Welterweight Go: Boxing, Structural Subtyping, and Generics

RAYMOND HU, Queen Mary University of London, United Kingdom
JULIEN LANGE, Royal Holloway, University of London, United Kingdom
BERNARDO TONINHO, Instituto Superior Técnico, University of Lisbon, Portugal and INESC-ID, Portugal
PHILIP WADLER, University of Edinburgh, United Kingdom
ROBERT GRIESEMER, Google, USA
KEITH RANDALL, Google, USA

Go's unique combination of structural subtyping between generics and types with non-uniform runtime representations presents significant challenges for formalising the language.

We introduce WG (Welterweight Go), a core model of Go that captures key features excluded by prior work, including underlying types, type unions and type sets, and proposed new features, such as generic methods. We also develop LWG, a lower-level language that models Go's runtime mechanisms, notably the distinction between raw struct values and interface values that carry runtime type information (RTTI).

We give a type-directed compilation from WG to LWG that demonstrates how the proposed features can be implemented while observing important design and implementation goals for Go: compatibility with separate compilation, and no runtime code generation. Unlike existing approaches based on static monomorphisation, our compilation strategy uses runtime type conversions and adaptor methods to handle the complex interactions between structural subtyping, generics, and Go's runtime infrastructure.

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

Additional Key Words and Phrases: Go, Generics, Boxing

ACM Reference Format:
Raymond Hu, Julien Lange, Bernardo Toninho, Philip Wadler, Robert Griesemer, and Keith Randall. 2026. Welterweight Go: Boxing, Structural Subtyping, and Generics. *Proc. ACM Program. Lang.* 10, POPL, Article 79 (January 2026), 28 pages. https://doi.org/10.1145/3776721

1 Introduction

Go is a popular programming language that is widely used in industry. A key characteristic of Go is *structural* subtyping of interfaces, unlike other memory managed OO languages such as Java and C# that are based on nominal typing.

The first work on formalising Go by Griesemer et al. [2020] tackled the extension of Go with *generics* (bounded parametric polymorphism). It presented a system that integrates Go's structural subtyping and generics in a manner that is compatible with static monomorphisation (as opposed to, e.g., erasure in Java, and dynamic monomorphisation in C#). This shaped the release of generics in Go 1.18 in 2022, described as the biggest change in its history [Griesemer and Taylor 2022].

Authors' Contact Information: Raymond Hu, Queen Mary University of London, London, United Kingdom, r.hu@qmul.ac.uk; Julien Lange, Royal Holloway, University of London, Egham, United Kingdom, Julien.Lange@rhul.ac.uk; Bernardo Toninho, Instituto Superior Técnico, University of Lisbon, Lisbon, Portugal and INESC-ID, Lisbon, Portugal, bernardo.toninho@tecnico.ulisboa.pt; Philip Wadler, University of Edinburgh, Edinburgh, United Kingdom, wadler@inf.ed.ac.uk; Robert Griesemer, Google, Mountain View, USA, gri@google.com; Keith Randall, Google, Mountain View, USA, keithr@alum.mit.edu.

ACM 2475-1421/2026/1-ART79
https://doi.org/10.1145/3776721

This paper tackles several problems in the ongoing design and formal investigation of Go.

- We introduce WG (Welterweight Go). Its purpose is to provide a minimal formal model that captures essential type system features of Go needed to study sound compilation strategies for generics, and to investigate new features proposed for future versions of Go, notably generic methods and method set intersections for generic interfaces. Structural typing in Go implies, for example, that methods can accept differently-named structs with the same underlying structure: several subtleties related to various kinds of type coercion are needed, which are absent from previous work. We focus on generic type unions and type sets, and model the key areas that interact with Go's use of structural typing: anonymous types, underlying types, and primitive data and operators. The combination of these features is absent from models found in the literature. Our design of WG distils the least set of features that captures all the relevant interactions. We prove that WG satisfies preservation and progress properties (Theorems 3.5 and 3.6).
- We formalise a lower level language, LWG, to model key mechanisms of the Go *runtime*. LWG captures essential elements of the design and implementation of Go—notably the runtime distinction between struct types and interface types—that WG (or any prior formalism [Ellis et al. 2022; Griesemer et al. 2020; Sulzmann and Wehr 2023]) on its own does not. We develop LWG as a typed language and establish its own preservation and progress properties (Theorems 4.1 and 4.2).
- We formalise compilation as a type-directed translation from WG to LWG and prove its correctness by a behavioural equivalence (Theorem 5.3). We have designed LWG and compilation to support the proposed new features while respecting the key criteria set by the Go Team: compatibility with separate compilation and no code generation during runtime. Our compilation strategy is based on boxing and runtime type conversions.

Overall, this paper presents the first full account of structural typing for a language with generic structs, methods and interfaces. We establish type safety results and the design of a type and behaviour preserving compilation to a low-level model that is feasible in practice. It provides a framework for investigating the design, implementation and correctness of future extensions to the Go language and runtime. Method set intersections have been proposed but not yet implemented [dominikh 2022; The Go Team 2024]; these would allow interfaces to contain methods common to all listed types. Generic methods have been under consideration in a long-standing proposal [Macías 2021], but have yet to be included in Go due to uncertainty about the best implementation approach [Taylor and Griesemer 2021]. It is important for language designers to investigate if there are bad interactions between features. For instance, Amin and Tate [Amin and Tate 2016] have shown that Java generics are unsound. The issue they found could not be expressed in Featherweight Generic Java [Igarashi et al. 2001], due to its limited feature set.

Formalising LWG and compilation is important for ensuring that the proposed features are indeed compatible with the requirements of Go and demonstrating how they can be implemented in practice. A key distinction between base or struct types and interface types in the Go runtime is that interface types carry runtime type information (RTTI) for dynamic operations (e.g., type assertions) whereas constants and structs do not. The design of Go crucially depends on the compiler being able to *statically* insert runtime operations for safely converting between the distinct cases for constants or structs and interfaces as related by structural subtyping. The prior high level formalisms lack this perspective and fail to faithfully model the Go runtime. For example, they use terms $S\{\ldots\}$ to denote *runtime* struct values, and $S\{\ldots\}.(T)$ for type assertions on structs; whereas actual Go structs, as mentioned, do not carry any RTTI corresponding to the S, and expressions that would dynamically depend on such RTTI are thus not valid in practice.

Unlike the original core formalism of Griesemer et al. [2020] that is based *solely* on static monomorphisation, the actual Go runtime records RTTI for generic type arguments dynamically using dictionaries that are generated by the compiler and passed around at runtime [Randall 2022]. LWG models Go interfaces by pairing RTTI with method tables, capturing a key aspect of the actual Go implementation. We show that our approach in fact allows WG, including the proposed new features, to be directly compiled to LWG without relying on monomorphisation to eliminate genericity or incurring its associated restrictions (e.g., on polymorphic recursion).

A key challenge for the compilation is to deal with structural subtyping and generic types. The interplay between these means that generic code must, in general, be statically compiled to handle two distinct kinds of arguments that can be passed at runtime: base or struct types on the one hand and interface types on the other. Our solution builds on Go's framework for implicit runtime type conversions. It involves generating adaptor code that we can quantify as one extra wrapper method per method defined in the user program. The benefits are that it supports generic methods with separate compilation, as opposed to monomorphisation, which requires a whole program analysis, while as mentioned lifting other present limitations of Go such as the restriction on polymorphic recursion. In general, our compilation is more efficient and results in less code bloat than monomorphisation, but involves more type conversions at runtime.

This work was motivated by discussions with the Go Team who requested feedback on the design of generic type unions and type sets. Two members of the Go Team are coauthors of this paper.

Roadmap. Sec. 2 gives an overview of key concepts in WG, LWG and compilation. Sec. 3 defines WG and establishes preservation and progress properties. Sec. 4 defines LWG and establishes preservation and progress properties (independently of WG). Sec. 5 defines the compilation of WG into LWG and shows that a well-typed WG program and its LWG compilation are behaviourally equivalent. Sec. 6 discusses related work. We have implemented a minimal prototype of WG and our compilation approach in an accompanying artifact [Hu et al. 2025].

2 Overview

2.1 WG by Example

Figure 1 gives a WG example. For readability, we use functions (as opposed to methods), anonymous functions and function types; in our core formalism these can be represented using interfaces, structs and methods. Type `MyInt` is defined with *underlying* basic type **`int`**. `MyInt` is nominally distinct from **`int`**; it can be thought of as a wrapper with no runtime overhead (akin to `newtype` in Haskell). Type `MyFloat` is similar. `MyInt` and `MyFloat` have a method `String` with implementations that involve *static type conversions* based on the underlying types. Interface `MyNum` contains a *type union* specifying that the only members of this type are `MyInt` and `MyFloat`.

Generic type `List` has a type parameter `a` with upper bound **`any`**, an alias for the empty interface (i.e., **`interface`**`{}`), meaning that `a` can be instantiated by any type. `List` is defined as an interface supporting a single method `FoldL[b]`, a fold operation that takes a second type parameter `b`. By structural subtyping, any type `T` that implements `FoldL[b]` is considered a subtype of `List[T]`. Structs `Nil[a]` and `Cons[a]` implement `List[a]`. The two fields of `Cons[a]` depend on its type parameter. Generic function `join` takes a `List` of any type bounded by `MyNum`, and folds the list by calling the `String` method on each item and concatenating them. The idea is that the formals of the (anonymous) combining function safely support `String` because they are bounded by `MyNum`, all members of which implement `String`. The `main` method calls `join` to fold a list of `MyFloat`.

