Featherweight Go

ROBERT GRIESEMER, Google, USA
RAYMOND HU, University of Hertfordshire, UK
WEN KOKKE, University of Edinburgh, UK
JULIEN LANGE, Royal Holloway, University of London, UK
IAN LANCE TAYLOR, Google, USA
BERNARDO TONINHO, NOVA-LINCS, FCT-NOVA, Universidade Nova de Lisboa, Portugal
PHILIP WADLER, University of Edinburgh, UK
NOBUKO YOSHIDA, Imperial College London, UK

We describe a design for generics in Go inspired by previous work on Featherweight Java by Igarashi, Pierce, and Wadler. Whereas subtyping in Java is nominal, in Go it is structural, and whereas generics in Java are defined via erasure, in Go we use monomorphisation. Although monomorphisation is widely used, we are one of the first to formalise it. Our design also supports a solution to The Expression Problem.

CCS Concepts: • **Theory of computation → Program semantics**; **Type structures**; • **Software and its engineering → Polymorphism**.

Additional Key Words and Phrases: Go, Generics, Monomorphisation

ACM Reference Format:
Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. 2020. Featherweight Go. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 149 (November 2020), 29 pages. https://doi.org/10.1145/3428217

1 INTRODUCTION

Google introduced the Go programming language in 2009 [Griesemer et al. 2009; The Go Team 2020]. Today it sits at position 12 on the Tiobe Programming Language Index and position 10 on the IEEE Spectrum Programming Language Ranking (Haskell sits at positions 41 and 29, respectively). Recently, the Go team mooted a design to extend Go with generics [Taylor and Griesemer 2019], and Rob Pike wrote Wadler to ask:

> Would you be interested in helping us get polymorphism right (and/or figuring out what "right" means) for some future version of Go?

This paper is our response to that question.

Two decades ago, Igarashi, Pierce, and Wadler [1999; 2001], introduced Featherweight Java. They considered a tiny model of Java (FJ), extended that model with generics (FGJ), and translated FGJ to

Authors' addresses: Robert Griesemer, Google, USA; Raymond Hu, University of Hertfordshire, School of Engineering and Computer Science, Hatfield, UK, r.z.h.hu@herts.ac.uk; Wen Kokke, University of Edinburgh, Laboratory for Foundations of Computer Science, Edinburgh, UK, wen.kokke@ed.ac.uk; Julien Lange, Royal Holloway, University of London, Department of Computer Science, Egham, UK, julien.lange@rhul.ac.uk; Ian Lance Taylor, Google, USA; Bernardo Toninho, Departamento de Informática, NOVA-LINCS, FCT-NOVA, Universidade Nova de Lisboa, Portugal, btoninho@fct.unl.pt; Philip Wadler, University of Edinburgh, School of Informatics, Edinburgh, UK, wadler@inf.ed.ac.uk; Nobuko Yoshida, Imperial College London, Computing, London, UK, n.yoshida@imperial.ac.uk.

2475-1421/2020/11-ART149
https://doi.org/10.1145/3428217

FJ (via *erasure*). In their footsteps, we introduce Featherweight Go. We consider a tiny model of Go (FG), extend that model with generics (FGG), and translate FGG to FG (via *monomorphisation*).

Go differs in interesting ways from Java. Subtyping in Java is nominal, whereas in Go it is structural. Casts in Java correspond to type assertions in Go: casts in Java with generics are restricted to support erasure, whereas type assertions in Go with generics are unrestricted thanks to monomorphisation. Monomorphisation is widely used, but we are among the first to formalise it. The Expression Problem was first formulated by Wadler [1998] in the context of Java, though Java never supported a solution; our design does.

We provide a full formal development: for FG and FGG, type and reduction rules, and preservation and progress; for monomorphisation, a formal translation from FGG to FG that preserves types and is a bisimulation. The complete proofs are available in [Griesemer et al. 2020].

Structural subtyping. Go is based on structures and interface types. Whereas most programming languages use *nominal subtyping*, Go is unique among mainstream typed programming languages in using *structural subtyping*. A structure implements an interface if it defines all the methods specified by that interface, and an interface implements another if the methods of the first are a superset of the methods specified by the second.

In Java, the superclasses of a class are fixed by the declaration. If lists are defined before collections, then one cannot retrofit collections as a superclass of lists—save by rewriting and recompiling the entire library. In Haskell the superclasses of a type class are fixed by the declaration. If monads are defined before functors, then one cannot retrofit functors as a superclass of monads—save by rewriting and recompiling the entire library. In contrast, in Go one might define lists or monads first, and later introduce collections or functors as an interface that the former implements—without rewriting or recompiling the earlier code.

The Expression Problem. The Expression Problem was formulated by Wadler [1998]. It gave a name to issues described by Cook [1990], Reynolds [1994], and Krishnamurthi et al. [1998], and became the basis of subsequent work by Torgersen [2004], Zenger and Odersky [2004], Swierstra [2008], and many others. Wadler defines The Expression Problem this way:

> 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.

And motivates its interest as follows:

> Whether a language can solve the Expression Problem is a salient indicator of its capacity for expression. 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.

One can come close to solving The Expression Problem in Go as it exists now, using dynamic checking via type assertions. We show how to provide a fully static solution with generics. We had to adjust our design: our first design for generics used nonvariant matching on bounds in receivers of methods, but to solve The Expression Problem we had to relax this to covariant matching.

Monomorphisation. FGJ translates to FJ via *erasure*, whereas FGG translates to FG via *monomorphisation*. Two instances `List<int>` and `List<bool>` in FGJ both translate to `List` in FJ (where `<>` are punctuation), whereas two instances `List(int)` and `List(bool)` in FGG translate to separate types `List<int>` and `List<bool>` in FG (where `()` are punctuation, but `<>` are taken as part of the name).

Erasure is more restrictive than monomorphisation. In Java with generics, a cast `(a)x` is illegal if `a` is a type variable, whereas in Go with generics, the equivalent type assertion `x.(a)` is permitted. Erasure is often less efficient than monomorphisation. In Java with generics, all type variables are boxed, whereas in Go with generics type variables may instantiate to be unboxed. However, erasure is linear in the size of the code, whereas monomorphisation can suffer an exponential blowup; and erasure is suited to separate compilation, whereas monomorphisation requires the whole program. We choose to look at monomorphisation in the first instance, because it is simple, efficient, and the first design looked at by the Go team. Other designs offer other tradeoffs, e.g., the restriction on a type assertion `x.(a)` could also be avoided by passing runtime representations of types. This solution avoids exponential blowup and offers better support for separate compilation, but at a cost in efficiency and complexity. We expect the final solution will involve a mix of both monomorphisation and runtime representations of types, see Section 8 for more details.

Template expansion in C++ [Stroustrup 2013, Chapter 26] corresponds to monomorphisation. Generics in .NET are implemented by a mixture of erasure and monomorphisation [Kennedy and Syme 2001]. The MLton compiler for Standard ML [Cejtin et al. 2000] and the Rust programming language [The Rust Team 2017] both apply techniques closely related to monomorphisation, as described by Fluet [2015] and Turon [2015] on web pages and blog posts. We say more about related work in Section 7, but we have found only a handful of peer-reviewed publications that touch on formalisation of monomorphisation. Monomorphisation is possible only when it requires a finite set of instances of types and methods. We believe we are the first to formalise computation of instance sets and determination of whether they are finite.

The bookkeeping required to formalise monomorphisation of instances and methods is not trivial. Monomorphising an interface with type parameters that contains a method with type parameters may require different instances of the interfaces to contain different instances of the methods. It took us several tries over many months to formalise it correctly. While the method for monomorphisation described here is specialised to Go, we expect it to be of wider interest, since similar issues arise for other languages and compilers such as C++, .Net, MLton, or Rust.

Featherweight vs complete. A reviewer of an earlier revision of this paper wrote:

> It is also quite common for semantics to strive for "completeness", instead of being "featherweight". There is a lot of value in having featherweight semantics, but the argument for completeness is that it helps language designers understand bad interactions between features. (For example, Amin and Tate [2016] recently showed that Java generics are unsound, but the bug is beyond the scope of Featherweight Generic Java.)

We agree with these words. Since the review was a reject, we deduce an implicit claim that it is better to be complete. Here, with respect, we disagree. We argue both "featherweight" and "complete" descriptions have value. As evidence, compare citations counts for the paper on Featherweight Java, Igarashi et al. [2001], with the four most-cited papers on more complete models, Drossopoulou and Eisenbach [1997]; Flatt et al. [1998]; Nipkow and von Oheimb [1998]; Syme [1999]: 1070 as compared with 549, 248, 174, 158, respectively (Google Scholar, April 2020).

Impact. The original proposal for generics in Go [Taylor and Griesemer 2019] was based on *contracts*, which are syntactically convenient but lack a clear semantics. One result of our work is that the new proposal [Taylor and Griesemer 2020] is based on *interfaces*, which are already well defined in Go. After we submitted the draft of this paper, Griesemer wrote to Wadler:

> 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!

Another result is the proposal for covariant receiver typing, a feature required by The Expression Problem. It is not part of the Go team's current design, but they have noted it is backward compatible and are considering adding it in the future.

In this paper we adopt the syntax originally proposed by the Go team in July 2019 [Taylor and Griesemer 2019]. In September 2020 [Taylor and Griesemer 2020], they proposed a revised syntax where type parameters are declared within square brackets and the **type** keyword is omitted. We prefer the new syntax, but retained the old since our artifact uses the old syntax and artifact evaluation was already complete.

Outline. Section 2 introduces FG and FGG, presents a solution to The Expression Problem, and introduces monomorphisation. Sections 3 and 4 present FG and FGG; we give formal rules for types and reductions, and prove preservation and progress. Section 5 presents monomorphisation, which translates FGG back to FG; we prove the translation preserves types and is a bisimulation. Section 6 describes our prototype implementation. Section 7 describes related work. Section 8 concludes. Additional examples and details of all proofs are available in [Griesemer et al. 2020].

2 FEATHERWEIGHT GO BY EXAMPLE

Formally, FG and FGG are tiny languages, containing only structures, interfaces, and methods. Our examples use features of Go missing in FG and FGG, including booleans, integers, strings, and variable bindings. We show how to declare booleans in FG and FGG in [Griesemer et al. 2020].

2.1 FG by Example

Functions in FG. Figure 1 shows higher-order functions in FG. Interface `Any` has no methods, and so is implemented by any type. Interface `Function` has a single method, `Apply(x Any) Any`, which has an argument and result of type `Any`. It is implemented by any structure that defines a method with the same name and same signature. In Go structures and methods are declared separately, as compared to Java where they are grouped together in a class declaration. We give three examples.

Structure `incr` has a single field, `n`, of type **`int`**. Its `Apply` method has receiver `this` of type `incr`, argument `x` of type `Any`, and result type `Any`, and increments its argument by `n`. You might expect the argument and result to instead have type **`int`**, but then the declared method would not implement the `Function` interface, because the method name and signature must match exactly. In the method's body, `x.(int)` is a *type assertion* that checks its argument is an integer; otherwise it *panics*, which is Go jargon for raising a runtime error. A structure is created by a *literal*, consisting of the structure name and its field values in braces. For instance, `incr{-5}.Apply(3)` returns `-2`. A field of a structure is accessed in the usual way, `this.n`. Here `this` is a variable bound to the receiver, not a keyword.

Structure `pos` has no fields, and its apply method returns `true` if given a positive integer. Structure `compose` has two fields, each of which is a function, and its apply method applies the two in succession. The top-level `main` method composes `incr{-5}` with `pos{}` and applies it to `3`, yielding false. One cannot pass a value of type `Any` where a boolean is expected, so the type assertion `.(bool)` is required.

Bound variable names are irrelevant when comparing method signatures, but method names and type names must match exactly. For example, the following signatures are considered equivalent:

```
Apply(x Any) Any          Apply(arg Any) Any
```

Equality in FG. Figure 2 shows equality in FG. Interface `Eq` has one method with signature `Equal(that Eq) bool`. If a type implements this interface we say it supports equality.

