.f_i : u_i}$$

$$\text{T-ASSERT}_I\quad \frac{t_I\ \mathit{ok} \qquad \Gamma \vdash e : u_I}{\Gamma \vdash e.(t_I) : t_I}$$

$$\text{T-ASSERT}_S\quad \frac{t_S\ \mathit{ok} \qquad \Gamma \vdash e : u_I \qquad t_S <: u_I}{\Gamma \vdash e.(t_S) : t_S}$$

$$\boxed{\text{T-STUPID}\quad \frac{t\ \mathit{ok} \qquad \Gamma \vdash e : u_S}{\Gamma \vdash e.(t) : t}}$$

Reduction

$\boxed{d \longrightarrow e}$

R-FIELD

$$\frac{(\overline{f\ t}) = \mathit{fields}(t_S)}{t_S\{\overline{v}\}.f_i \longrightarrow v_i}$$

R-CALL

$$\frac{(x : t_S, \overline{x : t}).e = \mathit{body}(\mathit{type}(v).m)}{v.m(\overline{v}) \longrightarrow e[x := v, \overline{x := v}]}$$

R-ASSERT

$$\frac{\mathit{type}(v) <: t}{v.(t) \longrightarrow v}$$

R-CONTEXT

$$\frac{d \longrightarrow e}{E[d] \longrightarrow E[e]}$$

FGG
Featherweight Generic Go
Formalised

Field name	f
Method name	m
Variable name	x
Structure type name	t_S, u_S
Interface type name	t_I, u_I
Type name	$t, u ::= t_S \mid t_I$
Type parameter	α
Method signature	$M ::= (\Psi)(\overline{x\ \tau})\ \tau$
Method specification	$S ::= mM$
Type Literal	$T ::=$
Structure	$\textbf{struct}\ \{\overline{f\ \tau}\}$
Interface	$\textbf{interface}\ \{\overline{S}\}$
Declaration	$D ::=$
Type declaration	$\textbf{type}\ t(\Phi)\ T$
Method declaration	$\textbf{func}\ (x\ t_S(\Phi))\ mM\ \{\textbf{return}\ e\}$
Program	$P ::= \textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{\_ = e\}$

Type	$\tau, \sigma ::=$
Type parameter	α
Named type	$t(\overline{\tau})$
Structure type	$\tau_S, \sigma_S ::= t_S(\overline{\tau})$
Interface type	$\tau_I, \sigma_I ::= t_I(\overline{\tau})$
Interface-like type	$\tau_J, \sigma_J ::= \alpha \mid \tau_I$
Type formal	$\Phi, \Psi ::= \textbf{type}\ \overline{\alpha\ \tau_I}$
Type actual	$\phi, \psi ::= \overline{\tau}$
Expression	$e ::=$
Variable	x
Method call	$e.m(\overline{\tau})(\overline{e})$
Structure literal	$\tau_S\{\overline{e}\}$
Select	$e.f$
Type assertion	$e.(\tau)$

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

$$\textsc{T-Var}\quad \frac{(x : \tau) \in \Gamma}{\Delta; \Gamma \vdash x : \tau}$$

$$\textsc{T-Call}\quad \frac{\begin{array}{c}(m(\Psi)(\overline{x\ \sigma})\ \sigma) \in \mathit{methods}_\Delta(\tau) \\ \Delta; \Gamma \vdash e : \tau \qquad \Delta; \Gamma \vdash \overline{e : \tau} \qquad \eta = (\Psi :=_\Delta \psi) \qquad \Delta \vdash (\overline{\tau <: \sigma})[\eta]\end{array}}{\Delta; \Gamma \vdash e.m(\psi)(\overline{e}) : \sigma[\eta]}$$

$$\textsc{T-Literal}\quad \frac{\Delta \vdash \tau_S\ \mathit{ok} \quad \Delta; \Gamma \vdash \overline{e : \tau} \quad (\overline{f\ \sigma}) = \mathit{fields}(\tau_S) \quad \Delta \vdash \overline{\tau <: \sigma}}{\Delta; \Gamma \vdash \tau_S\{\overline{e}\} : \tau_S}$$

$$\textsc{T-Field}\quad \frac{\Delta; \Gamma \vdash e : \tau_S \quad (\overline{f\ \tau}) = \mathit{fields}(\tau_S)}{\Delta; \Gamma \vdash e.f_i : \tau_i}$$

$$\textsc{T-Assert}_I\quad \frac{\Delta \vdash \tau_J\ \mathit{ok} \quad \Delta; \Gamma \vdash e : \sigma_J}{\Delta; \Gamma \vdash e.(\tau_J) : \tau_J}$$

$$\textsc{T-Assert}_S\quad \frac{\Delta \vdash \tau_S\ \mathit{ok} \quad \Delta; \Gamma \vdash e : \sigma_J \quad \tau_S <: \mathit{bounds}_\Delta(\sigma_J)}{\Delta; \Gamma \vdash e.(\tau_S) : \tau_S}$$

$$\boxed{\textsc{T-Stupid}\quad \frac{\Delta \vdash \tau\ \mathit{ok} \quad \Delta; \Gamma \vdash e : \sigma_S}{\Delta; \Gamma \vdash e.(\tau) : \tau}}$$

Reduction $\boxed{d \longrightarrow e}$

R-FIELD
$$\frac{(\overline{f\ \tau}) = \mathit{fields}(\tau_S)}{\tau_S\{\overline{v}\}.f_i \longrightarrow v_i}$$

R-CALL
$$\frac{(x : \tau_S, \overline{x : \tau}).e = \mathit{body}(\mathit{type}(v).m(\psi))}{v.m(\psi)(\overline{v}) \longrightarrow e[x := v, \overline{x := v}]}$$

R-ASSERT
$$\frac{\emptyset \vdash \mathit{type}(v) <:\ \tau}{v.(\tau) \longrightarrow v}$$

R-CONTEXT
$$\frac{d \longrightarrow e}{E[d] \longrightarrow E[e]}$$

Future Work

Featherweight Go
Welterweight Go
Cruiserweight Go

Impact

I want to thank you and your team for all the type theory work on Go so far—it really helped clarify our understanding to a massive degree. So thanks! — Robert Griesemer

Featherweight Go

Robert Griesemer, Raymond Hu, Wen Kokke,
Julien Lange, Ian Lance Taylor, Bernardo Toninho,
Philip Wadler, Nobuko Yoshida

https://arxiv.org/abs/2005.11710