The code above the line in Figure 1 is supported by Go, but not by prior formalisms due to lacking underlying types, type unions and static conversions. The code below is WG only because Go does not yet support: (1) generic methods (see lines 7,10-12); nor (2) *method set intersections* for type unions which is required to support the `String` call at line 15.

```
package main
type MyInt int
func (x MyInt) String() string { return strconv.Itoa(int(x)) }
type MyFloat float64
func (x MyFloat) String() string { return strconv.FormatFloat(float64(x), 'E', -1, 64) }
type MyNum interface { MyInt | MyFloat }
```

```
type List[a any] interface { FoldL[b any](f func(v b, w a) b, z b) b }
type Nil[a any] struct {}
type Cons[a any] struct { head a; tail List[a] }
func (x Nil[a]) FoldL[b any](f func(v b, w a) b, z b) b {return z}
func (x Cons[a]) FoldL[b any](f func(v b, w a) b, z b) b {
  return x.tail.FoldL[b](f, f(z, x.head))
}
func join[a MyNum](x List[a]) String {
  return x.FoldL[string](func(v string, w a) string { return v + ",␣" + w.String() }, "")
}
func main() {
  var xs List[MyFloat] = Cons[MyFloat]{MyFloat(1), Cons[MyFloat]{MyFloat(2), Nil[MyFloat]{}}}
  fmt.Println(join[MyFloat](xs)) // Prints: ", 1E+00, 2E+00"
}
```

Fig. 1. (top) Code supported by Go and WG; (bottom) code supported by WG only.

2.2 LWG: A First Mini Example

This mini example demonstrates how structural subtyping between base or struct types and interface types in WG is handled by the Go compiler and runtime.

```
type C struct { f any }    type INum interface { String() string } // MyInt (etc) implements INum
func bar(x INum) MyInt { return C{x}.f.(MyInt) }    func main() { return bar(MyInt(42)).String() }
```

The function `bar` takes an `INum` and wraps it in a `C` struct with field type **any**. It then accesses the field, and performs a type assertion to `MyInt`. It returns the `MyInt`. The `main` body expression first calls `bar` and passes a `MyInt`, which is well typed because the `MyInt` base type structurally implements all the methods of the `INum` interface. It then calls `String` on the result.

The LWG output of compiling the above code is as follows. For readability, we simplify slightly and use an abridged notation compared to the full formal definitions later.

```
func bar(x INum) MyInt { return {x.(change any)}.0.(MyInt) }
func main() { return bar(42.(make MyInt ρ))#MyInt.String() }
```

LWG models that in the Go runtime, interfaces are implemented as fat pointers that box a value and carry its RTTI and a method table, whereas base types and structs are unboxed raw values with no RTTI or method table. The compiler generates operations for dynamically converting between the two kinds, via the **make** and **change** operations.

- The expression `42.(`**make** `MyInt` ρ`)` will convert (i.e., box) the raw value `42` to an *interface value* of interface type `INum`. While the raw value itself incurs no runtime overhead, when boxed as an interface value it will explicitly carry the RTTI denoted by `MyInt` and the method table denoted by ρ. Here ρ is the *method table* of `MyInt` projected to `INum`, i.e., the `String` method. Both elements are statically computed by the compiler and embedded into the output code.
- The expression `x.(`**change any**`)` will convert the interface value held by `x` at runtime to an interface value for the **any** interface. In general such conversions must be done dynamically as the exact base or struct type (i.e., the RTTI) boxed by the interface value is statically unknown.

- The expression e`#MyInt.String()` is a method call that has been statically resolved by the compiler to be a `String` call on the type `MyInt`. Since the receiver sub-expression e, i.e., `bar(42.(make MyInt ρ))` is of type `MyInt`, it will evaluate to a raw value at runtime; all such calls on a receiver of base or struct type must be statically resolved because the raw values at runtime do not carry any RTTI or method tables for dynamic dispatch.

The LWG reduction proceeds as follows.

	`bar(42.(make MyInt ρ))#MyInt.String()`	Make the interface value `(MyInt,ρ,42)`
⟶	`bar((MyInt,ρ,42))#MyInt.String()`	Call function `bar` (substitute the method body)
⟶	`{(MyInt,ρ,42).(change any)}.0.(MyInt)#MyInt.String()`	Change the interface value to **any**
⟶	`{(MyInt,ϵ,42)}.0.(MyInt)#MyInt.String()`	Select the first field of the struct (i.e., `f` in `C`)
⟶	`(MyInt,ϵ,42).(MyInt)#MyInt.String()`	Assert the boxed type is `MyInt` (and unbox)
⟶	`42#MyInt.String()`	Call the statically resolved method `MyInt.String`
$\longrightarrow^*$	`"42"`	Final result value

Note the **change** to **any** results in an empty method table ϵ since **any** has no methods. Overall, using runtime conversions to keep method tables aligned with the expected interface type is key to Go's implementation of structural subtyping. Since an interface can be openly implemented by any struct with an arbitrary superset of the required methods, aligning the method table with the interface allows calls to be dynamically dispatched based on the method offset for that interface.

The type assertion to `MyInt` dynamically checks the RTTI and, in this case, *unboxes* the value.

2.3 LWG and Generics

The current implementation of Go uses a combination of static monomorphisation and runtime management of RTTI for generic type arguments [Randall 2022]. Prior work [Griesemer et al. 2020] has formalised monomorphisation as a translation from a core subset of generic Go to non-generic Go. By contrast, this paper focuses on formalising the lower level mechanisms of the Go runtime in LWG, and the compilation from WG to LWG. We model Go's RTTI for generic types by reusing the existing infrastructure for interface values: in an interface value (R,ρ,v), we allow the RTTI element R to be a ground generic type, e.g., $S[\overline{T}]$ – $\overline{T}$ is a sequence of ground types that corresponds to the dictionaries used by the Go runtime. In our system, we could consider monomorphisation as an optimisation to eliminate some of the runtime boxing and conversion operations: with appropriate restrictions, we could monomorphise generic WG to non-generic WG and then compile to LWG. However, this paper shows that WG can be compiled directly to LWG and, unlike monomorphisation, requires no restrictions.

We briefly illustrate some of the challenges and subtleties of LWG and our compilation arising from the interplay between structural subtyping, generics and Go's runtime infrastructure.

One key aspect is the type system for LWG. Consider again a runtime interface value `(MyInt,ϵ,42)` that boxes a `MyInt` under the **any** interface. How should we type this interface value? Should it be typed as an **any** according to its interface? Unfortunately, this would break type safety of the compiled code. Below on the left is a well-typed WG expression (recall the generic bound of `Cons` is **any**). On the right is the LWG compilation and its reduction step that converts the constant into the interface value mentioned above: neither expression before nor after the step would be well-typed because the struct requires its first field expression to be of type `MyInt`, not **any**.

```
Cons[MyInt]{MyInt(42), ...}    Cons[MyInt]{42.(make MyInt ϵ), ...} ⟶ Cons[MyInt]{(MyInt,ϵ,42), ...}
```

Should the interface value be typed as a `MyInt` according to its RTTI? Unfortunately, this would break progress for LWG: bogus LWG expressions such as `(MyInt,ϵ,42).String()` would be well-typed but stuck, as the `String` call cannot be dispatched by an empty method table ϵ. Neither option is adequate on its own. Section 4 presents the LWG type system which tracks *both*.

```
type List[a any] interface { ...; Fold(f func(v a, w a) a, z a) a }
func (x Cons[a]) Fold(f func(v a, w a) a, z a) a {return x.FoldL[a](f, z)}
type MyIntCons struct { head MyInt; tail List[MyInt] }
func (x MyIntCons) FoldL[b any](f func(v b, w MyInt) b, z b) b {return x.tail.FoldL[b](...)}
func (x MyIntCons) Fold(f func(v MyInt, w MyInt) MyInt, z MyInt) MyInt {return x.FoldL[MyInt](...)}
```

Fig. 2. Extending the example from Fig. 1.

As a final example for this overview, consider the WG code in Figure 2 where we add a second method to `List` and define a version of `Cons` specifically for `MyInt`. Note that both `Cons[MyInt]` and `MyIntCons` are structural subtypes of `List[MyInt]`. Now consider this WG function:

```
func foo(x List[MyInt]) MyInt { return x.Fold(..., 0) }
```

How should we compile this function? How should the argument `0` be passed in the call to `Fold`? Statically, we do not know if the runtime value of `x` will be (i) a `Cons[MyInt]`, noting that the compilation of `Fold` for `Cons[a]` expects an interface value for its generic formal bounded by **`any`**, or (ii) a `MyIntCons`, where `Fold` expects a raw `MyInt`. Sec. 5 presents our compilation that addresses this problem by building on Go's approach to statically generating runtime type conversions.

3 Welterweight Go

In this section we introduce Welterweight Go (WG), an extension of Featherweight Generic Go (FGG) [Griesemer et al. 2020], a core calculus of Go with generics. WG extends FGG with a selection of core and recent type-level additions to Go, notably anonymous (interface and struct) types, type unions (an idiomatic form of untagged sum types) and static type conversions.

Type unions bridge Go's interface bounds with basic types and operators. Go overloads operators like $<$ on basic types (integers, floats, strings) but requires arguments of the same type. Since operators are not methods, interface bounds alone cannot permit their use on generic types, creating a gap that type unions fill. For example, it is not possible to define a generic `Min` function by appealing to the overloaded $<$ operator via interface bounds alone. To account for this lack of expressiveness, the Go team introduced type unions, which enable the kind of genericity mentioned above. A generic `Min` function can be written as:

```
type  Ordered interface { int | float64 | ~string }
func Min[T Ordered](x T, y T) T { if x < y { return x } else { return y } }
```

The `Ordered` interface denotes a *type union*, satisfied by any **`int`**, **`float64`** or any type whose *underlying type* is **`string`** ($\sim U$ refers to the set of all types whose underlying type is U). The use of `Ordered` as a bound for type parameter `T` warrants the use of any built-in operator that is common to all types in the union, making the definition correct. The mix of $\sim$ and non-$\sim$ elements offers fine-grained control over the constraint: `T` may be instantiated by **`int`** specifically but not other named types with **`int`** as the underlying representation, whereas `T` may be instantiated by any type with **`string`** as its underlying representation.