A type declaration introduces `Int` as a synonym for integers, and a method declaration ensures that type supports equality. Since signatures must match exactly, in the method the argument has type `Eq` and the body uses a type assertion to convert it to `Int` as required.

```
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))
}
func main() {
  var f Function = compose{incr{-5},pos{}}
  var _ bool = f.Apply(3).(bool) // false
}
```

Fig. 1. Functions in FG

```
type Eq interface {
  Equal(that Eq) bool
}
type Int int
func (this Int) Equal(that Eq) bool {
  return this == that.(Int)
}
type Pair struct {
  left Eq
  right Eq
}
func (this Pair) Equal(that Eq) bool {
  return this.left.Equal(that.(Pair).left) &&
    this.right.Equal(that.(Pair).right)
}
func main() {
  var i, j Int = 1, 2
  var p Pair = Pair{i, j}
  var _ bool = p.Equal(p) // true
}
```

Fig. 2. Equality in FG

```
type List interface {
  Map(f Function) List
}
type Nil struct {}
type Cons struct {
  head Any
  tail List
}
func (xs Nil) Map(f Function) List {
  return Nil{}
}
func (xs Cons) Map(f Function) List {
  return Cons{f.Apply(xs.head), xs.tail.Map(f)}
}
func main() {
  var xs List = Cons{3, Cons{6, Nil{}}}
  var ys List = xs.Map(incr{-5})
    // Cons{-2, Cons{1, Nil{}}}
  var _ List = ys.Map(pos{})
    // Cons{false, Cons{true, Nil{}}}
}
```

Fig. 3. Lists in FG

A second type declaration introduces the structure `Pair` with two fields `left` and `right` which may be of any type that supports equality, and the method declaration ensures that pairs themselves support equality. Again, the argument has type `Eq` and the body uses a type assertion to convert it to a `Pair` as required. The top level `main` method builds a pair of integers and compares it to itself for equality, yielding `true`.

Since pairs are to support equality, their components are also required to support equality. In general, if a structure is to satisfy some interface we may need to require that each field of that structure satisfies the same interface—a property we refer to as *type pollution*. An alternative design

would give fields the type `Any`, and to replace `this.left` by `this.left.(Eq)`, and similarly for the other component. The alternative design is more flexible—it permits fields of the pair to have any type—but is less efficient (the type assertions must be checked at runtime) and less reliable (the type assertions may fail). As we will see, FGG will let us avoid type pollution, providing flexibility, efficiency, and reliability all at the same time.

Lists in FG. Figure 3 shows lists in FG. Interface `List` has a single method: `Map(f Function) List`, which applies its argument to each element of its receiver. We define two structures that implement the list interface. Structure `Nil` has no fields, while structure `Cons` has two fields, a head of any type and a tail which is a list. The methods to define `Map` are straightforward, and the main method shows an example of its use.

Go is designed to enable efficient implementation. Structures are laid out in memory as a sequence of fields, while an interface is a pair of a pointer to an underlying structure and a pointer to a dictionary of methods. To ensure the layout of a structure is finite, a structure that recurses on itself is forbidden. Thus, the declaration

```
type Bad struct { oops Bad }
```

is not allowed, and similarly for mutual recursion. However, structures that recurse through an interface are permitted, such as

```
type Cons struct { head Any; tail List }
```

where the `tail` field of type `List` may itself contain a `Cons`, since `Cons` implements interface `List`.

2.2 FGG by Example

We now adapt the examples of the previous section to generics.

Functions in FGG. Figure 4 shows higher-order functions in FGG. The interface for functions now takes two type parameters, `Function(type a Any, b Any)`. Each type parameter is followed by an interface it must implement, called its *bound*. Here the bounds indicate that the argument and result may be of any type. The signature for the method is now `Apply(x a) b`, where the first type parameter is the argument type and the second the result type.

Structures `incr` and `pos` are as before. However, they now have more natural signatures for their apply methods, where all occurrences of `Any` are replaced by `int` or `bool` as appropriate. Type assertions in the method bodies are no longer needed, and the types ensure a panic never occurs.

The structure for composition now takes three type parameters. In the `main` method, type parameters are added and the type assertion at the end is no longer required.

Equality in FGG. Figure 5 shows equality in FGG. The interface for equality is now written `Eq(type a Eq(a))`. It accepts a type parameter `a` where the bound is itself `Eq(a)`. The method has signature `Equal(that a) bool`. The situation where a type parameter appears in its own bound is known as *F-bounded polymorphism* [Canning et al. 1989], and a similar idiom is common in Java with generics [Bracha et al. 1998; Naftalin and Wadler 2006].

Since we use a type parameter for the argument to `Equal`, in the method declaration for `Int` the argument must now have type `Int` instead of type `Eq`. A type assertion in the method body is no longer required, increasing efficiency and reliability.

The type declaration for `Pair` now take two type parameters, `a` and `b`, which are both bounded by `Any`. The method declaration for equality on pairs also uses two type parameters, `a` and `b`, bounded by `Eq(a)` and `Eq(b)` respectively, so the call to `Equality` on the components of the pair is permitted.

Crucially, FGG permits the bounds on the type parameter in a receiver to *implement* the bound on the type parameter in the corresponding structure declaration (receiver type parameters are

```
type Function(type a Any, b Any) interface {
  Apply(x a) b
}
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 Any, b Any, c Any) struct {
  f Function(a, b)
  g Function(b, c)
}
func (this compose(type a Any, b Any, c Any))
    Apply(x a) c {
  return this.g.Apply(this.f.Apply(x))
}
func main() {
  var f Function(int,bool) =
    compose(int,int,bool){incr{-5},pos{}}
  var _ bool = f.Apply(3) // false
}
```

Fig. 4. Functions in FGG

```
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 Pair(type a Any, b Any) struct {
  left a
  right b
}
```

```
func (this Pair(type a Eq(a), b Eq(b)))
    Equal(that Pair(a,b)) bool {
  return this.left.Equal(that.left) &&
    this.right.Equal(that.right)
}
func main() {
  var i, j Int = 1, 2
  var p Pair(Int, Int) = Pair(Int, Int){i, j}
  var _ bool = p.Equal(p) // true
}
```

Fig. 5. Equality in FGG

```
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})
  var _ List(bool) = ys.Map(bool)(pos{})
}
```

Fig. 6. Lists in FGG

```
type Edge(type e Edge(e,v),
          v Vertex(e,v)) interface {
   Source() v
   Target() v
}
```

```
type Vertex(e Edge(e,v),
            v Vertex(e,v)) interface {
   Edges() List(e)
}
```

Fig. 7. Mutually recursive bounds

covariant). This is in contrast to method signatures, which must exactly match the signature in the interface (signatures are *nonvariant*). In this case, the bounds on `a` and `b` in the type declaration for pairs are both `Any`, while the bounds on `a` and `b` in the receiver are `Eq(a)` and `Eq(b)`. By covariance, since `Eq(a)` and `Eq(b)` implement `Any`, the method declaration is allowed.

The Go team's current design does not support covariant receiver typing. Instead, receivers are nonvariant, just like method signatures. With that design, the bounds on `a` and `b` in the declaration for pairs must exactly match those in the method receiver. Either the type and method declarations must both use bounds `Eq(a)` and `Eq(b)` (in which case one cannot have pairs where the components do not support equality, even if don't need that pair to itself support equality, reintroducing type pollution and reducing flexibility), or they must both use bounds `Any` (in which case the method body will need to add type assertions to `Eq(a)` and `Eq(b)`, reducing efficiency and reliability).

Lists in FGG. Figure 6 shows lists in FGG. Interface `List` now takes as a parameter the type of the elements of the list, bounded by `Any`. The signature for the map method is now `Map(type b Any)(f Function(a, b)) List(b)`. The list interface takes a parameter `a` for the type of elements of the receiver list, while the map method itself takes an additional parameter `b` for the type of elements of the result list.

The two structures that implement lists now also take a type parameter, again bounded by `Any`. It may seem odd that `Nil` requires a parameter, since it represents a list with no elements. However, without this parameter we could not declare that `Nil` has method `Map`, whose signature mentions the type of the list elements.

The main method simply adds type parameters. In Go, no name can be bound to both a value and a type in a given scope, so it is always unambiguous as to whether one is parsing a type or an expression. In practice, writing out all type parameters in full can be tedious, and generic Go permits such parameters to be omitted when they can be inferred. Here we always require type parameters, leaving inference for future work.

Type parameter names and variable names are irrelevant when comparing method signatures, but method names, bounds on type parameters, and type names must match exactly. For example, the following two signatures are considered equivalent:

```
Map(type b Any)(f Function(a, b)) List(b)   Map(type bob Any)(fred Function(a, bob)) List(bob)
```

If we wanted to define equality on lists without type assertions, we would need to bound the elements of the list so that they support equality, changing every occurrence of `(type a Any)`, in the code to `(type a Eq(a))`, and similarly for `b` in the signature of `Map`. This is a form of type pollution. An alternative design that avoids pollution, based on our solution to The Expression Problem, can be found in [Griesemer et al. 2020].

Mutual recursion in type bounds. In a declaration that introduces a list of type parameters, the bounds of each may refer to any of the others. Figure 7 shows two mutually-recursive interface declarations that may be useful in representing graphs. It is parameterised over types for edges and vertexes. Each edge has a source and target vertex, while each vertex has a list of edges.

2.3 The Expression Problem

Following Wadler [1998], we present The Expression Problem pared to a minimum. Our solution appears in Figure 8. There are just two structures that construct expressions, `Num` and `Plus`, which denote numbers and the sum of two expressions, respectively; and two methods that operate on expressions, `Eval` and `String`, which evaluate an expression and convert it to a string, respectively. We show that each constructor and operation can be added independently, proceeding in four steps:

```
// 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 _ Int = e.Eval() // 3
  var _ string = e.String() // "(1+2)"
}
```

Fig. 8. Expression problem in FGG

(1) define `Eval` on `Num`
(2) define `Eval` on `Plus`
(3) define `String` on `Num`
(4) define `String` on `Plus`

The order of steps 2 and 3 can be reversed: we may extend either by adding a new constructor or a new operation. We assume availability of the library function `fmt.Sprintf` to format strings.

Structure `Num` contains an integer value. Structure `Plus` contains two fields, left and right, which are themselves expressions. Typically, we might expect the type of these fields to be an interface specifying all operations we wish to perform on expressions. But the whole point of the expression problem is that we may add other operations later! Thus, plus takes a type parameter for the type of these fields. We bound the type parameter with `Any`, permitting it to be instantiated by any type.

For each operation, `Eval` and `String`, we define a corresponding interface to specify that operation, `Evaler` and `Stringer`. (The naming convention is typical of Go.) When defining `Eval` on `Plus`, the receiver's type parameter is bounded by interface `Evaler`, allowing `Eval` to be recursively invoked on the left and right expressions. Similarly, when defining `String` on `Plus`, the receiver's type parameter is bounded by interface `Stringer`, allowing `String` to be recursively invoked. Note this depends crucially on FGG's support for covariant receivers in method declarations. Since the bounds or the receivers in the method declarations, `Evaler` and `Stringer`, implement the bounds in the type declarations, `Any`, the method declarations are allowed.

A last step shows how to tie it all together. We define an interface `Expr` embedding `Evaler` and `Stringer`, and show how to build an `Expr` value which we can both evaluate and convert to a string.

How close could we get without generics? If we know all operations in advance, then in place of the type parameter in `Plus` we can use interface `Expr`, defining all required operations; but that violates the requirement that we can add operations later. Alternatively, in place of the type parameter in `Plus` we can use interface `Any`, with type assertions to `Evaler` or `Stringer` before the

```
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 _ List<bool> = ys.Map<bool>(pos{})
}
```

Fig. 9. Monomorphisation: example of FG translation

```
type Box(type a Any) struct {
  value a
}
func (this Box(type a Any)) Nest(n int) Any {
  if (n == 0) { return this }
  else { return Box(Box(a)){this}.Nest(n-1) }
}
```

Fig. 10. FGG code that cannot be monomorphised

recursive calls; that allows us to add operations later, but violates the requirement that all types be checked statically.

2.4 Monomorphisation by Example

We translate FGG into FG via *monomorphisation*. As an example, consider the FGG code in Figures 4 and 6. We only include the code relevant to the main method, so omit composition and equality. The given code monomorphises to the FG program shown in Figure 9.

Each parametric type and method in FGG is translated to a family of types and methods in FG, one for each possible instantiation of the type parameters. FGG type `List(a)` is instantiated at types **`int`** and **`bool`**, so it translates to the two FG types `List<int>` and `List<bool>`. For convenience, we assume that angle brackets and commas "`<,>`" may appear in FG identifiers, although that is not allowed in Go. In our prototype, we use Unicode letters that resemble angle brackets and a dash: Canadian Syllabics Pa (U+1438), Po (U+1433), and Final Short Horizontal Stroke (U+1428).

Monomorphisation tracks for each method the possible types of its receiver and type parameters. In this particular program, we need two instances of `Map` over lists of integers, one that yields a list of integers and one that yields a list of booleans, and none for `Map` over lists of booleans.

Each interface also contains an instance of a *dummy* version of `Apply` or `Map`, here called, e.g., `Map<2>`, where the number in brackets stands for a hash computed from the method signature. A dummy method is provided for every source FGG method; these dummy methods are needed to ensure a correct implementation relation between structures and interfaces is maintained at runtime. For instance, if `f` is bound to `incr{1}` then the type assertion `f.(List<bool>)` should fail; but without the dummy, interface `List<bool>` would have no methods and hence any structure or interface would implement it.

Monomorphisation yields specialised type declarations for structures and interfaces, and specialised method declarations, plus the required dummy methods. The source FGG and its translation to FG are both well-typed, and both evaluate to corresponding terms: we will show the translation preserves typing and is a bisimulation.

Not all typable FGG source can be monomorphised. Figure 10 shows a program that exhibits *polymorphic recursion*, where a method called at one type recursively calls itself at a different type. Here, calling method `Nest` on a receiver of type `Box(a)` leads to a recursive call on a receiver of type `Box(Box(a))`. Monomorphisation is impossible because we cannot determine in advance to what depth the types will nest. We will present a theorem stating that if source code does not exhibit problematic polymorphic recursion then it can be monomorphised.

3 FEATHERWEIGHT GO

3.1 FG Syntax

Figure 11 presents FG syntax. We let f range over field names, m range over method names, x range over variable names, t_S, u_S range over structure names, and t_I, u_I range over interface names.

We let t, u range over type names, which are either structure or interface type names. We let d and e range over expressions, which have five forms: variable x, method call $e.m(\overline{e})$, structure literal $t_S\{\overline{e}\}$, selection $e.f$, and type assertion $e.(t)$. By convention, $\overline{e}$ stands for the sequence $e_1, \ldots, e_n$. We consider e and $\overline{e}$ to be distinct metavariables.

A method signature M has the form $(\overline{x\ t})\ t$. Here $\overline{x}$ stands for $x_1, \ldots, x_n$ and $\overline{t}$ stands for $t_1, \ldots, t_n$, and hence $\overline{x\ t}$ stands for $x_1\ t_1, \ldots, x_n\ t_n$. We use similar conventions extensively.

A method specification S is a method name followed by a method signature. A type literal T is either a structure **struct** $\{\overline{f\ t}\}$ or an interface **interface** $\{\overline{S}\}$. A declaration D is either a type declaration **type** $t\ T$ or a method declaration **func** $(x\ t_S)\ mM\ \{$**return** $e\}$. In our examples, interface declarations may contain interface embeddings, i.e., a reference to another interface; for our formalism, we assume these are always expanded out to the corresponding method specifications.

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	**struct** $\{\overline{f\ t}\}$		
Interface	**interface** $\{\overline{S}\}$		
Declaration	$D ::=$		
Type declaration	**type** $t\ T$		
Method declaration	**func** $(x\ t_S)\ mM\ \{$**return** $e\}$		
Program	$P ::=$ **package main**; $\overline{D}$ **func main**() $\{\_ = e\}$		

Fig. 11. FG syntax

A program P consists of a sequence of declarations $\overline{D}$ and a top-level expression e, written in the stylised form shown in the figure to make it legal Go. We sometimes abbreviate it as $\overline{D} \triangleright e$.

3.2 Auxiliary Functions

Figure 12 presents several auxiliary definitions. All definitions assume a given program with a fixed sequence of declarations $\overline{D}$.

Function $\mathit{fields}(t_S)$ looks up the structure declaration for t_S and returns a sequence $(\overline{f\ t})$ of field names and their types. Write $t_S.m$ to refer to the method declaration with receiver type t_S and name m. Function $\mathit{body}(t_S.m)$ returns $(x : t_S, \overline{x : t}).e$, where $x : t_S$ is the receiver parameter and its type, $\overline{x : t}$ the argument parameters and their types, and e the body from the declaration of a method with receiver of type t_S and name m. In the phrase **func** $(x\ t_S)\ m(\overline{x\ t})\ t$, each of t_S, $\overline{t}$, and t is considered a distinct metavariable, and similarly for x and $\overline{x}$.

Function $\mathit{type}(v)$ is explained in Section 3.4. Predicate $\mathit{unique}(\overline{S})$ holds if for every method specification mM in $\overline{S}$ the method name m uniquely determines the method signature M.

Function $\mathit{tdecls}(\overline{D})$ returns a sequence with the name of every type declared in $\overline{D}$. Function $\mathit{mdecls}(\overline{D})$ returns a sequence with a pair $t_S.m$ for every method declared in $\overline{D}$. Predicate *distinct*, not defined in the figure, takes a sequence, and holds if no item in the sequence is duplicated. We are careful to distinguish between sets and sequences. Predicate *distinct* must take a sequence rather than a set, because items in a set are distinct by definition. Sequences may implicitly coerce to sets, but not vice-versa. When comparing method signatures, the names of formal parameters are ignored; signatures are considered equal if they contain the same types in the same sequence.

Function $\mathit{methods}(t)$ returns the set of all method specifications belonging to type t. If t is a structure type the method specifications are those from the method declarations with a receiver of the given type. It t is an interface type, the method specifications are those given in the interface.

Figure 13 presents the FG typing rules. Let Γ range over environments, which are sequences of variables paired with type names, $\overline{x : t}$. We write $\emptyset$ for the empty environment.

Judgement $t <: u$ holds if type t *implements* type u. A structure type t_S is only implemented by itself, while type t implements interface type t_I if the methods defined on t are a superset of those defined on t_I. It follows from the definition that $<:$ is reflexive and transitive.

$$\frac{(\textbf{type}\ t_S\ \textbf{struct}\{\overline{f\ t}\}) \in \overline{D}}{\mathit{fields}(t_S) = \overline{f\ t}} \qquad \frac{(\textbf{func}\ (x\ t_S)\ m(\overline{x\ t})\ t\ \{\textbf{return}\ e\}) \in \overline{D}}{\mathit{body}(t_S.m) = (x : t_S, \overline{x : t}).e}$$

$$\mathit{type}(t_S\{\overline{v}\}) = t_S \qquad \frac{mM_1, mM_2 \in \overline{S}\ \text{implies}\ M_1 = M_2}{\mathit{unique}(\overline{S})}$$

$$\mathit{tdecls}(\overline{D}) = [t \mid (\textbf{type}\ t\ T) \in \overline{D}] \qquad \mathit{mdecls}(\overline{D}) = [t_S.m \mid (\textbf{func}\ (x\ t_S)\ mM\ \{\textbf{return}\ e\}) \in \overline{D}]$$

$$\mathit{methods}(t_S) = \{mM \mid (\textbf{func}\ (x\ t_S)\ mM\ \{\textbf{return}\ e\}) \in \overline{D}\} \qquad \frac{\textbf{type}\ t_I\ \textbf{interface}\{\overline{S}\} \in \overline{D}}{\mathit{methods}(t_I) = \overline{S}}$$

Fig. 12. FG auxiliary functions

3.3 FG Typing

We write *ok* to indicate a construct is well-formed. Judgement $t\ ok$ holds if type t is declared. Judgement $S\ ok$ holds if method specification S is well formed: all formal parameters $\overline{x}$ in it are distinct, and all the types $\overline{t}, t$ in it are declared. Judgement $T\ ok$ holds if type literal T is well formed: for a structure, all field names must be distinct and all types declared; for an interface, all its method specifications must be well formed. Judgement $D\ ok$ holds if declaration D is well formed: for a type declaration, its type literal must be well formed; for a method declaration, its receiver and formal parameters must be distinct, all types must be declared, the method body must be well typed in the appropriate environment, and the expression type must implement the declared return type.

Judgement $\Gamma \vdash e : t$ holds if in environment Γ expression e has type t. Rules for variable, method call, structure literal, and field selection are straightforward. For instance, the four hypotheses for a method call check the type of the receiver, check the types of the arguments, look up the signature of the method, and confirm the types of the arguments implement the types of the parameters.

Type assertions are more interesting. A type assertion $e.(t)$ always returns a value of type t (if it doesn't panic). Let e have type u. There are three cases. If u and t are both interface types (T-ASSERT$_I$) then the assertion is always allowed, since e could always conceivably evaluate to a structure that implements t. If u is an interface but t is a structure (T-ASSERT$_S$) then the assertion is allowed only if t implements u, since otherwise e could not possibly contain a structure of type t. If u is a structure type (T-STUPID) then it is stupid to write the assertion in source code, since the assertion could be checked at compile time, making it pointless. Nonetheless, during reduction a variable of interface type will be replaced by a value of structure type, so without such stupid type assertions an expression would become ill-typed during reduction. We write a box around this rule to indicate that it doesn't apply to source terms, but may apply to terms that result from reducing the source. Stupid type assertions are similar to stupid casts as found in Featherweight Java.

Judgement $P\ ok$ holds if program P is well formed: all its type declarations are distinct, all its method declarations are distinct (each pair of a receiver type with a method name is distinct), all its declarations are well formed, and its body is well typed in the empty environment.

3.4 FG Reduction

Figure 14 presents the FG reduction rules. A value v is a structure literal $t_S\{\overline{v}\}$ where each field is itself filled with a value. The auxiliary function $\mathit{type}(v)$ returns t_S when $v = t_S\{\overline{v}\}$. Evaluation contexts E are defined in the usual way. Judgement $d \longrightarrow e$ holds if expression d steps to expression

Implements, well-formed type $\boxed{t <: u}$ $\boxed{t\ ok}$

$$\frac{}{t_S <: t_S}\ {}^{<:S} \qquad \frac{methods(t) \supseteq methods(t_I)}{t <: t_I}\ {}^{<:I} \qquad \frac{(\textbf{type}\ t\ T) \in \overline{D}}{t\ ok}\ \text{T-NAMED}$$

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

$$\frac{distinct(\overline{x}) \qquad \overline{t\ ok} \qquad t\ ok}{m(\overline{x\ t})\ t\ ok}\ \text{T-SPECIFICATION} \qquad \frac{distinct(\overline{f}) \qquad \overline{t\ ok}}{\textbf{struct}\ \{\overline{f\ t}\}\ ok}\ \text{T-STRUCT} \qquad \frac{unique(\overline{S}) \qquad \overline{S\ ok}}{\textbf{interface}\ \{\overline{S}\}\ ok}\ \text{T-INTERFACE}$$

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

$$\frac{T\ ok}{\textbf{type}\ t\ T\ ok}\ \text{T-TYPE} \qquad \frac{distinct(x, \overline{x}) \qquad t_S\ ok \qquad \overline{t\ ok} \qquad u\ ok \qquad x : t_S, \overline{x : t} \vdash e : t \qquad t <: u}{\textbf{func}\ (x\ t_S)\ m(\overline{x\ t})\ u\ \{\textbf{return}\ e\}\ ok}\ \text{T-FUNC}$$

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

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

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

$$\frac{t_I\ ok \qquad \Gamma \vdash e : u_I}{\Gamma \vdash e.(t_I) : t_I}\ \text{T-ASSERT}_I \qquad \frac{t_S\ ok \qquad \Gamma \vdash e : u_I \qquad t_S <: u_I}{\Gamma \vdash e.(t_S) : t_S}\ \text{T-ASSERT}_S \qquad \boxed{\frac{t\ ok \qquad \Gamma \vdash e : u_S}{\Gamma \vdash e.(t) : t}\ \text{T-STUPID}}$$

Programs $\boxed{P\ ok}$

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

Fig. 13. FG typing

e. There are four rules, for field selection, method call, type assertion, and closure under evaluation contexts. All are straightforward.

3.5 FG Properties

We have the usual results relating typing and reduction.

Lemma 3.1 (Well formed). *If* $\Gamma \vdash e : t$ *then* $t\ ok$.