In their full generality, type unions can include lists of both types and methods, and can be used as named or anonymous type parameter bounds in the language (although they cannot be used directly as types). Type unions in the language generalises interfaces to be understood not only as sets of methods but *also* as sets of *types.*

From FGG to WG. Whereas FGG provides a faithful model of Go's subtyping relation for *named* interfaces, WG fully models both implementability of anonymous and named interface types, relying on method and type sets to faithfully account for type unions, as well as Go's concept of

Field name	f	Type term	$C ::= T \mid {\sim}T$
Method name	m	Type union	$E ::= C \mid C{\mid}E$
Variable name	x	Interface element	$F ::= mM \mid E$
Type parameter	α	Constants	$c ::= 0 \mid 1 \mid \ldots$
Type name	$t, u \qquad (t, u \neq B)$	Operators	$\circ ::= + \mid * \mid \wedge \mid \ldots$
Base types	$B ::=$ `int` $\mid$ `bool` $\mid \ldots$	Expression	$e ::=$
Type	$R, S, T, U ::= B \mid L \mid t[\overline{S}] \mid \alpha$	Variable	x
Type/bound pair	$\Phi, \Psi ::= T@V$	Method call	$e.m[\overline{S}](\overline{e})$
Type literal	$L ::=$	Structure literal	$T\{\overline{e}\}$
Structure	$\textbf{struct}\ \{\overline{f\ \Phi}\}$	Select	$e.f$
Interface	$\textbf{interface}\ \{\overline{F}\}$	Type assertion	$e.(T)$
Method signature	$M ::= [\overline{\alpha\ S}](\overline{x\ \Phi})\ \Psi$	Type conversion	$T(e)$
Declaration	$D ::=$	Constant	$T(c)$
Type declaration	$\textbf{type}\ t[\overline{\alpha\ S}]\ T$	Operation	$\circ(\overline{e})$
Method declaration	$\textbf{func}\ (x\ t[\overline{\alpha\ S}])\ mM\ \{\textbf{return}\ e\}$		
Program	$P ::= \textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{\_ = e\}$		

Fig. 3. WG syntax

assignability, which generalises implementability to account for anonymous types. Moreover, to accurately model the full range of features provided by type unions, we consider base types and operators in WG. Finally, in WG we can also naturally account for Go's type *conversion* operator. This is necessarily absent from FGG, which includes only type *assertion*.

Assignability in Go relies on the notion of *underlying type* which in turn is a cornerstone of the language formalism. In the type union `Ordered`, the clause that includes string types is written ~**`string`**. Go allows a new type with the same underlying memory representation as another to be defined as a nominally distinct type and to which, for instance, additional methods can be attached. By using the ~ type operator we can include not just the type (e.g.) **`string`** itself, but also all types whose *underlying type* is **`string`** (i.e., all types whose underlying representation is a **`string`**).

3.1 WG Syntax

The syntax of WG is given in Figure 3. Field names f, method names m, variable names x, type parameters α, and type names t form the basic identifiers. Types T include base types B (such as `int` and `bool`), type literals L, (parameterised) named types $t[\overline{S}]$, type parameters α. We write $\overline{S}$ for a (possibly empty) list of S_i. We use the notation $T@V$ to annotate types with their bounds, where T is the actual type and V is its bound. In WG, when T is a type parameter α, V represents its declared bound; when T is a concrete instantiation of a type parameter, V represents the bound of that parameter; otherwise $V = T$. For instance, in a struct type literal $\textbf{struct}\ \{\overline{f\ \Phi}\}$, we assume that every field type is annotated with its bound. Such annotations can be straightforwardly inferred during type checking. Our formalisation of WG would be identical if we replaced annotations $T@V$ with simple types T. However, the type-directed compilation to LWG described in Section 5 would become significantly more complex to present without these annotations.

An interface type literal $\textbf{interface}\ \{\overline{F}\}$ contains a sequence of elements, i.e., method signatures (M) or *unions* of type terms (E). Each union specifies that the interface is only satisfied by its members. Type terms C can be either types T, or their approximation ${\sim}T$ which specifies all types whose underlying type is T. Note, a type declaration $\textbf{type}\ U\ T$ introduces a new named type U with underlying type given by T: if T is also named, Go's notion of assignability (Sec. 3.2) distinguishes

$$\mathit{under}_\Delta(B) = B \qquad \mathit{under}_\Delta(L) = L \qquad \frac{(\mathbf{type}\ t[\overline{\alpha\ S}]\ T) \in \overline{D} \qquad \eta = (\overline{\alpha := R})}{\mathit{under}_\Delta(t[\overline{R}]) = \mathit{under}_\Delta(T[\eta])} \qquad \frac{(\alpha : T) \in \Delta}{\mathit{under}_\Delta(\alpha) = \mathit{under}_\Delta(T)}$$

$$\mathit{under}_\Delta(\sim V) = V \qquad \frac{(\alpha : T) \in \Delta}{\mathit{tyvar}_\Delta(\alpha)} \qquad \frac{(\alpha : T) \in \Delta}{\mathit{noUnion}_\Delta(\alpha)} \qquad \frac{\mathit{struct}(T)}{\mathit{noUnion}_\Delta(T)} \qquad \frac{\mathit{base}(T)}{\mathit{noUnion}_\Delta(T)}$$

$$\frac{\mathit{under}_\Delta(T) = \mathbf{interface}\ \{\overline{mM}\}}{\mathit{noUnion}_\Delta(T)} \qquad \frac{}{T@V\downharpoonleft_1 = T\downharpoonleft_1} \qquad \frac{}{T@V\downharpoonleft_2 = V} \qquad \frac{(\alpha : T) \in \Delta}{\mathit{bounds}_\Delta(\alpha) = T}$$

$$\frac{\neg\mathit{tyvar}_\Delta(T)}{\mathit{bounds}_\Delta(T) = T} \qquad \frac{\mathit{under}_\Delta(T) = \mathbf{struct}\ \{\overline{f\ \Phi}\}}{\mathit{fields}_\Delta(T) = \overline{f\ \Phi}} \qquad \frac{}{(m[\overline{\alpha\ S}](\overline{x\ \Phi})\ \Psi)\downharpoonleft_1 = m[\overline{\alpha\ S}](\overline{x\ \Phi\downharpoonleft_1})\ \Psi\downharpoonleft_1}$$

$$\frac{(\mathbf{func}\ (x\ t[\overline{\alpha\ S}])\ m[\overline{\beta\ T}](\overline{y\ \Phi})\ \Psi\ \{\mathbf{return}\ e\}) \in \overline{D} \qquad \theta = (\overline{\alpha := S'} \cup \overline{\beta := T'})}{\mathit{body}(t[\overline{S'}].m[\overline{T'}]) = (x, \overline{y : \Phi\downharpoonleft_1}[\theta]).e : \Psi\downharpoonleft_1[\theta]}$$

Fig. 4. WG: auxiliary definitions

T and U nominally, but $\sim T$ allows to express compatibility of (nominally-distinct) types with the same underlying type. Method signatures M take the form $[\overline{\alpha\ S}](\overline{x\ \Phi})\ \Psi$, where type parameters $\overline{\alpha}$ are bounded by types $\overline{S}$, parameters $\overline{x}$ have types $\overline{\Phi}$, and the return type is given by Ψ. Type literals can be used in place of named types, in which case we call them *anonymous types.*

Expressions include variables, method calls, structure literals, field selection, type assertions, type conversions, typed constants, and operations. We model only explicitly typed constants $T(c)$ such as **`int`**`(42)` or **`float64`**`(42.0)`, abstracting away Go's untyped constants as their type inference is orthogonal to our work. 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 often abbreviate it as $\overline{D} \rhd e$.

3.2 Typing and Subtyping in WG

Go's type system is built around interface satisfaction through structural subtyping (a.k.a. "duck typing"). A key ingredient of subtyping in Go is the notion of *underlying type* (written $\mathit{under}_\Delta(T)$, where Δ is a type environment) as formalised in Figure 4 along with other auxiliary definitions.

The underlying type of a base type (resp. type literal) is itself. The underlying type of a type variable is its bound, recorded in Δ, and the underlying type of a named type is the underlying type of the RHS of its declaration. We use the following notation: $\mathit{iface}(T)$ holds when T is an interface or a type variable, $\mathit{struct}(T)$ holds when T is a struct, $\mathit{base}(T)$ when T's underlying type is a base type (see the online extended version for formal definitions). We define a few additional predicates: *tyvar* is used to distinguish type variables from other types, *noUnion* is used to discriminate interfaces that specify type unions. Figure 4 also introduces functions to retrieve the bound of a type, the fields of a struct, and the body of a method. Function $T@V\downharpoonleft_1$ returns the first element of a pair type, recursively applying the transformation; $T@V\downharpoonleft_2$ returns the second such element.

The Go specification defines interface satisfaction with two flavours: implementability and assignability. We formalise both relations in Figure 5.

Implementability determines whether a type satisfies an interface. Assignability, on the other hand, governs when a value of one type can be assigned to a variable of another type, extending implementability. Before Go 1.18, an interface was solely defined by its methods and so any type implementing those methods would satisfy the interface, regardless of its declared type. To

Types of interfaces $\boxed{\mathit{types}_\Delta(T)}\ \boxed{\mathit{types}_\Delta(F)}\ \boxed{\mathit{types}_\Delta(C)}\ \boxed{\mathit{types}_\Delta(E)}$

$$\frac{\neg \mathit{iface}(T)}{\mathit{types}_\Delta(T) = \{T \downharpoonleft_1\}} \qquad \frac{\mathit{under}_\Delta(T) = \textbf{interface}\ \{\overline{F}\}}{\mathit{types}_\Delta(T) = \bigcap\{\mathit{types}_\Delta(F_i) \mid F_i \in \overline{F}\}} \qquad \frac{}{\mathit{types}_\Delta(mM) = \mathbb{U}}$$

$$\frac{}{\mathit{types}_\Delta({\sim}T) = \{{\sim}T \downharpoonleft_1\}} \qquad \frac{}{\mathit{types}_\Delta(E) = \bigcup\{\mathit{types}_\Delta(C) \mid C \in E\}}$$

Methods of interface $\boxed{\mathit{methods}_\Delta(T)}\ \boxed{\mathit{methods}_\Delta(F)}\ \boxed{\mathit{methods}_\Delta(C)}$

$$\frac{}{\mathit{methods}_\Delta(B) = \emptyset} \qquad \frac{\mathit{struct}(L)}{\mathit{methods}_\Delta(L) = \emptyset} \qquad \frac{\mathit{under}_\Delta(T) = \textbf{interface}\ \{\overline{F}\}}{\mathit{methods}_\Delta(T) = \bigcup\{\mathit{methods}_\Delta(F_i) \mid F_i \in \overline{F}\}}$$

$$\frac{\neg \mathit{iface}(t[\overline{S'}])}{\mathit{methods}_\Delta(t[\overline{S'}]) = \{mM[\eta] \mid (\textbf{func}\ (x\ t[\overline{\alpha\ S}])\ mM\ \{\textbf{return}\ e\}) \in \overline{D},\ \eta = (\overline{\alpha := S'})\}}$$

$$\frac{}{\mathit{methods}_\Delta(mM) = \{mM\}} \qquad \frac{}{\mathit{methods}_\Delta({\sim}T) = \mathit{methods}_\Delta(T)} \qquad \frac{|E| > 1}{\mathit{methods}_\Delta(E) = \bigcap\{\mathit{methods}_\Delta(C) \mid C \in E\}}$$

Implements $\boxed{T <:_\Delta U}$

$$<:_\alpha\ \frac{}{\alpha <:_\Delta \alpha} \qquad <:_T\ \frac{\neg \mathit{iface}(T)}{T <:_\Delta T}$$

$$<:_I\ \frac{\begin{array}{c}\mathit{iface}(U) \qquad \neg \mathit{tyvar}_\Delta(U) \qquad \mathit{methods}_\Delta(T)\downharpoonleft_1 \supseteq \mathit{methods}_\Delta(U)\downharpoonleft_1 \\ \forall C \in \mathit{types}_\Delta(T) : C \in \mathit{types}_\Delta(U) \lor ({\sim}V \in \mathit{types}_\Delta(U) \land \mathit{under}_\Delta(C) = V)\end{array}}{T <:_\Delta U}$$

Assignable $\boxed{T \prec:_\Delta U}$

$$\prec:_{<:}\ \frac{T <:_\Delta U}{T \prec:_\Delta U} \qquad \prec:_{tL}\ \frac{\mathit{under}_\Delta(t[\overline{S}]) = L}{t[\overline{S}] \prec:_\Delta L} \qquad \prec:_{Lt}\ \frac{L = \mathit{under}_\Delta(t[\overline{S}])}{L \prec:_\Delta t[\overline{S}]}$$

Fig. 5. Implements and Assignability relations. $\mathbb{U}$ is defined s.t. $T \in \mathbb{U}$ for all T.

handle operators on generic types, interfaces were extended in Go 1.18, to also include explicit type unions, allowing developers to restrict which types can satisfy an interface beyond just method requirements. Figure 5 formalises this through two auxiliary functions that return type sets ($\mathit{types}_\Delta(T)$) and method sets ($\mathit{methods}_\Delta(T)$).

The $\mathit{types}_\Delta(T)$ function returns the type set of a type, and particularly the set of types explicitly listed in an interface, possibly flagged with ${\sim}T$ notation. The type set of a non-interface type, is the singleton set of this type. The type set of elements of an interface correspond to the intersection of their respective type sets. The type set of a method signature is the universe of all types ($\mathbb{U}$), i.e., methods alone do not constrain types. Let $\mathbb{U}$ be the universe of all types, axiomatically defined such that for any type T we have that $T \in \mathbb{U}$. The type set of an approximation is itself (${\sim}T$). The type set of a union of types is the union of their type sets.

The $\mathit{methods}_\Delta(T)$ function extracts method signatures attached to a type T. The method set of base types and anonymous structs is empty. The method set of an interface is the union of the method set of its elements. The method set of a named, non-interface type is the (instantiated) signatures of its declared methods. The method set of a method signature is the singleton of itself.

Well-formed types $\boxed{\Delta \vdash T \checkmark_{\mathrm{b}}}$ $\boxed{\Delta \vdash \Phi \checkmark_{\mathrm{m}}}$ $\boxed{\Delta \vdash T \checkmark}$

$$\textsc{T-NC}\quad \frac{\Delta \vdash T \checkmark \qquad \mathit{noUnion}_{\Delta}(T)}{\Delta \vdash T \checkmark_{\mathrm{b}}} \qquad\qquad \textsc{T@V-BC}\quad \frac{\Delta \vdash T \checkmark_{\mathrm{b}} \qquad V = \mathit{bounds}_{\Delta}(T)}{\Delta \vdash T@V \checkmark_{\mathrm{m}}}$$

$$\textsc{T-NAMED}\quad \frac{\Delta \vdash \overline{R'} \checkmark \qquad (\textbf{type}\ t[\overline{\alpha\ R}]\ T) \in \overline{D} \qquad \eta = (\overline{\alpha := R'}) \qquad \overline{\alpha[\eta] <:_{\Delta} R[\eta]}}{\Delta \vdash t[\overline{R'}] \checkmark}$$

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

$$\textsc{T-TYPE}\quad \frac{\Delta = \overline{\alpha : R} \qquad \neg \mathit{tyvar}_{\Delta}(T) \qquad \emptyset \vdash \overline{\alpha\ R} \checkmark \qquad \Delta \vdash T \checkmark}{\textbf{type}\ t[\overline{\alpha\ R}]\ T \checkmark}$$

$$\textsc{T-FUNC}\quad \frac{\begin{array}{c} \neg \mathit{iface}(t[\overline{\alpha}]) \qquad \mathit{distinct}(x, \overline{y}) \qquad (\textbf{type}\ t[\overline{\alpha\ R}]\ \_) \in \overline{D} \qquad \emptyset \vdash \overline{\alpha\ R} \checkmark \qquad \overline{\alpha : R} \vdash \overline{\beta\ S} \checkmark \\ \Delta = \overline{\alpha : R}, \overline{\beta : S} \qquad \Delta \vdash \overline{\Phi} \checkmark_{\mathrm{m}} \qquad \Delta \vdash \Psi \checkmark_{\mathrm{m}} \qquad \Delta; x : t(\overline{\alpha}), \overline{y : \Phi\downharpoonleft_1} \vdash e : U \qquad U \prec:_{\Delta} \Psi\downharpoonleft_1 \end{array}}{\textbf{func}\ (x\ t[\overline{\alpha\ R}])\ m[\overline{\beta\ S}](\overline{y\ \Phi})\ \Psi\ \{\textbf{return}\ e\} \checkmark}$$

Fig. 6. Well-formed types and declarations

The method set of $\sim T$ is the method set of T. The method set of a union of types is the intersection of the method sets of its elements (note that when $|E| = 1$, case T or case $\sim T$ applies).

We can now define subtyping precisely. Figure 5 (bottom) defines interface implementation ($<:_{\Delta}$) and assignability ($\prec:_{\Delta}$) relations. A type T implements an interface U if (1) T's method set is a superset of U's method set and (2) T's type set is a subset of U's type set. The function $\_\downharpoonleft_1$ essentially erases (recursively) our type annotations in method signatures and within types, e.g., $T@V\downharpoonleft_1 = T\downharpoonleft_1$. Aspect (2) involves checking that each element in the type set of T is directly in the type set of U or is otherwise covered by an appropriate $\sim V$.

We note that the official Go Language Specification [The Go Team 2025] defines interface satisfaction purely in terms of type sets. We give an equivalent definition that explicitly tracks both type sets and method sets as separate components of our subtyping judgements. This design choice faithfully models the actual implementation of the Go compiler, which maintains analogous internal representations tracking both aspects independently.

Assignability extends implementation with rules for dealing with named types and their underlying types, capturing Go's implicit type conversion rules. Note that $\prec:$ is reflexive via $<:$, but it is not transitive, as Rules $\prec:_{tL}$ and $\prec:_{Lt}$ require one of the types to be anonymous.

The WG type system relies on type and method declarations to be well-formed. We show key rules in Figure 6. The judgement $\mathcal{J} \checkmark$ ensures types are well-formed, possibly under variable and type parameter contexts. We specify two variants of this judgement: $\Delta \vdash T \checkmark_{\mathrm{b}}$ holds when T is a basic interface, i.e., not a union as defined in Figure 4. In WG and in Go, variables and parameters must be assigned a type T such that $T \checkmark_{\mathrm{b}}$ holds. Judgement $\Delta \vdash \Phi \checkmark_{\mathrm{m}}$ simply enforces that our meta-theoretic annotations are well-formed.