The substitution lemma is straightforward. It is sufficient to consider empty environments for the substituted terms, since FG has no binding constructs (such as lambda) in expressions.

	Value	$v ::= t_S\{\overline{v}\}$	
Evaluation context	$E ::=$		
Hole	$\Box$	Structure	$t_S\{\overline{v}, E, \overline{e}\}$
Method call receiver	$E.m(\overline{e})$	Select	$E.f$
Method call arguments	$v.m(\overline{v}, E, \overline{e})$	Type assertion	$E.(t)$

Reduction $\boxed{d \longrightarrow e}$

$$\textsc{r-field}\ \frac{(\overline{f\ t}) = \mathit{fields}(t_S)}{t_S\{\overline{v}\}.f_i \longrightarrow v_i} \qquad \textsc{r-call}\ \frac{(x : t_S, \overline{x : t}).e = \mathit{body}(\mathit{type}(v).m)}{v.m(\overline{v}) \longrightarrow e[x := v, \overline{x := v}]} \qquad \textsc{r-assert}\ \frac{\mathit{type}(v) <: t}{v.(t) \longrightarrow v} \qquad \textsc{r-context}\ \frac{d \longrightarrow e}{E[d] \longrightarrow E[e]}$$

Fig. 14. FG reduction

Lemma 3.2 (Substitution). *If $\emptyset \vdash \overline{v : t}$ and $\overline{x : u} \vdash e : u$ and $\overline{t <: u}$ then $\emptyset \vdash e[\overline{x := v}] : t$ for some type t with $t <: u$.*

The following are straightforward adaptions of the usual results. We say expression e *panics* if there exist evaluation context E, value v, and type t such that $e = E[v.(t)]$ and $\mathit{type}(v) \not<: t$.

Theorem 3.3 (Preservation). *If $\emptyset \vdash d : u$ and $d \longrightarrow e$ then $\emptyset \vdash e : t$ for some t with $t <: u$.*

Theorem 3.4 (Progress). *If $\emptyset \vdash d : u$ then either d is a value, $d \longrightarrow e$ for some e, or d panics.*

4 FEATHERWEIGHT GENERIC GO

4.1 FGG Syntax

Figure 15 presents FGG syntax, with the differences from FG syntax highlighted. We let α range over type parameters and let τ, σ range over types. A type is either a type parameter α or a named type $t(\overline{\tau})$. We also let τ_S, σ_S range over structure types of the form $t_S(\overline{\tau})$; τ_I, σ_I range over interface types of the form $t_I(\overline{\tau})$; and, τ_J, σ_J range over types that are either type parameters or interfaces τ_I.

Expressions and declarations are updated to replace type names by types, and method calls are updated to include type parameters: a structure declaration is now **struct** $\{\overline{f\ \tau}\}$ and a method call is now $e.m(\overline{\tau})(\overline{e})$, a structure literal is now $\tau_S\{\overline{e}\}$, and a type assertion is now $e.(\tau)$.

We let Φ, Ψ range over type formals, which have the form **type** $\overline{\alpha\ \tau_I}$, pairing type parameters with their bounds, which are interface types. The bounds in type formals are mutually recursive, i.e., each interface in $\overline{\tau_I}$ may refer to any parameter in $\overline{\alpha}$. Type declarations **type** $t(\Phi)\ T$, and signatures $(\Psi)(\overline{x\ \tau})\ \tau$, and method declarations **func** $(x\ t_S(\Phi))\ mM$ {**return** e} now include type formals.

We let ϕ, ψ range over type actuals, which are sequences of types.

4.2 Auxiliary Functions

Figure 16 presents several auxiliary definitions. As before, Γ ranges over environments, which are now sequences that pair variables with types, $\overline{x : \tau}$. In addition, Δ ranges over type environments, which are sequences that pair type parameters with bounds, $\overline{\alpha : \tau_I}$. Type formals Φ, Ψ may implicitly coerce to type environments.

We write $\eta = (\Phi := \phi)$ for the substitution of formals Φ by actuals ϕ, and $\eta = (\Phi :=_\Delta \phi)$ for the partial function that also checks that ϕ respects the bounds imposed by Φ. If a partial function that is undefined appears in the hypothesis of a rule, then the corresponding premise does not hold. We write $\hat{\Phi}$ for the type parameters of Φ.

Functions $\mathit{fields}(\tau_S)$ and $\mathit{body}(\tau_S.m(\psi))$ are updated to replace type names by types, and for the latter to include method type arguments. The definitions are adjusted to include type formals

Field name	f	Type	$\tau, \sigma ::=$
Method name	m	Type parameter	α
Variable name	x	Named type	$t(\overline{\tau})$
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	$\tau_I, \sigma_I ::= t_I(\overline{\tau})$
Type name	$t, u ::= t_S \mid t_I$	Interface-like type	$\tau_J, \sigma_J ::= \alpha \mid \tau_I$
Type parameter	α	Type formal	$\Phi, \Psi ::= \textbf{type}\ \overline{\alpha\ \tau_I}$
Method signature	$M ::= (\Psi)(\overline{x\ \tau})\ \tau$	Type actual	$\phi, \psi ::= \overline{\tau}$
Method specification	$S ::= mM$	Expression	$e ::=$
Type Literal	$T ::=$	Variable	x
Structure	$\textbf{struct}\ \{\overline{f\ \tau}\}$	Method call	$e.m(\overline{\tau})(\overline{e})$
Interface	$\textbf{interface}\ \{\overline{S}\}$	Structure literal	$\tau_S\{\overline{e}\}$
Declaration	$D ::=$	Select	$e.f$
Type declaration	$\textbf{type}\ t(\Phi)\ T$	Type assertion	$e.(\tau)$
Method declaration	$\textbf{func}\ (x\ t_S(\Phi))\ mM\ \{\textbf{return}\ e\}$		
Program	$P ::= \textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{\_ = e\}$		

Fig. 15. FGG syntax

which are instantiated appropriately. Functions *type*, *tdecls*, and *mdecls* and predicate *unique* are updated to replace type names by types and to include type formals. Function $bounds_\Delta(\tau)$ takes a type parameter to its bound, and leaves its argument unchanged otherwise.

Function $methods_\Delta(\tau)$ is updated to accept a type environment and to replace type names by types. The definition is adjusted to include type formals which are instantiated appropriately. If *methods* is applied to a type parameter, that parameter behaves the same as its bounding interface, so $methods_\Delta(\tau) = methods_\Delta(bounds_\Delta(\tau))$ for all τ.

4.3 FGG Typing

Figure 17 presents the FGG typing rules. Judgement $\Delta \vdash \tau <: \sigma$ now depends on a type environment and relates types rather than type names. The definition is adjusted so that a type parameter implements its bound. It still follows from the definition that $<:$ is reflexive and transitive. Judgement $\Phi <: \Psi$ compares the corresponding bounds of two type formals under an empty type environment.

Judgement $\Delta \vdash \tau\ ok$ holds if a type is well formed: all type parameters in it must be declared in Δ and all named types must be instantiated with type arguments that satisfy the bounds of the corresponding type parameters. Judgement $\Delta \vdash \phi\ ok$ holds if under environment Δ all types in ϕ are well formed.

Judgement $\Phi \vdash \Psi\ ok$ holds if under type environment Φ the type formal Ψ is well formed: all type parameters bound by Φ and Ψ are distinct, and all the bounds in Ψ are well formed in the type environment that results from combining Φ and Ψ. Note this permits mutually recursive bounds in a type formal. Judgement $\Phi; \Psi\ ok\ \Delta$ holds if a method declaration with receiver formals Φ and method formals Ψ is well formed, yielding type environment Δ: it requires that Φ is well formed under the empty environment, Ψ is well formed under Φ, and Δ is the concatenation of Φ and Ψ. Hence, the type formals of the receiver are in scope when declaring the type formals of the method, but not vice versa, and both are in scope for declaring the types of the arguments and result.

Judgement $\Phi \vdash S\ ok$ holds if under type environment Φ method specification S is well formed: it requires $\Phi, \Psi\ ok\ \Delta$ where Ψ is the type formals of the method specification, and the rest is similar

$$\frac{(\textbf{type}\ \overline{\alpha\ \tau_I}) = \Phi \qquad \eta = (\overline{\alpha := \tau})}{(\Phi := \overline{\tau}) = \eta} \qquad \frac{(\textbf{type}\ \overline{\alpha\ \tau_I}) = \Phi \qquad \eta = (\Phi := \phi) \qquad \Delta \vdash (\overline{\alpha <: \tau_I})[\eta]}{(\Phi :=_\Delta \phi) = \eta}$$

$$\frac{(\textbf{type}\ t_S(\Phi)\ \textbf{struct}\ \{\overline{f\ \tau}\}) \in \overline{D} \qquad \eta = (\Phi := \phi)}{\mathit{fields}(t_S(\phi)) = (\overline{f\ \tau})[\eta]}$$

$$\frac{(\textbf{func}\ (x\ t_S(\Phi))\ m(\Psi)(\overline{x\ \tau})\ \tau\ \{\textbf{return}\ e\}) \in \overline{D} \qquad \theta = (\Phi, \Psi := \phi, \psi)}{\mathit{body}(t_S(\phi).m(\psi)) = (x : t_S(\phi), \overline{x : \tau}).e[\theta]}$$

$$\frac{(\textbf{type}\ \overline{\alpha\ \tau_I}) = \Phi}{\hat{\Phi} = \overline{\alpha}} \qquad \mathit{type}(\tau_S\{\overline{v}\}) = \tau_S \qquad \frac{mM_1, mM_2 \in \overline{S}\ \text{implies}\ M_1 = M_2}{\mathit{unique}(\overline{S})}$$

$$\mathit{tdecls}(\overline{D}) = [t \mid (\textbf{type}\ t(\Phi)\ T) \in \overline{D}]$$

$$\mathit{mdecls}(\overline{D}) = [t_S.m \mid (\textbf{func}\ (x\ t_S(\Phi))\ mM\ \{\textbf{return}\ e\}) \in \overline{D}]$$

$$\frac{(\alpha : \tau_I) \in \Delta}{\mathit{bounds}_\Delta(\alpha) = \tau_I} \qquad \mathit{bounds}_\Delta(\tau_S) = \tau_S \qquad \mathit{bounds}_\Delta(\tau_I) = \tau_I$$

$$\mathit{methods}_\Delta(t_S(\phi)) = \{(mM)[\eta] \mid (\textbf{func}\ (x\ t_S(\Phi))\ mM\ \{\textbf{return}\ e\}) \in \overline{D},\ \eta = (\Phi :=_\Delta \phi)\}$$

$$\frac{\textbf{type}\ t_I(\Phi)\ \textbf{interface}\ \{\overline{S}\} \in \overline{D} \qquad \eta = (\Phi := \phi)}{\mathit{methods}_\Delta(t_I(\phi)) = \overline{S}[\eta]} \qquad \frac{(\alpha : \tau_I) \in \Delta}{\mathit{methods}_\Delta(\alpha) = \mathit{methods}_\Delta(\tau_I)}$$

Fig. 16. FGG auxiliary functions

to before but now under type environment Δ. Judgement $\Phi \vdash T\ ok$ holds if under type formals Φ type literal T is well formed, and again is a straightforward adjustment of its earlier definition.

Judgement $D\ ok$ holds if declaration D is well formed. The definitions are similar to previous definition. For a type declaration, its type formals must be well formed under the empty type environment, and its type literal must be well formed under the environment given by the type formals. For a method declaration, we require $\Phi; \Psi\ ok\ \Delta$, where Φ are the type formals of the receiver and Ψ are the type formals of the method. The receiver type must be declared with formals Φ', where $\Phi <: \Phi'$. An alternative, simpler design would require Φ and Φ' to be identical; but that would rule out the solution to the expression problem given in Section 2.3. The rest is a straightforward adjustment of its earlier definition.

Judgement $\Delta; \Gamma \vdash e : \tau$ holds if under type environment Δ and environment Γ expression e has type τ. The adjustments are straightforward. Method calls are adjusted so that the type of the arguments and result are instantiated by the method type arguments. Type assertions are adjusted to take into account that type names are replaced by types. In method calls and type assertions, type parameters are treated as equivalent to the parameters' bound.

4.4 FGG Reduction

Figure 18 presents the FGG reduction rules.