Type declarations in WG require their bound types and body to be well-formed. Method declarations enforce additional constraints: the receiver must be a non-interface type, parameters must be distinct, and the method body must be well-typed under the appropriate context, see Figure 6. We relegate some wellformedness rules for interface and struct definition to the online appendix as

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

T-CALL

$$\frac{\begin{array}{ccccc} & \Delta \vdash \overline{S'} \checkmark_{\mathrm{b}} & (m[\overline{\alpha\ S}](\overline{x\ \Phi})\ \Psi) \in \mathit{methods}_\Delta(R) & & \\ \Delta; \Gamma \vdash e : R & \Delta; \Gamma \vdash \overline{e : T} & \eta = (\overline{\alpha := S'}) & \overline{\alpha[\eta] <:_\Delta S[\eta]} & \overline{T[\eta] \prec:_\Delta \Phi\downharpoonleft_1[\eta]} \end{array}}{\Delta; \Gamma \vdash e.m[\overline{S'}](\overline{e}) : \Psi\downharpoonleft_1[\eta]}$$

T-LITERAL

$$\frac{\mathit{struct}(T) \qquad \Delta \vdash T \checkmark_{\mathrm{b}} \qquad \Delta; \Gamma \vdash \overline{e : S} \qquad (\overline{f\ \Phi}) = \mathit{fields}_\Delta(T) \qquad \overline{S \prec:_\Delta \Phi\downharpoonleft_1}}{\Delta; \Gamma \vdash T\{\overline{e}\} : T}$$

T-VAR

$$\frac{(x : T) \in \Gamma}{\Delta; \Gamma \vdash x : T}$$

T-FIELD

$$\frac{\Delta; \Gamma \vdash e : T \qquad \mathit{struct}(T) \qquad (\overline{f\ \Phi}) = \mathit{fields}_\Delta(T)}{\Delta; \Gamma \vdash e.f_i : \Phi_i\downharpoonleft_1}$$

T-OP

$$\frac{\Delta; \Gamma \vdash \overline{e} : T \qquad \mathit{under}_\Delta(T) = B \qquad B \in \mathit{dom}(\circ)}{\Delta; \Gamma \vdash \circ(\overline{e}) : T}$$

T-OP-α

$$\frac{\Delta; \Gamma \vdash \overline{e} : \alpha \qquad \{\mathit{under}_\Delta(V) \mid V \in \mathit{types}(\Delta(\alpha))\} \subseteq \mathit{dom}(\circ)}{\Delta; \Gamma \vdash \circ(\overline{e}) : \alpha}$$

Programs $\boxed{P \checkmark}$

T-PROG

$$\frac{\mathit{distinct}(\mathit{tdecls}(\overline{D})) \qquad \mathit{distinct}(\mathit{mdecls}(\overline{D})) \qquad \overline{D} \checkmark \qquad \emptyset; \emptyset \vdash e : T}{\textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{\_ = e\} \checkmark}$$

Fig. 7. WG typing

they are similar to those of FGG. A wellformedness restriction is that interfaces are restricted so that unions cannot include type variables nor interfaces with non-empty method sets.

We show key typing rules in Figure 7. For simplicity, we enforce that operators take as argument and return type values of the same type. Generalising the typing of operators is straightforward but orthogonal to our design. The expression typing judgement $\Delta; \Gamma \vdash e : T$ (where Δ maps type parameters to their bounds and Γ maps variables to their types) follows typical patterns. For instance, Rule T-CALL handles generic type parameter instantiation through substitution η, it also ensures that the instantiated type parameters satisfy their bounds. Observe how implementability ($<:_\Delta$) is used to check that type instances implements their bounds, while assignability ($\prec:_\Delta$) is used to check that struct fields and method arguments are compatible with their declared types. Rule T-OP deals with operators when the underlying type of the operands is a base type. We write $\mathit{dom}(\circ)$ for the set of types on which $\circ$ is defined. Rule T-OP-α allows the operands to be typed with a type variable α, in which case all types in the bound of α must support $\circ$.

For the sake of space, typing rules for type assertions (as in FGG), constants (standard), and conversions are in the online appendix. Essentially a conversion $T(e)$ is well-typed if e has type U and either $U \prec:_\Delta T$ or $\mathit{under}_\Delta(U) = \mathit{under}_\Delta(T)$, for a suitable Δ.

Example 3.1 (Assignability). *Consider the following well-typed WG (and Go) code:*

```
type Point struct { x int; y int }          func (p Point) move() {}
type Coord struct { x int; y int }          type Converter struct { }
func (c Converter) toPoint(coord struct{ x int, y int}) Point { return coord }
func (c Converter) toAnonymous(p Point) struct{ x int; y int} { return p }
```

The code above defines two identical named struct types `Point` and `Coord`, with `Point` having a method `move`, and a struct `Converter` with methods `toPoint` and `toAnonymous`, both with the same body. Method toPoint essentially converts its (anonymously typed) argument into a Point struct, while toAnonymous does the reverse. Intuitively, the body of toPoint is well-typed because an anonymous struct of the specified shape matches exactly with the shape of type Point and so the **return** is justified (**struct**{x **int**; y **int**} $\prec:$ Point). Dually, we can always treat a named type as its (anonymous) underlying type and so the body of toAnonymous is also well-typed (Point $\prec:$ **struct**{x **int**; y **int**}). Notably, the calls Converter{}.toPoint(**struct**{x **int**; y **int**}{1,1}), Converter{}.toPoint(Point{1,1}) and Converter{}.toPoint(Coord{1,1}) are all well-typed in WG (and Go). Similarly, Converter{}.toAnonymous(Point{1,1}) is well-typed. However, the invocation Converter{}.toAnonymous(Coord{1,1}) is ill-typed since Coord $\prec:$ Point does not hold.

A program is well-typed if its declarations are well-formed and pairwise distinct.

Example 3.2 (Implementability and Unions). *Consider the following well-typed WG code:*

```
type Addable interface { int | float64 | ~string }
type Printable interface { ~int | ~string ; customPrint() string }
type MyString string
func (x MyString) customPrint() string { ... }
type MyInt int
func (x MyInt) customPrint() string { ... }
type C struct { }
func (t C) Combine[T Addable](x T, y T) T { return x+y }
func (t C) Print[T Printable](x T) string { return x.customPrint() }
```

The Addable type union includes types **int**, **float** and all types whose underlying type is **string**. MyString and MyInt have both a custom printing method. It is *not* the case that both types satisfy the constraint specified by Addable: C{}.Combine[MyString](MyString("a"),MyString("b"))) is well-typed, but C{}.Combine[MyInt](MyInt(2),MyInt(3)) is not. Since Addable lists **int** explicitly but refers to ~**string**, we have *types*(Addable) = {**int**, **float**, **string**, MyString} and so MyString <: Addable but MyInt $\not<:$ Addable.

The example also defines interface Printable, consisting of all types whose underlying types are either **int** or **string** that also have a customPrint method defined on them. Thus, method Print and the calls C{}.Print[MyInt](MyInt(2)) and C{}.Print[MyString](MyString("a")) are all well-typed, since *types*(Printable) = {**int**, **string**, MyInt, MyString} and *methods*(Printable) = { customPrint() **string** } and thus MyString <: Printable and MyInt <: Printable.

3.3 Operational Semantics of WG

The operational semantics of WG are presented in Figure 8, defining a reduction relation $d \longrightarrow e$ defined over a standard evaluation context. Values in WG consist of structure literals $T\{v\}$ and typed constants $T(c)$. Given such a value we define $type(T\{\overline{v}\}) = type(T(c)) = T$.

Most rules are standard: Rule R-OP evaluates primitive operations via a semantic function δ; the type conversion rules R-CONVERT-B, -S and -I specify the behaviour of type conversions. A conversion to a base type or a struct type collapses all other (nested) type conversions. A conversion to an interface type is silently erased, due to the nature of values in WG. The type environment is empty in $\mathit{fields}_\emptyset$ in R-FIELD (resp. $<:_\emptyset$ in R-ASSERT) since reductions consider closed, ground terms.

While the essence of the operational semantics is standard, we note that field selection and method calls are augmented with type conversions. The field selection rule R-FIELD introduces a type conversion to the type of the projected field. Similarly, the method invocation rule R-CALL adds type conversions of the method arguments and its eventual return value.

These type conversions are key in bridging the gap between the type system's (structural) subtyping relations and the runtime representation of values. Recall Example 3.1. If we consider

Values	$v ::= T\{\overline{v}\} \mid T(c)$		
Evaluation context	$E ::=$		
Hole	$\Box$	Structure	$T\{\overline{v}, E, \overline{e}\}$
Operator	$\circ(\overline{v}, E, \overline{e})$	Select	$E.f$
Method call receiver	$E.m[\overline{T}](\overline{e})$	Type assertion	$E.(T)$
Method call arguments	$v'.m[\overline{T}](\overline{v}, E, \overline{e})$	Type conversion	$T(E)$

Reduction $\boxed{d \longrightarrow e}$

$$\textsc{r-field}\ \frac{(\overline{f\ \Phi}) = \mathit{fields}_{\emptyset}(T)}{T\{\overline{v}\}.f_i \longrightarrow \Phi_i \downharpoonright_1 (v_i)} \qquad \textsc{r-call}\ \frac{(x, \overline{y : T}).e : U = \mathit{body}(\mathit{type}(v').m[\overline{S}])}{v'.m[\overline{S}](\overline{v}) \longrightarrow U(e[x := v', \overline{y := T(v)}])} \qquad \textsc{r-assert}\ \frac{\mathit{type}(v) <:_{\emptyset} T}{v.(T) \longrightarrow v}$$

$$\textsc{r-op}\ \frac{\delta(\circ, \overline{v}) = v'}{\circ(\overline{v}) \longrightarrow v'} \qquad \textsc{r-convert-b}\ \frac{\mathit{base}(T)}{T(U(c)) \longrightarrow T(c)} \qquad \textsc{r-convert-s}\ \frac{\mathit{struct}(T)}{T(U\{\overline{v}\}) \longrightarrow T\{\overline{v}\}} \qquad \textsc{r-convert-i}\ \frac{\mathit{iface}(T)}{T(v) \longrightarrow v} \qquad \textsc{r-context}\ \frac{d \longrightarrow e}{E[d] \longrightarrow E[e]}$$

Fig. 8. WG reduction

the dynamics of calls to methods `toPoint` and `toAnonymous` from the example, the need for type conversions becomes clear: the call `Converter{}.toPoint(struct{x int; y int}{1,1})` requires a conversion to type `Point` since we must be able to treat the return value of `toPoint` as if it were a `Point` struct. For instance, we may use it as a receiver of a call to `move`, which is only defined on `Point`. Similarly, the call `Converter{}.toAnonymous(struct{x int; y int}{1,1})` also requires a conversion of its argument to `Point`—the body of the method can in principle call upon any method of `Point`. Since method calls look up the appropriate method body by inspecting the runtime type of the receiver, these type conversions ensure that the runtime objects line up accordingly. The call `Converter{}.toPoint(struct{x int; y int}{1,1})` thus evaluates to `Point{1,1}` as needed.

Note that although it is not the case that types `Coord` and `Point` are assignable, we may use type conversions between the two types. In a call such as `Converter{}.toPoint(Coord{1,1})`, our semantics will first type convert `Coord` to its anonymous underlying struct type (the type conversion of the method argument) and from that type to `Point` (the conversion of the return value). The method call results in the expression `Point(struct{x int; y int}(Coord{1,1}))`, which reduces in one step to `Point(struct{x int; y int}{1,1})` and then to the value `Point{1,1}`.

While the description above focuses on method calls, struct field accesses must behave similarly, and so those too require explicit type conversions in our semantics.

3.4 Metatheory of WG

In this section we report the type safety of WG, following mostly standard arguments of type preservation and progress. As usual, type preservation relies on type and expression-level substitution lemmas (and several standard related lemmas that relate substitutions, subtyping and type well-formedness which we omit from the main text for the sake of conciseness).

Notably, due to the type conversions that are generated by our operational semantics, the substitution property need not appeal to subtyping.

Lemma 3.3 (Term Substitution). *If* $\Delta; \Gamma, \overline{x : T} \vdash e : U$ *and* $\Delta; \Gamma \vdash \overline{e' : T}$ *then* $\Delta; \Gamma \vdash e[\overline{x := e'}] : U$.

As usual in languages with subtyping, type preservation is defined by appealing to the implements relation, essentially due to the dynamics of rules r-convert-i and r-assert. However, there is a subtlety due to the fact that most typing rules rely on *assignability*. As this relation is not transitive, the contextual cases of the type preservation proof are less standard. However, we can derive a

Field index	i		
Method table	$\rho ::= \overline{m \mapsto T.m} \cup \overline{\circ \mapsto \circ_T}$		
Expression	$e ::=$		
Variable	x		
Dynamic Method call	$e.m[\overline{S}](\overline{e})$	Type assertion	$e.(T@V)$
Static Method call	$e\#T.m[\overline{S}](\overline{e})$	Make Type	$e.(S \to_\rho \Phi)$
Structure literal	$T\{\overline{e}\}$	Change Type	$e.(S \to T@V)$
Interface value	$\Phi(S, \rho, e)$	Static Change	$e.(S \twoheadrightarrow T)$
Select	$e.i$	Constant	$T(c)$
Dynamic operation	$\circ(\overline{e})$	Static operation	$\circ\#T(\overline{e})$

Fig. 9. LWG syntax

quasi-transitivity property that relates implementable and assignable types accordingly, which allows us to conclude type preservation for the contextual cases as needed.

Lemma 3.4 (Quasi-transitivity). *Let $\Delta \vdash T, T', S\ \checkmark$. If $T <:_\Delta T'$ and $T' \prec:_\Delta S$ then $T \prec:_\Delta S$.*

Theorem 3.5 (Type Preservation). *If $\Delta; \Gamma \vdash e : T$ and $e \longrightarrow e'$ then $\Delta; \Gamma \vdash e' : U$ for some U such that $\vdash U <:_\Delta T$.*

We say expression e panics if there exists an evaluation context E, value v and type T such that $e = E[v.(T)]$ and $type(v) \not\prec: T$.

Theorem 3.6 (Progress). *If $\emptyset; \emptyset \vdash e : T$ then either e is a value, $e \longrightarrow e'$ for some e', or e panics.*

4 Low-Level Welterweight Go

We propose a *low-level* model of Go that, compared to WG, more closely reflects the runtime mechanisms of actual Go. It introduces an additional kind of value, the *interface value* which we illustrate with the following WG example:

```
type Drawable interface { ... }            type Shape interface { ... } // Assume Shape <:∅ Drawable
type C struct {}                           func (t C) f[T Drawable](x T) T { return x }
type Circle struct {p Point; r int} // Assume Circle ≺:∅ Shape
func main() { _ = C{}.f[Shape](Circle{}) }
```

where we assume that $\texttt{Circle} \prec:_\emptyset \texttt{Shape} <:_\emptyset \texttt{Drawable}$. In LWG when the value $\texttt{Circle}\{\ldots\}$ is passed to function `f` at runtime, it is boxed in an interface value such as:

$$\texttt{Shape@Drawable}(\texttt{Circle}, \rho, \texttt{Circle}\{\texttt{Point}\{1, 1\}, \texttt{int}(2)\})$$

In this interface value, the ghost Shape specifies the interface logical type (which may be an instantiated type parameter), Drawable (another interface) represents the type bound that Shape must satisfy, and `Circle` is the concrete implementation type (i.e., a struct). The pale red shade indicates ghost types, i.e., information that has no operational runtime significance. It is used in the formal type system and compilation only, not in the operational semantics. In the interface value above, the ghost Shape@Drawable specifies the type of the interface, i.e., it is a value of type Shape which has only access to the methods of `Drawable`.

We also introduce a notion of *boxy-ness*. We say a struct value by itself is an *unboxed* value, while an interface value is a box that contains a struct value (above, the nested $\texttt{Circle}\{\ldots\}$). The other elements of an interface value represent RTTI (the non-ghost `Circle`) and the *method table* ρ (mapping abstract methods of `Drawable` to their implementations in `Circle`). For any interface value $T@V(S, \rho, v)$ we always require that $S \prec:_\emptyset T <:_\emptyset V$.

In accordance with actual Go, our low-level model distinguishes two kinds of method calls.

$$\texttt{Circle}\{\ldots\}\#\texttt{Circle.Draw}() \qquad\qquad \texttt{Shape@Drawable}(\texttt{Circle}, \rho, \texttt{Circle}\{\ldots\}).\texttt{Draw}()$$

Above on the left is a *static* method call. The Go compiler generates these when the type of the receiver expression is a *struct* (or named base) type, which above is `Circle`. The `Circle.Draw` part of the static call can be interpreted as a *statically*-compiled jump address for the target method. Above on the right is a *dynamic* method call. Operationally, the receiver must be an *interface* value. The low-level operational semantics dynamically looks up the target method `Draw` in the table ρ.

A key point is that structural subtyping in WG's type system permits, e.g., a *struct* value to be passed as a method call argument for a compatible *interface*-typed parameter—yet the two kinds are *not* operationally interchangeable at the lower-level. Besides the distinct low-level mechanisms for method calls, the two kinds have different representations in memory: the layout of a struct value depends on its field types, while the layout of an interface value is a fixed “fat pointer” structure.

Mediation between the two kinds of values is resolved by the Go compiler. Based on static typing information, it inserts operations for “boxing” of struct values and “box conversion” between interface types. We add these operations as expressions to our low-level language, matching what can be concretely witnessed in the actual Go compiler.

The syntax of LWG (Figure 9) reflects the characteristics discussed above. For simplicity LWG re-uses the syntax of declarations and types from WG (Figure 3) but the syntax of expressions is revised to include: interface values, two types of method calls, and operations for boxing and conversions. Additionally, field names in field accesses are substituted by the index of the field, type assertion uses $T@V$ targets, and type conversions are no longer present. This changes reflects the implementation of actual Go and the assumptions that only interfaces carry RTTI at runtime.

The semantics of LWG is given in Figure 10. We discuss LWG's semantics before its type system because they directly correspond to Go's runtime behaviour, whereas LWG's type system (presented in Section 4.1) exists solely for our meta-theory. We start by giving a high-level overview of the operations for value conversions. Consider the following Go declarations:

```
type Shape interface { Draw() Shape }                func (x Circle) Draw() Shape { return x }
```

Boxing – a.k.a. “Make Type”. Consider the `Draw` method above which returns `x` of type `Circle` while its signature declares `Shape` as return type. Since `Circle` is a structural subtype of `Shape` (`Circle <: Shape`), the method is well-typed in WG. At the low-level, however, values of type `Circle` are of struct kind, while the method return type `Shape` is of interface kind, which have incompatible memory formats. To resolve such cases, the Go compiler inserts a `make` operation. In the compiled program, the return expression of `Draw` becomes: $\texttt{x.(Circle} \rightarrow_\rho \texttt{Shape@Shape)}$. The compiler knows `Shape` and `Circle` statically, hence ρ can be produced at compile-time. At runtime, this `make` operation will box the `Circle` struct value in a `Shape` interface value. Notice that the RTTI `Circle` comes from the `Circle` statically embedded by the compiler.

$$\texttt{Circle}\{\ldots\}.(\texttt{Circle} \rightarrow_\rho \texttt{Shape@Shape}) \quad\longrightarrow\quad \texttt{Shape@Shape}(\texttt{Circle}, \rho, \texttt{Circle}\{\ldots\})$$

Box conversion – a.k.a. “Change Type”. Assume we add another method:

```
func (x Circle) Render() any { return x.Draw() }
```

The return types of `Draw` and `Render` are both of interface kind. However, interface values contain method tables that must *align* with the type of the interface. The method table of a `Shape` interface value contains `Draw`, whereas the method table of `any` is empty, regardless of the boxed struct value (i.e., $\texttt{Circle}\{\ldots\}$ for both). To resolve such cases, the Go compiler inserts a *Change-Type* operation. The return expression of `Render` becomes $\texttt{x.Draw().(Shape} \rightarrow \texttt{any@any)}$.