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

$$\textsc{<:-param}\ \frac{}{\Delta \vdash \alpha <: \alpha} \qquad \textsc{<:}_S\ \frac{}{\Delta \vdash \tau_S <: \tau_S} \qquad \textsc{<:}_I\ \frac{methods_\Delta(\tau) \supseteq methods_\Delta(\tau_I)}{\Delta \vdash \tau <: \tau_I} \qquad \textsc{<:-formals}\ \frac{\emptyset \vdash \overline{\tau_I <: \sigma_I}}{(\textbf{type}\ \overline{\alpha\ \tau_I}) <: (\textbf{type}\ \overline{\alpha\ \sigma_I})}$$

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

$$\textsc{t-param}\ \frac{(\alpha : \tau_I) \in \Delta}{\Delta \vdash \alpha\ ok} \qquad \textsc{t-named}\ \frac{\Delta \vdash \phi\ ok \qquad (\textbf{type}\ t(\Phi)\ T) \in \overline{D} \qquad \eta = (\Phi :=_\Delta \phi)}{\Delta \vdash t(\phi)\ ok} \qquad \textsc{t-actual}\ \frac{\overline{\tau} = \phi \qquad \Delta \vdash \overline{\tau\ ok}}{\Delta \vdash \phi\ ok}$$

Well-formed type formals and nested formals $\boxed{\Phi \vdash \Psi\ ok}$ $\boxed{\Phi;\ \Psi\ ok\ \Delta}$

$$\textsc{t-formal}\ \frac{(\textbf{type}\ \overline{\alpha\ \tau_I}) = \Psi \qquad distinct(\hat{\Phi}, \overline{\alpha}) \qquad \Phi, \Psi \vdash \overline{\tau_I\ ok}}{\Phi \vdash \Psi\ ok} \qquad \textsc{t-nested}\ \frac{\emptyset \vdash \Phi\ ok \qquad \Phi \vdash \Psi\ ok \qquad \Delta = \Phi, \Psi}{\Phi;\ \Psi\ ok\ \Delta}$$

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

$$\textsc{t-specification}\ \frac{distinct(\overline{x}) \qquad \Phi;\ \Psi\ ok\ \Delta \qquad \Delta \vdash \overline{\tau\ ok} \qquad \Delta \vdash \tau\ ok}{\Phi \vdash m(\Psi)(\overline{x\ \tau})\ \tau\ ok} \qquad \textsc{t-struct}\ \frac{distinct(\overline{f}) \qquad \Phi \vdash \overline{\tau\ ok}}{\Phi \vdash \textbf{struct}\ \{\overline{f\ \tau}\}\ ok} \qquad \textsc{t-interface}\ \frac{unique(\overline{S}) \qquad \Phi \vdash \overline{S\ ok}}{\Phi \vdash \textbf{interface}\ \{\overline{S}\}}$$

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

$$\textsc{t-type}\ \frac{\emptyset \vdash \Phi\ ok \qquad \Phi \vdash T\ ok}{\textbf{type}\ t(\Phi)\ T\ ok} \qquad \textsc{t-func}\ \frac{distinct(x, \overline{x}) \qquad (\textbf{type}\ t_S(\Phi')\ T) \in \overline{D} \qquad \Phi <: \Phi' \qquad \Phi;\ \Psi\ ok\ \Delta \qquad \Delta \vdash \overline{\tau\ ok} \qquad \Delta \vdash \sigma\ ok \qquad \Delta;\ x : t_S(\hat{\Phi}),\ \overline{x : \tau} \vdash e : \tau \qquad \Delta \vdash \tau <: \sigma}{\textbf{func}\ (x\ t_S(\Phi))\ m(\Psi)(\overline{x\ \tau})\ \sigma\ \{\textbf{return}\ e\}\ ok}$$

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

$$\textsc{t-var}\ \frac{(x : \tau) \in \Gamma}{\Delta;\ \Gamma \vdash x : \tau} \qquad \textsc{t-call}\ \frac{\Delta;\ \Gamma \vdash e : \tau \qquad \Delta;\ \Gamma \vdash \overline{e : \tau} \qquad (m(\Psi)(\overline{x\ \sigma})\ \sigma) \in methods_\Delta(\tau) \qquad \eta = (\Psi :=_\Delta \psi) \qquad \Delta \vdash (\overline{\tau <: \sigma})[\eta]}{\Delta;\ \Gamma \vdash e.m(\psi)(\overline{e}) : \sigma[\eta]}$$

$$\textsc{t-literal}\ \frac{\Delta \vdash \tau_S\ ok \qquad \Delta;\ \Gamma \vdash \overline{e : \tau} \qquad (\overline{f\ \sigma}) = fields(\tau_S) \qquad \Delta \vdash \overline{\tau <: \sigma}}{\Delta;\ \Gamma \vdash \tau_S\{\overline{e}\} : \tau_S} \qquad \textsc{t-field}\ \frac{\Delta;\ \Gamma \vdash e : \tau_S \qquad (\overline{f\ \tau}) = fields(\tau_S)}{\Delta;\ \Gamma \vdash e.f_i : \tau_i}$$

$$\textsc{t-assert}_I\ \frac{\Delta \vdash \tau_J\ ok \qquad \Delta;\ \Gamma \vdash e : \sigma_J}{\Delta;\ \Gamma \vdash e.(\tau_J) : \tau_J} \qquad \textsc{t-assert}_S\ \frac{\Delta \vdash \tau_S\ ok \qquad \Delta;\ \Gamma \vdash e : \sigma_J \qquad \tau_S <: bounds_\Delta(\sigma_J)}{\Delta;\ \Gamma \vdash e.(\tau_S) : \tau_S} \qquad \boxed{\textsc{t-stupid}\ \frac{\Delta \vdash \tau\ ok \qquad \Delta;\ \Gamma \vdash e : \sigma_S}{\Delta;\ \Gamma \vdash e.(\tau) : \tau}}$$

Programs $\boxed{P\ ok}$

$$\textsc{t-prog}\ \frac{distinct(tdecls(\overline{D})) \qquad distinct(mdecls(\overline{D})) \qquad \overline{D\ ok} \qquad \emptyset;\ \emptyset \vdash e : \tau}{\textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{\_ = e\}\ ok}$$

Fig. 17. FGG typing

	Value	$v ::= \tau_S\{\overline{v}\}$	
Evaluation context	$E ::=$		
Hole	$\Box$	Structure	$\tau_S\{\overline{v}, E, \overline{e}\}$
Method call receiver	$E.m(\overline{\tau})(\overline{e})$	Select	$E.f$
Method call arguments	$v.m(\overline{\tau})(\overline{v}, E, \overline{e})$	Type assertion	$E.(\tau)$

Reduction $\boxed{d \longrightarrow e}$

$$\textsc{r-field}\ \frac{(\overline{f\ \tau}) = fields(\tau_S)}{\tau_S\{\overline{v}\}.f_i \longrightarrow v_i} \qquad \textsc{r-call}\ \frac{(x : \tau_S, \overline{x : \tau}).e = body(type(v).m(\psi))}{v.m(\psi)(\overline{v}) \longrightarrow e[x := v, \overline{x := v}]} \qquad \textsc{r-assert}\ \frac{\emptyset \vdash type(v) <: \tau}{v.(\tau) \longrightarrow v} \qquad \textsc{r-context}\ \frac{d \longrightarrow e}{E[d] \longrightarrow E[e]}$$

Fig. 18. FGG reduction

The adjustment to values, the auxiliary function *type*, and to evaluation contexts are simple, replacing type names by types and adding type arguments as appropriate. Judgement $d \longrightarrow e$ holds if expression d steps to expression e. Again, the adjustments are all simple.

4.5 FGG Properties

The results of the previous section adapt straightforwardly.

Lemma 4.1 (Well formed). *If* $\Delta; \Gamma \vdash e : \tau$ *then* $\Delta \vdash \tau$ *ok.*

The substitution lemma is adapted to take into account that in a method declaration the type parameters of the receiver are substituted before the type parameters of the method.

Lemma 4.2 (Substitution). *Let* $\eta = (\overline{\alpha := \tau})$ *be a substitution.*

- *If* $\emptyset \vdash \overline{\alpha <: \tau_I}[\eta]$ *and* $\overline{\alpha : \tau_I}, \Delta \vdash \tau$ *ok then* $\Delta[\eta] \vdash \tau[\eta]$ *ok.*
- *If* $\emptyset \vdash \overline{\alpha <: \tau_I}[\eta]$ *and* $\overline{\alpha : \tau_I}, \Delta; \Gamma \vdash e : \tau$ *then* $\Delta[\eta]; \Gamma[\eta] \vdash e[\eta] : \tau[\eta]$.
- *If* $\emptyset; \emptyset \vdash \overline{v : \tau}$ *and* $\emptyset; \overline{x : \sigma} \vdash e : \sigma$ *and* $\emptyset \vdash \overline{\tau <: \sigma}$ *then* $\emptyset; \emptyset \vdash e[\overline{x := v}] : \tau$ *for some type* τ *with* $\emptyset \vdash \tau <: \sigma$.

The remaining results are easy to adjust.

Theorem 4.3 (Preservation). *If* $\emptyset; \emptyset \vdash d : \sigma$ *and* $d \longrightarrow e$ *then* $\emptyset; \emptyset \vdash e : \tau$ *for some* τ *with* $\emptyset \vdash \tau <: \sigma$.

Expression d *panics* if there exist evaluation context E, value v, and type τ such that $d = E[v.(\tau)]$ and $\emptyset \vdash type(v) \not<: \tau$.

Theorem 4.4 (Progress). *If* $\emptyset; \emptyset \vdash d : \sigma$ *then either* d *is a value,* $d \longrightarrow e$ *for some* e*, or* d *panics.*

5 MONOMORPHISATION

The monomorphisation process consists of two phases. In the first phase, a set of types and method instantiations are collected from an FGG program. In the second phase, an FGG program is translated to its FG equivalent following the instance set computed in the first phase.

Throughout this section, we illustrate the monomorphisation process with the FGG program in Figure 19 (left). This program contains a `Dispatcher` structure which processes abstract `Events`. A `Dispatcher` processes events, and events are objects that can be processed. For the sake of space, the program in Figure 19 includes only one implementation of `Events`, i.e., `UIEvents`, but other implementations may be easily added following the same pattern.

```
type Any interface {}
type Int struct {}

type Event interface {
  Process(type b Any)(y b) Int
}
type UIEvent struct {}
func (x UIEvent) Process(type b Any)(y b) Int {
  return Int{}
}

type Dispatcher struct {}
func (x Dispatcher) Dispatch(y Event) Int {
  return y.Process(Int)(Int{})
}

func main() {
  _ = Dispatcher{}.Dispatch(UIEvent{})
}
```

```
type Top struct {}
type Int struct {}
type Event interface {
  Process<Int>(y Int) Int
  Process<1>() Top
}
type UIEvent struct {}
func (x UIEvent) Process<Int>(y Int) Int {
  return Int{} }
func (x UIEvent) Process<1>() Top {
  return Top{} }
type Dispatcher struct {}
func (x Dispatcher) Dispatch(y Event) Int {
  return y.Process<Int>(Int{}) }
func (x Dispatcher) Dispatch<1>() Top {
  return Top{} }
func main() {
_ = Dispatcher{}.Dispatch(UIEvent{})
}
```

Fig. 19. Dispatcher example: FGG source (left) and FG translation (right)

5.1 Collecting Type and Method Instances

Let ω, Ω range over instance sets, which contain elements of type τ or pairs of a type with a method and its type arguments, $\tau.m(\psi)$. In Figure 20 we define a judgement $P \blacktriangleright \Omega$ which computes the set of instances of types and methods required to correctly monomorphise an FGG program.

Judgement $\Delta; \Gamma \vdash e \blacktriangleright \omega$ holds if ω is the instance set for expression e, given environments Δ and Γ. In the rules for variables, structure literals, field selections and type assertions, we simply collect the occurrences of type instances and proceed inductively. The rule for method calls additionally records the instantiation of the method $\tau.m(\psi)$ where τ is the type of its receiver. In the rules for structure literals and method calls, we assume that sequences of instance sets, e.g., $\overline{\omega}$, coerce to a set consisting of the union of the elements of the sequence.