At runtime, this operation will *re*-box the `Circle` struct value (that was boxed by the `Shape` interface value returned by `Draw`) in an `any` interface value. To do so, it *dynamically* creates the

Values	$w ::= T\{\overline{v}\} \mid T(c)$	$v ::= w \mid \Phi(S, \rho, w)$	
Evaluation context	$E ::=$		
Hole	$\square$	Structure	$T\{\overline{v}, E, \overline{e}\}$
Dyn Operator	$\circ(\overline{v}, E, \overline{e})$	Select	$E.i$
Operator	$\circ\#T(\overline{v}, E, \overline{e})$	Type assertion	$E.(T@V)$
Dyn Method call receiver	$E.m[\overline{T}](\overline{e})$	Make type	$E.(S \to_\rho \Phi)$
Dyn Method call arguments	$v'.m[\overline{T}](\overline{v}, E, \overline{e})$	Change type	$E.(S \to T@V)$
Method call receiver	$E\#T.m[\overline{T}](\overline{e})$	Static change	$E.(S \twoheadrightarrow T)$
Method call arguments	$v'\#T.m[\overline{T}](\overline{v}, E, \overline{e})$		

Auxiliary functions

$$\dfrac{(\mathbf{func}\ (x\ t[\overline{\alpha\ S}])\ m[\overline{\beta\ T}](\overline{y\ \Phi})\ \Psi\ \{\mathbf{return}\ e\}) \in \overline{D}}{\mathit{body}_L(t.m) = [\overline{\alpha};\ \overline{\beta}](x, \overline{y}).e} \qquad \dfrac{L <:_\Delta T}{\mathsf{mkTable}_\Delta(L, T) = \emptyset}$$

$$\dfrac{t[\overline{R}] <:_\Delta T \qquad O = \{\circ \mid \{\mathit{under}_\Delta(V) \mid V \in \mathit{types}_\Delta(T)\} \subseteq \mathit{dom}(\circ)\}}{\mathsf{mkTable}_\Delta(t[\overline{R}], T) = [m \mapsto t.m \mid mM \in \mathit{methods}_\Delta(T)] \cup [\circ \mapsto \circ_B \mid \circ \in O, B = \mathit{under}(t[\overline{R}])]}$$

Reductions $\boxed{e \longrightarrow e'}$

$$\textsc{Select}\ \dfrac{}{S\{\overline{v}\}.i \longrightarrow v_i} \qquad \textsc{Make}\ \dfrac{}{w.(S \to_\rho T@U) \longrightarrow T@U(S, \rho, w)} \qquad \textsc{Change-Type}\ \dfrac{\rho' = \mathsf{mkTable}_\emptyset(S, U)}{\Phi(S, \rho, w).(T \to T'@U) \longrightarrow T'@U(S, \rho', w)}$$

$$\textsc{Static-Change-S}\ \dfrac{}{S\{\overline{v}\}.(S \twoheadrightarrow T) \longrightarrow T\{\overline{v}\}} \qquad \textsc{Static-Change-C}\ \dfrac{}{S(c).(S \twoheadrightarrow T) \longrightarrow T(c)} \qquad \textsc{Assert-Ok}_S\ \dfrac{}{\Phi(S, \rho, w).(S@S) \longrightarrow w}$$

$$\textsc{Assert-Ok}_I\ \dfrac{\mathit{iface}(V) \quad S <:_\emptyset U \quad \rho = \mathsf{mkTable}_\emptyset(S, V)}{\Phi(S, \rho', w).(U@V) \longrightarrow U@V(S, \rho, w)} \qquad \textsc{Call-Dyn}\ \dfrac{U = t[\overline{R}] \quad [\overline{\alpha};\ \overline{\beta}](x, \overline{y}).e = \mathit{body}_L(\rho(m))}{\Phi(U, \rho, v).m[\overline{S}](\overline{v}) \longrightarrow e[\overline{\alpha := R}][\overline{\beta := S}][x := v][\overline{y := v}]}$$

$$\textsc{Call-Static}\ \dfrac{[\overline{\alpha};\ \overline{\beta}](x, \overline{y}).e = \mathit{body}_L(t.m)}{w\#t[\overline{R}].m[\overline{S}](\overline{v}) \longrightarrow e[\overline{\alpha := R}][\overline{\beta := S}][x := w][\overline{y := v}]} \qquad \textsc{Op-Dyn}\ \dfrac{\forall i.\, v_i = \Phi(S, \rho, v'_i) \qquad \delta(\rho(\circ), \overline{v'}) = v''}{\circ(\overline{v}) \longrightarrow \Phi(S, \rho, v'')}$$

$$\textsc{Op-Static}\ \dfrac{\mathit{under}_\emptyset(T) = B \qquad \delta(\circ_B, \overline{v}) = v'}{\circ\#T(\overline{v}) \longrightarrow v'} \qquad \textsc{Context}\ \dfrac{e \longrightarrow d}{E[e] \longrightarrow E[d]}$$

Fig. 10. LWG reduction

new method table (cf. Make) based on the *RTTI* `Circle` carried by the old interface value and the *new interface type* `any` (embedded by the compiler)—in this case, yielding the empty method table.

$$\mathtt{Shape}@\mathtt{Shape}(\mathtt{Circle}, \rho, \mathtt{Circle}\{\ldots\}).(\mathtt{Shape} \to \mathtt{any}@\mathtt{any}) \quad \longrightarrow \quad \mathtt{any}@\mathtt{any}(\mathtt{Circle}, \emptyset, \mathtt{Circle}\{\ldots\})$$

Static cast – a.k.a. “Static change”. Assume we add another method:

```
func (x Circle) asStruct() struct{p Point; r int} { return x }
```

The “Static-Change” operation performs a “conversion” between assignable non-interface types. This operation does not perform any actions related to boxing such as RTTI or method tables (in Go it supports reflection which we are not modelling here). The return expression of `asStruct` becomes: $\mathtt{x}.(\mathtt{Circle} \twoheadrightarrow \mathbf{struct}\ \{p\ \mathtt{Point}, r\ \mathtt{int}\})$.

Since it is converting between struct types, the compiler statically knows both the old and the new types anyway. Static-change is necessary to obtain a type safety result for LWG.

$$\texttt{Circle}\{\texttt{Point}\{1,1\},\texttt{int}(2)\}.(\texttt{Circle} \twoheadrightarrow \textbf{struct}\ \{p\ \texttt{Point}, r\ \texttt{int}\}) \longrightarrow \textbf{struct}\ \{p\ \texttt{Point}, r\ \texttt{int}\}\{\texttt{Point}\{1,1\},\texttt{int}(2)\}$$

We review Figure 10 in more detail. Values in LWG consist of structure literals $T\{\overline{v}\}$, typed constants $T(c)$, and interface value $\Phi(S,\rho,w)$. The reduction rules rely on context E, similar to WG. Function $\mathsf{mkTable}_\Delta(S,T)$ returns a table mapping method names specified in interface T to those implemented in the type named t when $S = t[\overline{R}]$, as well as operators defined on all types listed in T to those defined on t ($\circ_t$ denote the definition of $\circ$ for t). When S is an anonymous type, then the table is empty. Function $body_L(t.m)$ returns the body of method m defined for type t as well relevant bindings for parameters. Note that the type arguments $\overline{T}$ passed to generic types ($t[\overline{T}]$) and methods ($m[\overline{T}](\overline{e})$) correspond to the dictionaries used by the Go runtime (cf. Section 2). For readability, we syntactically separate type parameters $\overline{\alpha}$; $\overline{\beta}$ from value parameters $x, \overline{y}$, but both are treated uniformly as substitutable variables in the operational semantics.

Following the discussion above, the reduction rules are straightforward. Rule SELECT extract a field from a struct, using the field index i. Rules MAKE, CHANGE-TYPE, and STATIC-CHANGE-S, STATIC-CHANGE-C formalise the behaviour of boxing and type conversion operations discussed above. Observe that ASSERT-OK$_S$ allows an interface value to be "unboxed". Rule CALL-DYN and CALL-STATIC are for dynamic and static method calls, respectively. In the dynamic case, we first have to look-up the "address" of method m in the method table ρ; while we can jump directly to $t.m$ in the static case. In both cases, formal (type) parameters are substituted by the (type) arguments. The rules for operators work similarly. In the dynamic case, we look-up the implementation of $\circ$ for type S via the table ρ; in the static case we can perform the concrete operation directly.

4.1 Typing in LWG

The typing rules for LWG are presented in Figures 11 and 12. A key feature of the LWG type system are types of the form $T@V$, where T represents the logical type and V tracks the available methods and the runtime representation constraint. Our compilation (Section 5) uses these annotated types to determine boxing status. Indeed type variables denote boxed values, but after instantiation this information is lost—a type variable instantiated with a base type loses its boxing annotation. Without this information, the compiler cannot determine when (un)boxing is required.