The instance set of a program P is the limit of function G applied to the instance set of its body. $G_\Delta(\omega)$ is defined via four auxiliary functions that compute the type and method instances required by ω. It returns all type and method instantiations that are required to monomorphise declarations (*F-closure*, *M-closure*) and to preserve the <: relation (*I-closure*, *S-closure*).

F-closure finds all the type instances that occur in the declarations of structures, while *M-closure* finds all the type instances that occur in the declarations of method instances.

I-closure finds all the method signature instances that are required to preserve the <: relation over interfaces. *S-closure* finds all the type and method sets required by method calls, inter-procedurally. For each method instance $\tau.m(\psi)$ in ω it finds all instance sets of all implementations of method m, following the <: relation. Intuitively, *I-closure* and *S-closure* are used to guarantee that if $\tau <: \sigma$, then the monomorphised version of τ also implements the monomorphised version of σ.

Consider the example in Figure 19 (left). For the top-level method, we have

$$\emptyset; \emptyset \vdash \texttt{Dispatcher\{\}.Dispatch(UIEvent\{\})} \blacktriangleright \{\texttt{Dispatcher}, \texttt{Dispatcher.Dispatch()}\}$$

Posing $\omega_0 = \{\texttt{Dispatcher}, \texttt{Dispatcher.Dispatch()}\}$, we compute the limit of G applied to ω_0. We have $G_\emptyset(\omega_0) = \omega_0 \cup \{\texttt{Event}, \texttt{Int}, \texttt{Dispatcher.Dispatch()}, \texttt{Event}, \texttt{Event.Process(Int)}\}$ where Event and Int

Instance sets $\boxed{\omega, \Omega}$

ω, Ω range over sets containing elements of the form τ or $\tau.m(\psi)$.

Expressions and programs $\boxed{\Delta; \Gamma \vdash e \blacktriangleright \omega}$ $\boxed{P \blacktriangleright \Omega}$

$$\textsc{I-var}\ \frac{}{\Delta; \Gamma \vdash x \blacktriangleright \emptyset} \qquad \textsc{I-literal}\ \frac{\Delta; \Gamma \vdash \overline{e \blacktriangleright \omega}}{\Delta; \Gamma \vdash \tau_S\{\overline{e}\} \blacktriangleright \{\tau_S\} \cup \overline{\omega}} \qquad \textsc{I-field}\ \frac{\Delta; \Gamma \vdash e \blacktriangleright \omega}{\Delta; \Gamma \vdash e.f_i \blacktriangleright \omega} \qquad \textsc{I-assert}\ \frac{\Delta; \Gamma \vdash e \blacktriangleright \omega}{\Delta; \Gamma \vdash e.(\tau) \blacktriangleright \{\tau\} \cup \omega}$$

$$\textsc{I-call}\ \frac{\Delta; \Gamma \vdash e : \tau \quad \Delta; \Gamma \vdash e \blacktriangleright \omega \quad \Delta; \Gamma \vdash \overline{e \blacktriangleright \omega}}{\Delta; \Gamma \vdash e.m(\psi)(\overline{e}) \blacktriangleright \{\tau, \tau.m(\psi)\} \cup \omega \cup \overline{\omega}} \qquad \textsc{I-prog}\ \frac{\emptyset; \emptyset \vdash e \blacktriangleright \omega \qquad \Omega = \lim_{n\to\infty} G^n_\emptyset(\omega)}{\textbf{package main}; \overline{D}\ \textbf{func main}()\ \{\_ = e\} \blacktriangleright \Omega}$$

Auxiliary functions

$$G_\Delta(\omega) = \omega \cup \mathit{F\text{-}closure}(\omega) \cup \mathit{M\text{-}closure}_\Delta(\omega) \cup \mathit{I\text{-}closure}_\Delta(\omega) \cup \mathit{S\text{-}closure}_\Delta(\omega)$$

$$\mathit{F\text{-}closure}(\omega) = \bigcup \left\{ \overline{\tau} \;\middle|\; \tau_S \in \omega,\ (\overline{f\ \tau}) = \mathit{fields}(\tau_S) \right\}$$

$$\mathit{M\text{-}closure}_\Delta(\omega) = \bigcup \left\{ \overline{\sigma}[\eta] \cup \{\sigma[\eta]\} \;\middle|\; \tau.m(\psi) \in \omega,\ (m(\Psi)(\overline{x\ \sigma})\ \sigma) \in \mathit{methods}_\Delta(\tau),\ \eta = (\Psi := \psi) \right\}$$

$$\mathit{I\text{-}closure}_\Delta(\omega) = \left\{ \tau'_I.m(\psi) \;\middle|\; \tau_I.m(\psi) \in \omega,\ \tau'_I \in \omega,\ \Delta \vdash \tau'_I <: \tau_I \right\}$$

$$\mathit{S\text{-}closure}_\Delta(\omega) = \bigcup \left\{ \{\tau_S.m(\psi)\} \cup \Omega \;\middle|\; \begin{array}{l} \tau.m(\psi) \in \omega,\ \tau_S \in \omega,\ \Delta \vdash \tau_S <: \tau,\ \Delta; x : \tau_S, \overline{x : \sigma} \vdash e \blacktriangleright \Omega \\ (x : \tau_S, \overline{x : \sigma}).e = \mathit{body}(\tau_S.m(\psi)) \end{array} \right\}$$

Fig. 20. Computing instance sets

are obtained from $\mathit{M\text{-}closure}_\emptyset(\omega_0)$, while `Dispatcher.Dispatch()`, `Event`, and `Event.Process(Int)` are obtained from $\mathit{S\text{-}closure}_\emptyset(\omega_0)$.

The limit of function G is reached after two iterations, i.e., $\lim_{n\to\infty} G^n_\emptyset(\omega_0) = G_\emptyset(G_\emptyset(\omega_0)) = G_\emptyset(\omega_0) \cup \{\texttt{UIEvent.Process(Int)}\}$. Note that `UIEvent.Process(Int)` is obtained via *S-closure* using the fact that `UIEvent <: Event` holds.

5.2 Monomorphisation Judgement

We now define a judgement $\vdash P \mapsto P^\dagger$ where P is a program in FGG, and $P^\dagger$ is a corresponding monomorphised program in FG. This judgement is in turn defined by judgements for each of the syntactic categories in FGG. Some of the judgements are also parameterised by instance sets (ranged over by Ω), substitutions that map type parameters to ground types (ranged over by η), or method instance sets (ranged over by μ).

Figure 21 formalises how we recursively apply a consistent renaming to generate FG code. To monomorphise types, type formals, method names, and method formals, given a substitution η, we assume a map from closed types to identifiers. For instance, if `f` is a type with two arguments, and `g` and `h` are types with no arguments, then closed type `f(g(),h())` might correspond to the identifier "`f<g<>,h<>>`" assuming "`<,>`" are allowed as letters in identifiers. We write $t^\dagger = \langle\tau\rangle$ to compute the identifier $t^\dagger$ that corresponds to closed type τ. Similarly, we write $m^\dagger = \langle m(\psi)\rangle$ to compute the identifier $m^\dagger$ that corresponds to closed method instantiation $m(\psi)$.

Types and methods $\boxed{\eta \vdash \tau \mapsto t^\dagger}$ $\boxed{\eta \vdash t(\Phi) \mapsto t^\dagger}$ $\boxed{\eta \vdash m(\psi) \mapsto m^\dagger}$ $\boxed{\eta \vdash m(\Psi) \mapsto m^\dagger}$

$$\textsc{m-type}\ \frac{t^\dagger = \langle \tau[\eta] \rangle}{\eta \vdash \tau \mapsto t^\dagger} \qquad \textsc{m-tformal}\ \frac{t^\dagger = \langle t(\overline{\alpha}[\eta]) \rangle}{\eta \vdash t(\mathbf{type}\ \overline{\alpha\ \tau_I}) \mapsto t^\dagger} \qquad \textsc{m-method}\ \frac{m^\dagger = \langle m(\psi[\eta]) \rangle}{\eta \vdash m(\psi) \mapsto m^\dagger} \qquad \textsc{m-mformal}\ \frac{m^\dagger = \langle m(\overline{\alpha}[\eta]) \rangle}{\eta \vdash m(\mathbf{type}\ \overline{\alpha\ \tau_I}) \mapsto m^\dagger}$$

Expression $\boxed{\eta \vdash e \mapsto e^\dagger}$

$$\textsc{m-var}\ \frac{}{\eta \vdash x \mapsto x} \qquad \textsc{m-value}\ \frac{\eta \vdash \tau_S \mapsto t_S^\dagger \qquad \eta \vdash \overline{e \mapsto e^\dagger}}{\eta \vdash \tau_S\{\overline{e}\} \mapsto t_S^\dagger\{\overline{e^\dagger}\}} \qquad \textsc{m-select}\ \frac{\eta \vdash e \mapsto e^\dagger}{\eta \vdash e.f \mapsto e^\dagger.f}$$

$$\textsc{m-call}\ \frac{\eta \vdash e \mapsto e^\dagger \qquad \eta \vdash m(\psi) \mapsto m^\dagger \qquad \eta \vdash \overline{e \mapsto e^\dagger}}{\eta \vdash e.m(\psi)(\overline{e}) \mapsto e^\dagger.m^\dagger(\overline{e^\dagger})} \qquad \textsc{m-assert}\ \frac{\eta \vdash e \mapsto e^\dagger \qquad \eta \vdash \tau \mapsto t^\dagger}{\eta \vdash e.(\tau) \mapsto e^\dagger.(t^\dagger)}$$

Method signature $\boxed{\eta \vdash M \mapsto M^\dagger}$ $\boxed{\eta \vdash S \mapsto S^\dagger}$

$$\textsc{m-sig}\ \frac{\eta \vdash \overline{\tau \mapsto t^\dagger} \qquad \eta \vdash \tau \mapsto u^\dagger}{\eta \vdash (\overline{x\ \tau})\ \tau \mapsto (\overline{x\ t^\dagger})\ u^\dagger} \qquad \textsc{m-id}\ \frac{hash(mM[\eta]) = m^*}{\eta \vdash mM \mapsto m^*()\ \mathsf{Top}}$$

Type literal $\boxed{\eta;\ \mu \vdash T \mapsto T^\dagger}$

$$\textsc{m-struct}\ \frac{\eta \vdash \overline{\tau \mapsto t^\dagger}}{\eta;\ \mu \vdash \mathbf{struct}\{\overline{f\ \tau}\} \mapsto \mathbf{struct}\{\overline{f\ t^\dagger}\}} \qquad \textsc{m-interface}\ \frac{\eta;\ \mu \vdash \overline{S \mapsto \mathcal{S}}}{\eta;\ \mu \vdash \mathbf{interface}\{\overline{S}\} \mapsto \mathbf{interface}\{\bigcup \overline{\mathcal{S}}\}}$$

Fig. 21. Monomorphisation of FGG into FG — name mapping

To monomorphise an expression given a substitution η, we recursively monomorphise all the types and expressions contained within this expression. We proceed similarly to monomorphise method signatures in Rule M-SIG.

Rule M-ID is used to generate a dummy method signature that represents uniquely the alpha-equivalence class of its FGG counterpart. The signature specifies no parameters and the return type Top. It is necessary to generate such methods to ensure that if a type does not implement another in an FGG program, then this is also the case in its monomorphised counterpart. We assume that $hash(mM_1) = hash(mM_2)$ for all $M_1 = M_2$, using the same notion of equality as in *unique* and <:.

To monomorphise a structure given a substitution, we recursively monomorphise all the types contained within its field declarations. To monomorphise an interface, we recursively monomorphise each of its signatures and flatten the result in a single sequence of declarations.

Figure 22 formalises how declarations are generated from Ω. Here we let N range over $(\overline{x\ \tau})\ \tau$.

To monomorphise an interface specification we pass *two* environments. One is a substitution from type parameters to ground types η and the other is a set of method instances μ, i.e., a set of entries of type $m(\psi)$. For each entry in μ, we compute a new substitution θ which extends η and is used to generate a monomorphised instance of the corresponding signature. In addition, we generate dummy signature $S^\dagger$ that uniquely identifies the FGG signature. Each parameterised method may produce zero or more monomorphised instances, plus a dummy method signature.

Interface specification $\boxed{\eta;\ \mu \vdash S \mapsto \mathcal{S}}$

M-SPEC

$$\frac{\mathcal{S} = \left\{ m^\dagger N^\dagger \;\middle|\; m(\psi) \in \mu,\ \theta = (\eta, \Psi := \psi),\ \theta \vdash m(\Psi) \mapsto m^\dagger,\ \theta \vdash N \mapsto N^\dagger \right\} \qquad \eta \vdash m(\Psi)N \mapsto S^\dagger}{\eta;\ \mu \vdash m(\Psi)N \mapsto \mathcal{S} \ \cup\ \{S^\dagger\}}$$

Declaration $\boxed{\Omega \vdash D \mapsto \mathcal{D}}$

M-TYPE

$$\frac{\mathcal{D} = \left\{ \textbf{type}\ t^\dagger\ T^\dagger \;\middle|\; t(\phi) \in \Omega,\ \eta = (\Phi := \phi),\ \mu = \{m(\psi) \mid t(\phi).m(\psi) \in \Omega\},\ \eta;\ \mu \vdash T \mapsto T^\dagger \right\}}{\Omega \vdash \textbf{type}\ t(\Phi)\ T \mapsto \mathcal{D}}$$

M-FUNC

$$\frac{\begin{array}{l}\mathcal{D} = \left\{ \textbf{func}\ (x\ t_S^\dagger)\ m^\dagger N^\dagger\ \{\textbf{return}\ e^\dagger\} \;\middle|\; \begin{array}{l} t_S(\phi).m(\psi) \in \Omega,\ \theta = (\Phi := \phi, \Psi := \psi),\ \theta \vdash t_S(\Phi) \mapsto t_S^\dagger \\ \theta \vdash m(\Psi) \mapsto m^\dagger,\ \theta \vdash N \mapsto N^\dagger,\ \theta \vdash e \mapsto e^\dagger \end{array} \right\} \\ \mathcal{D}' = \left\{ \textbf{func}\ (x\ t_S^\dagger)\ S^\dagger \{\textbf{return}\ \mathsf{Top}\{\}\} \;\middle|\; t_S(\phi) \in \Omega,\ \eta = (\Phi := \phi),\ \eta \vdash t_S(\Phi) \mapsto t_S^\dagger,\ \eta \vdash m(\Psi)N \mapsto S^\dagger \right\}\end{array}}{\Omega \vdash \textbf{func}\ (x\ t_S(\Phi))\ m(\Psi)N\ \{\textbf{return}\ e\} \mapsto \mathcal{D} \cup \mathcal{D}'}$$

Program $\boxed{\vdash P \mapsto P^\dagger}$

M-PROGRAM

$$\frac{\textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{\_ = e\} \blacktriangleright \Omega \qquad \Omega \vdash \overline{D \mapsto \mathcal{D}} \qquad \overline{D^\dagger} = \{\textbf{type}\ \mathsf{Top}\ \textbf{struct}\ \{\}\} \cup \bigcup \overline{\mathcal{D}} \qquad \emptyset \vdash e \mapsto e^\dagger}{\vdash \textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{\_ = e\} \mapsto \textbf{package main};\ \overline{D^\dagger}\ \textbf{func main}()\ \{\_ = e^\dagger\}}$$

Fig. 22. Monomorphisation of FGG into FG — instance generation, where N ranges over $(\overline{x\ \tau})\ \tau$

To monomorphise the declaration of a type t given an instance set, for each instance of t, we generate a substitution η and a method instance set μ. Then we recursively produce a monomorphised declaration for each generated pair of η and μ. Note that each type declaration may produce zero or more monomorphised declarations.

To monomorphise a method declaration given an instance set, we compute a substitution θ for each method instance in Ω. Then we produce a monomorphised version of a method for each of its instantiations. Note that each method declaration may produce zero or more monomorphised declarations. In addition, for each type instance and each of its methods, we generate a dummy method that returns an instance of Top.

To monomorphise a program P, we compute its instance set Ω then monomorphise its declaration and body given with respect to Ω. We additionally add the declaration of the empty structure Top.

The FGG program in Figure 19 (left) is translated to the FG program on the righthand side of the figure. The translation starts with rule M-PROGRAM where Ω = {`Int`, `Event`, `Event.Process(Int)`, `UIEvent`, `UIEvent.Process(Int)`, `Dispatcher`, `Dispatcher.Dispatch()`}. Rule M-TYPE is used to generate the instances of types `Int`, `Event`, `UIEvent`, and `Dispatcher`. Rule M-FUNC is used to generate the methods `UIEvent.Process(Int)` and `Dispatcher.Dispatch()`, as well as their dummy counterparts `Process<1>()` and `Dispatch<1>()`. Rule M-SPEC is used to generate the method signatures of interface `Event`, i.e., `Event.Process(Int)` and its dummy counterpart `Process<1>()`.

Not monomorphisable $\boxed{P \textit{ nomono}}$ $\boxed{D \textit{ nomono}}$

$$\textsc{mn-program}\ \frac{D_i \textit{ nomono}}{\textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{\_ = e\}\ \textit{nomono}}$$

$$\textsc{mn-func}\ \frac{\Delta = \Phi, \Psi \qquad \text{exists } n \in \mathbb{N} \text{ and } (t_S(\phi).m(\psi)) \in G^n_\Delta(\{t_S(\hat{\Phi}), t_S(\hat{\Phi}).m(\hat{\Psi})\}) \text{ s.t. } \Phi \prec \phi \text{ or } \Psi \prec \psi}{\textbf{func}\ (x\ t_S(\Phi))\ m(\Psi)(\overline{y\ \tau})\ \tau\ \{\textbf{return}\ e\}\ \textit{nomono}}$$

Occurs check $\boxed{\Phi \prec \phi}$ $\boxed{\alpha \prec \tau}$

$$\frac{\phi = \overline{\tau} \qquad \alpha_i \prec \tau_i}{(\textbf{type}\ \overline{\alpha\ \tau_I}) \prec \phi} \qquad \frac{\tau \neq \alpha \qquad \alpha \in fv(\tau)}{\alpha \prec \tau}$$

Fig. 23. Monomorphisability check

5.3 Ruling Out Non-Monomorphisable Programs

It is not possible to monomorphise all FGG programs since programs that contain polymorphic recursive methods may produce infinitely many type instances. To address this issue, we propose a predicate *P nomono* which conservatively identifies programs that can produce infinitely many type instances. Programs for which this predicate does *not* hold are guaranteed to be monomorphisable. Note that there exist programs which produce finitely many type instances but for which the predicate does hold, e.g., programs containing a polymorphic recursive method that is never called.

The predicate is formally defined in Figure 23, notably reusing our instance generation procedures. *P nomono* holds if *D nomono* holds for at least one of its method declarations. *D nomono* holds if it is possible to find an element in its instance set, inductively constructed using function *S-closure*, such that the occurs check is satisfied. The occurs check holds when a type variable appears under a type constructor in a type instance or method call (in the same position it occupies in the type or method formal). We write $fv(\tau)$ for the set of type parameters occurring in τ.

5.4 Monomorphisation Properties

Not all programs are monomorphisable, however we can decide whether a program is monomorphisable with the *P nomono* predicate.

Theorem 5.1 (Decidability). *If P ok then it is decidable whether or not P nomono holds.*

For all well-typed programs P such that *P nomono* does not hold, their instance sets are finite.

Theorem 5.2 (Monomorphisable). *If P ok and P nomono doesn't hold then* $P \blacktriangleright \Omega$ *with* Ω *finite.*

Monomorphisation preserves typing, i.e., the translation of a well-typed FGG program is a well-typed FG program.

Theorem 5.3 (Sound). *If P ok and* $\vdash P \mapsto P^\dagger$ *then* $P^\dagger$ *ok.*

The reduction behaviour of well-typed FGG programs is preserved and reflected by their monomorphised counterpart (see Figure 24). Write $\vdash d \mapsto d^\dagger$ to abbreviate $\emptyset; \emptyset \vdash d \mapsto d^\dagger$.

Theorem 5.4 (Bisimulation). *Let P ok and* $\vdash P \mapsto P^\dagger$ *with* $P = \overline{D} \triangleright d$ *and* $P^\dagger = \overline{D^\dagger} \triangleright d^\dagger$. *Then:*
(a) if $d \longrightarrow e$ *then there exists* $e^\dagger$ *such that* $d^\dagger \longrightarrow e^\dagger$ *and* $\vdash e \mapsto e^\dagger$;
(b) if $d^\dagger \longrightarrow e^\dagger$ *then there exists e such that* $d \longrightarrow e$ *and* $\vdash e \mapsto e^\dagger$.

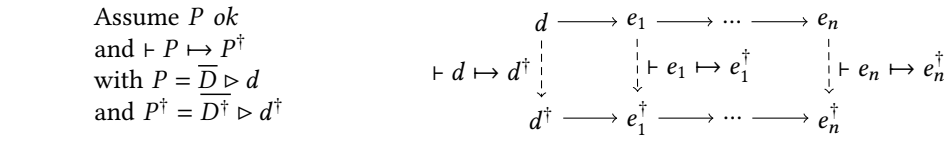

Fig. 24. Bisimulation

6 IMPLEMENTATION

We have made available a prototype implementation,[1] which contains FG and FGG type checkers and interpreters, and a monomorphiser from FGG to FG (including the *nomono* check). We wrote the implementation in Go to facilitate interactions with the Go designers and community. Our interface includes some extensions to our tiny syntax for FG and FGG, such as direct support for interface embedding, some primitive data types, and minimal I/O. Versions of all the presented examples have been tested using the implementation.

We took advantage of the implementation to perform extensive testing. FG evaluation results are compared to those using the official Go compiler, and the FG and FGG interpreters support dynamic checking of preservation and progress. To test monomorphisation, we added the test of bisimulation depicted in Figure 24: given a well-typed FGG program we generate its FG monomorphisation; we step the FGG and FG terms and confirm that the new FGG term monomorphises to the new FG term; and repeat until termination.

Besides handcrafted examples and stress tests, we used NEAT [Claessen et al. 2015; Duregård 2016] to lazily enumerate all well-typed programs (up to some size relative to the total number of occurrences of method and type symbols) from a subset of FGG (similar to SmallCheck [Runciman et al. 2008]). The subset consists of programs which have: (1) at least one method and one field; (2) at most one empty interface; and (3) at most two empty structs. And where: (1) each method has at most two arguments; (2) each struct has at most two fields; (3) each interface has at most two members and two parents; and (4) each method or type constructor has at most two type parameters. Moreover, we disallow mutually recursive type definitions. These measures are taken to truncate the space of possible programs. We generate all FGG programs in our subset up to size 20, and confirm they pass the bisimulation test described above.

7 RELATED WORK

This paper is the first to present a core formalism for the Go language. Our presentation is styled after that of Featherweight Java by Igarashi et al. [2001]. Like them, we focus on a tiny, functional subset of the language; we define versions with and without generics; and we consider translation from one to the other. We also mark as "stupid" casts/type assertions that are disallowed in source but are required for reduction to preserve types.

Our work resembles the development of generics for Java by Bracha et al. [1998] and for .NET by [Kennedy and Syme 2001] in that we build on a well-established base language. We note that in Featherweight Go, Featherweight Generic Go (and in the Go language), since method signatures are nonvariant, there are no fundamental decidability issues related to F-bounded polymorphism and variance [Pierce 1992], and so there is no need to consider more sophisticated techniques such as those of Greenman et al. [2014] to ensure decidability of type checking.

In terms of formalisations of generics, prior work on generics adopts one, or a combination, of three main approaches, *erasure*, *runtime representation of types as values*, and *monomorphisation*.

[1] https://github.com/rhu1/fgg/

Erasure. Bracha et al. [1998] present a translation from Java with generics to Java without generics that erases all information about type parameters. The translation relies on *bridge* methods, which in turn rely on method overloading, which is not supported in Go. Igarashi et al. [1999] formalised the translation for the FJ subset of Java (avoiding bridge methods) and proved it preserves typing and reductions. Downsides of erasure are that casts to generic types must be restricted and creation of generic arrays becomes tricky (see [Naftalin and Wadler 2006]). Moreover, erased code is often less efficient than monomorphised code. An upside is that erasure is linear in the size of the source, whereas monomorphisation can lead to an explosion in code size.

Runtime representation. In contrast to Java erasure, Kennedy and Syme [2001] developed an extension of the .NET Common Language Runtime (CLR) and C♯ with direct support for generics. They generate a mixture of specialised and shared code: the former is compiled separately for each primitive type and is similar to monomorphisation; the latter is compiled once for every object type and is similar to erasure. JIT compilation is exploited to perform specialisation *dynamically*, avoiding potential code bloat. Code sharing is implemented by storing runtime *type-reps* [Crary et al. 1998] for type parameters.

The overhead of runtime assembly of type-reps can be optimised by pre-computing and caching maps from open types to their reps when a generic type or method is instantiated [Kennedy and Syme 2001; Viroli and Natali 2000]. In future work, we will also look to techniques of optimising the coexistence of uniform (boxed) and non-uniform representations in polymorphic code [Leroy 1992] for dealing with the analogous mixture of struct and interface values in generic Go code.

Monomorphisation. Although monomorphisation has been employed for languages such as C++, Standard ML and Rust [Turon 2015], we found a relative lack of peer-reviewed literature on this topic. This section discusses works related to monomorphisation that do not state or prove any correctness results.

Stroustrup [2013, Chapter 26] describes template instantiation in C++. It is widely used, and infamous for code bloat.

Benton et al. [1998] describes a whole-program compiler from SML'97 to Java bytecode, where polymorphism is fully monomorphised. Monomorphisation is alway possible, since Standard ML rules out polymorphic recursion (unlike FGG). Fluet [2015] sketches a similar approach used in the SML optimising compiler MLton.

Tolmach and Oliva [1998, Section 6] develop a typed translation from an ML-like language to Ada, based on monomorphisation and presented in detail. Unlike us, they do not address subtyping (structural or otherwise) and they presume the absence of polymorphic recursion.

Jones [1995] describes the use of specialisation to efficiently compile type classes for Haskell, which bears some resemblance to monomorphisation.

Formalisation. We now consider works that formalise some aspect of monomorphisation.

Yu et al. [2004] formalise the mixed specialisation and sharing mechanisms of the .NET JIT compiler [Kennedy and Syme 2001]. The work describes a type and semantics preserving translation to a *polymorphic* .NET Intermediate Language (IL), where polymorphic behaviours are driven by *type-reps* [Crary et al. 1998], codifying runtime type data that can be used in e.g. dynamic casts. Their approach only generates code where type variables are instantiated with basic data types, using a uniform (i.e. boxed) representation for all other types. This sidesteps the key challenges of monomorphising code with polymorphic recursion and parameterised methods. Notably, Kennedy and Syme [2001] state that "some polymorphism simply cannot be specialized statically (polymorphic recursion, first-class polymorphism)". In contrast, we present an algorithm that can determine whether monomorphisation is possible in the presence of polymorphic recursion.

Siek and Taha [2006] formalise the C++ template instantiation mechanism. They model partial specialization, template parameterisation and instantiation, and prove type soundness of template expansion. Unlike us, they do not state or prove bisimulation or preservation of reductions. Since C++ templates are Turing-complete, their soundness results are modulo termination, whereas our algorithms are guaranteed to terminate (see Theorem 5.1 and [Griesemer et al. 2020]).

Tanaka et al. [2018, Section 2] report on a monomorphisation algorithm for Coq (Gallina) used in generation of low-level C code. Unlike us, they do not test for polymorphic recursion.

Monomorphisation and logic. In a related area, Blanchette et al. [2016]; Bobot and Paskevich [2011] study monomorphisation for polymorphic first-order logic formulas, targeting the untyped or multi-sorted logics found in automated theorem provers.

8 CONCLUSION

In this work we studied generics for a minimal subset of Go and their compilation via monomorphisation. We chose monomorphisation since it is a simple first way of concretely explaining the semantics of generics using (simplified) Go and it avoids restrictions required by the erasure-based approach used in FGJ (e.g. type assertions on type variables). Another key benefit of monomorphisation is that of enabling 0-cost abstractions – programs that do not use generics incur no runtime penalty and generic code is translated into code that is equivalent to hand coded instantiations. The cost is that of requiring a whole program analysis and disallowing programs that would result in infinite instantiations (Section 5.3). Clearly, this is the beginning of the story, not the end.

In future work, we plan to look at other methods of implementation beside monomorphisation, and in particular we plan to consider an implementation based on passing runtime representations of types, similar to that used for .NET generics [Kennedy and Syme 2001]. The idea is to automatically equip methods and structs with data that codifies type arguments used in generic code at runtime. For instance, a `Cons` struct would carry at runtime an additional field that specifies the type of the element contained in the cell, and a method that constructs a tree from a `List` would thread the runtime type information of the list cells into the tree. This approach requires translating all structs and methods of a program to account for runtime type passing and construction and thus the resulting programs can incur some runtime penalty. A mixed approach that uses monomorphisation sometimes and passing runtime representations sometimes might be best, again similar to that used for .NET generics. We will study the trade-offs and performance impact of this spectrum of approaches in future work.

Featherweight Go is restricted to a tiny subset of Go. We plan a model of other important features such as assignments, arrays, slices, and packages, which we will dub Bantamweight Go; and a model of Go's innovative concurrency mechanism based on "goroutines" and message passing, which we will dub Cruiserweight Go.

ACKNOWLEDGMENTS

We thank Nicholas Ng, Sam Lindley, and our referees for comments and suggestions. This work was funded under EPSRC EP/K034413/1, EP/T006544/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T006544/1, EP/T014709/1 and EP/V000462/1, NCSS/EPSRC VeTSS, NOVA LINCS (UIDB/04516/2020) with the financial support of FCT- Fundação para a Ciência e a Tecnologia, and EU MSCA-RISE BehAPI (ID:778233).

REFERENCES

Nada Amin and Ross Tate. 2016. Java and Scala's type systems are unsound: the existential crisis of null pointers. In *Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*. 838–848.

Nick Benton, Andrew Kennedy, and George Russell. 1998. Compiling Standard ML to Java Bytecodes. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, 129–140. https://doi.org/10.1145/289423.289435

Jasmin Christian Blanchette, Sascha Böhme, Andrei Popescu, and Nicholas Smallbone. 2016. Encoding Monomorphic and Polymorphic Types. *Logical Methods in Computer Science* 12, 4 (2016).

François Bobot and Andrey Paskevich. 2011. Expressing Polymorphic Types in a Many-Sorted Language. In *FroCoS (Lecture Notes in Computer Science, Vol. 6989)*. Springer, 87–102.

Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. 1998. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998*. ACM, 183–200. https://doi.org/10.1145/286936.286957

Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C Mitchell. 1989. F-bounded polymorphism for object-oriented programming. In *Functional Programming Languages and Computer Architecture (FPCA)*. 273–280.

Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. 2000. Flow-directed closure conversion for typed languages. In *European Symposium on Programming (ESOP)*. Springer, 56–71.

Koen Claessen, Jonas Duregård, and Michał Pałka. 2015. Generating constrained random data with uniform distribution. *Journal of Functional Programming* 25 (2015). https://doi.org/10.1017/s0956796815000143

William R Cook. 1990. Object-oriented programming versus abstract data types. In *Workshop of the REX Project (LNCS, Vol. 489)*. Springer, 151–178.

Karl Crary, Stephanie Weirich, and J. Gregory Morrisett. 1998. Intensional Polymorphism in Type-Erasure Semantics. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*. ACM, 301–312. https://doi.org/10.1145/289423.289459

Sophia Drossopoulou and Susan Eisenbach. 1997. Java is type safe—probably. In *European Conference on Object-Oriented Programming*. Springer, 389–418.

Jonas Duregård. 2016. *Automating Black-Box Property Based Testing*. Ph.D. Dissertation. Chalmers University of Technology.

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1998. Classes and Mixins. In *Principles of Programming Languages (POPL)*. 171–183.

Matthew Fluet. 2015. MLton – Monomorphise. http://mlton.org/Monomorphise.

Ben Greenman, Fabian Muehlboeck, and Ross Tate. 2014. Getting F-bounded polymorphism into shape. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 89–99. https://doi.org/10.1145/2594291.2594308

Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. 2020. Featherweight Go. *CoRR* abs/2005.11710 (2020). arXiv:2005.11710 https://arxiv.org/abs/2005.11710

Robert Griesemer, Rob Pike, Ken Thompson, Ian Taylor, Russ Cox, Jini Kim, and Adam Langley. 2009. Hey! Ho! Let's Go! https://opensource.googleblog.com/2009/11/hey-ho-lets-go.html

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 1999. Featherwieght Java: A Minimal Core Calculus for Java and GJ. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999.*, Brent Hailpern, Linda M. Northrop, and A. Michael Berman (Eds.). ACM, 132–146. https://doi.org/10.1145/320384.320395

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (2001), 396–450. https://doi.org/10.1145/503502.503505

Mark P Jones. 1995. Dictionary-free overloading by partial evaluation. *Lisp and Symbolic Computation* 8, 3 (1995), 229–248.

Andrew Kennedy and Don Syme. 2001. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*. ACM, 1–12. https://doi.org/10.1145/378795.378797

Shriram Krishnamurthi, Matthias Felleisen, and Daniel P Friedman. 1998. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 91–113.

Xavier Leroy. 1992. Unboxed Objects and Polymorphic Typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*. ACM Press, 177–188. https://doi.org/10.1145/143165.143205

Maurice Naftalin and Philip Wadler. 2006. *Java generics and collections*. O'Reilly. http://www.oreilly.de/catalog/javagenerics/index.html

Tobias Nipkow and David von Oheimb. 1998. Javalight is Type-safe—Definitely. In *Principles of Programming Languages (POPL)*. 161–170.

Benjamin C. Pierce. 1992. Bounded Quantification is Undecidable. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22,*

1992. 305–315. https://doi.org/10.1145/143165.143228

John C Reynolds. 1994. User-defined types and procedural data structures as complementary approaches to data abstraction. In *New Directions in Algorithmic Languages*. MIT Press, 13–23.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck. In *Proceedings of the first ACM SIGPLAN symposium on Haskell - Haskell '08*. ACM Press. https://doi.org/10.1145/1411286.1411292

Jeremy G. Siek and Walid Taha. 2006. A Semantic Analysis of C++ Templates. In *European Conference on Object-Oriented Programming (ECOOP) (LNCS, Vol. 4067)*. Springer, 304–327.

Bjarne Stroustrup. 2013. *The C++ Programming Language, 4th Edition*. Addison-Wesley.

Wouter Swierstra. 2008. Data types à la carte. *Journal of Functional Programming* 18, 4 (2008), 423–436.

Don Syme. 1999. Proving Java Type Soundness. In *Formal Syntax and Semantics of Java*. Springer-Verlag, 83–118.

Akira Tanaka, Reynald Affeldt, and Jacques Garrigue. 2018. Safe Low-level Code Generation in Coq Using Monomorphization and Monadification. *JIP* 26 (2018), 54–72.

Ian Lance Taylor and Robert Griesemer. 2019. Contracts — Draft Design. https://go.googlesource.com/proposal/+/master/design/go2draft-contracts.md

Ian Lance Taylor and Robert Griesemer. 2020. Type Parameters — Draft Design. https://go.googlesource.com/proposal/+/refs/heads/master/design/go2draft-type-parameters.md

The Go Team. 2020. The Go Programming Language Specification. https://golang.org/ref/spec

The Rust Team. 2017. The Rust programming language. http://rust-lang.org/

Andrew P. Tolmach and Dino Oliva. 1998. From ML to Ada: Strongly-Typed Language Interoperability via Source Translation. *Journal of Functional Programming* 8, 4 (1998), 367–412.

Mads Torgersen. 2004. The expression problem revisited. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 123–146.

Aaron Turon. 2015. Abstraction without overhead: traits in Rust. https://blog.rust-lang.org/2015/05/11/traits.html

Mirko Viroli and Antonio Natali. 2000. Parametric polymorphism in Java: an approach to translation based on reflective features. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000*. ACM, 146–165. https://doi.org/10.1145/353171.353182

Philip Wadler. 1998. The expression problem. Posted on the Java Genericity mailing list. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

Dachuan Yu, Andrew Kennedy, and Don Syme. 2004. Formalization of generics for the .NET common language runtime. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. ACM, 39–51. https://doi.org/10.1145/964001.964005

Matthias Zenger and Martin Odersky. 2004. *Independently extensible solutions to the expression problem.* Technical Report.