When $T = V$, we have standard Go types that are either consistently boxed (interfaces) or unboxed (structs/base types). When $T \neq V$, then T is a type parameter α or a concrete instantiation that must be boxed and satisfy interface V. For example, int@any represents an integer value that has been boxed into an any interface. As a general rule, if $\neg iface(V)$ then $T = V$. The V component determines both the runtime representation ("boxy-ness") and the available method set for dynamic dispatch, while T preserves the precise information needed for static typing. This dual tracking is essential because Go's structural subtyping allows the same logical operation to require different runtime representations depending on context: a method call on a struct receiver expects unboxed arguments, while a matching method signature in a generic interface may expect boxed arguments, yet we cannot always determine statically which will be needed.

The well-formedness checks for LWG are similar to those of WG. We do not repeat the rules for type declarations as they are the same as for WG. Judgement $\Delta \vdash T@V \checkmark_b$ holds when T is a basic interface (i.e., not a union) and V is a well-formed type. An LWG method declaration is well-formed if it is defined on a concrete type (t) and its body e has type Ψ, matching the method signature, with the appropriate context. Note how x has type $t[\overline{\alpha}]@t[\overline{\alpha}]$.

Method calls are treated similarly to WG but deal with the two forms of method calls and subtyping is not used between actual arguments and declared types. Rule T-STATIC-CALL applies

Well-formed types and declarations $\boxed{\Delta \vdash \Phi \checkmark_{\mathrm{b}}}$

T@V-B

$$\frac{\Delta \vdash T \checkmark_{\mathrm{b}} \qquad \Delta \vdash V \checkmark \qquad \neg \mathit{tyvar}_{\Delta}(V)}{\Delta \vdash T@V \checkmark_{\mathrm{b}}}$$

T-FUNC

$$\frac{\begin{array}{c} \neg \mathit{iface}(t[\overline{\alpha}]) \qquad \mathit{distinct}(x, \overline{y}) \qquad (\mathbf{type}\ t[\overline{\alpha\ R}]\ T) \in \overline{D} \qquad \emptyset \vdash \overline{\alpha\ R} \checkmark \\ \overline{\alpha : R} \vdash \overline{\beta\ S} \checkmark \qquad \Delta = \overline{\alpha : R}, \overline{\beta : S} \qquad \Delta \vdash \overline{\Phi} \checkmark_{\mathrm{b}} \qquad \Delta \vdash \Psi \checkmark_{\mathrm{b}} \qquad \Delta;\ x : t[\overline{\alpha}]@t[\overline{\alpha}], \overline{y : \Phi} \vdash e : \Psi \end{array}}{\mathbf{func}\ (x\ t[\overline{\alpha\ R}])\ m[\overline{\beta\ S}](\overline{y\ \Phi})\ \Psi\ \{\mathbf{return}\ e\} \checkmark}$$

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

T-DYN-CALL

$$\frac{\begin{array}{c} \Delta \vdash \overline{S} \checkmark_{\mathrm{b}} \qquad \Delta; \Gamma \vdash e : R@V \\ \mathit{iface}(V) \qquad (m[\overline{\beta\ S'}](\overline{y\ \Phi})\ \Psi) \in \mathit{methods}_{\Delta}(V) \qquad \eta = (\overline{\beta := S}) \qquad \Delta; \Gamma \vdash \overline{e : \Phi[\eta]} \qquad \overline{\beta[\eta] <:_{\Delta} S'[\eta]} \end{array}}{\Delta; \Gamma \vdash e.m[\overline{S}](\overline{e}) : \Psi[\eta]}$$

T-STATIC-CALL

$$\frac{\begin{array}{c} \Delta \vdash \overline{S} \checkmark_{\mathrm{b}} \qquad \Delta; \Gamma \vdash e : R@R \\ \neg\mathit{iface}(R) \qquad (m[\overline{\beta\ S'}](\overline{y\ \Phi})\ \Psi) \in \mathit{methods}_{\Delta}(R) \qquad \eta = (\overline{\beta := S}) \qquad \Delta; \Gamma \vdash \overline{e : \Phi[\eta]} \qquad \overline{\beta[\eta] <:_{\Delta} S'[\eta]} \end{array}}{\Delta; \Gamma \vdash e\#R.m[\overline{S}](\overline{e}) : \Psi[\eta]}$$

T-INTERFACE

$$\frac{\begin{array}{c} \emptyset \vdash T@V \checkmark_{\mathrm{b}} \qquad \neg\mathit{iface}(S) \\ \mathit{iface}(V) \qquad \neg\mathit{iface}(T) \Rightarrow S = T \qquad \emptyset; \Gamma \vdash e : S@S \qquad S \prec:_{\emptyset} T <:_{\emptyset} V \qquad \rho = \mathsf{mkTable}_{\emptyset}(S, V) \end{array}}{\emptyset; \Gamma \vdash T@V(S, \rho, e) : T@V}$$

T-VAR

$$\frac{(x : \Phi) \in \Gamma}{\Delta; \Gamma \vdash x : \Phi}$$

T-LITERAL

$$\frac{\Delta \vdash T \checkmark_{\mathrm{b}} \qquad \mathit{struct}(T) \qquad \Delta; \Gamma \vdash \overline{e : \Phi} \qquad (\overline{\_\ \Phi}) = \mathit{fields}_{\Delta}(T)}{\Delta; \Gamma \vdash T\{\overline{e}\} : T@T}$$

T-FIELD

$$\frac{\mathit{struct}(T) \qquad \Delta; \Gamma \vdash e : T@T \qquad (\overline{\_\ S@V}) = \mathit{fields}_{\Delta}(T)}{\Delta; \Gamma \vdash e.i : S_i@V_i}$$

Fig. 11. LWG: Typing for declarations and expressions

when the receiver expression has type $R@R$ with R not an interface. Rule T-DYN-CALL applies when the receiver has type $R@V$ with V an interface. The method must be in V's method table.

Rule T-INTERFACE ensures the boxed expression e is a concrete struct or base type, that S is assignable to the interface type T, and that T implements the bound V. The method table ρ must map the methods of the bound V to their concrete implementations in S. If the value is boxing into a concrete type, then the RTTI must match the logical type ($S = T$).

Rules T-VAR, T-LITERAL, and T-FIELD are straightforward. Note that, a struct type T is typed with $T@T$ which implies an unboxed value since $\neg\mathit{iface}(T)$.

We comment on the rules for boxing and conversion, see Figure 12. Make, $e.(T \rightarrow_{\rho} U@V)$, creates an interface value by boxing a concrete value. Hence Rule T-MAKE applies only when converting from a non-interface type to an interface type. Change-type, $e.(T \rightarrow U@V')$, converts between different interface types by updating the method table while preserving the boxed value, hence Rule T-CHANGE applies only when both source and target are interface types. Note that the premises of these rules match those of Rule T-INTERFACE as such terms reduce to interface

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

T-MAKE

$$\dfrac{\begin{array}{c}\Delta \vdash T \checkmark_{\mathrm{b}} \qquad \Delta \vdash U@V \checkmark_{\mathrm{b}} \qquad \neg \mathit{iface}(T) \\ \neg \mathit{iface}(U) \Rightarrow (T = U) \qquad \mathit{iface}(V) \qquad \Delta; \Gamma \vdash e : T@T \qquad T \prec:_{\Delta} U <:_{\Delta} V \qquad \rho = \mathsf{mkTable}_{\Delta}(T, V)\end{array}}{\Delta; \Gamma \vdash e.(T \rightarrow_{\rho} U@V) : U@V}$$

T-CHANGE

$$\dfrac{\begin{array}{c}\Delta \vdash T@V \checkmark_{\mathrm{b}} \qquad \Delta \vdash U@V' \checkmark_{\mathrm{b}} \\ \mathit{iface}(V) \qquad \neg \mathit{iface}(U) \Rightarrow T = U \qquad \mathit{iface}(V') \qquad \Delta; \Gamma \vdash e : T@V \qquad T \prec:_{\Delta} U <:_{\Delta} V'\end{array}}{\Delta; \Gamma \vdash e.(T \rightarrow U@V') : U@V'}$$

T-STATIC-CHANGE

$$\dfrac{\Delta \vdash T \checkmark_{\mathrm{b}} \qquad \Delta \vdash U \checkmark_{\mathrm{b}} \qquad \neg \mathit{iface}(T) \qquad \neg \mathit{iface}(U) \qquad \Delta; \Gamma \vdash e : T@T \qquad \mathit{under}_{\Delta}(U) = \mathit{under}_{\Delta}(T)}{\Delta; \Gamma \vdash e.(T \twoheadrightarrow U) : U@U}$$

T-ASSERT$_I$

$$\dfrac{\mathit{iface}(T) \qquad \Delta \vdash T@U \checkmark_{\mathrm{b}} \qquad \Delta; \Gamma \vdash e : S@V \qquad \mathit{iface}(V) \qquad \mathit{iface}(U) \qquad T \prec:_{\Delta} U}{\Delta; \Gamma \vdash e.(T@U) : T@U}$$

T-ASSERT$_S$

$$\dfrac{\neg \mathit{iface}(T) \qquad \Delta \vdash T \checkmark_{\mathrm{b}} \qquad \Delta; \Gamma \vdash e : S@V \qquad \mathit{iface}(V) \qquad T <:_{\Delta} V}{\Delta; \Gamma \vdash e.(T@T) : T@T}$$

$$\boxed{\begin{array}{l}\text{T-STUPID} \\ \dfrac{\neg \mathit{iface}(T) \qquad \Delta \vdash T@U \checkmark_{\mathrm{b}} \qquad \Delta; \Gamma \vdash e : S@V \qquad T \not\prec:_{\Delta} V \qquad T \prec:_{\Delta} U}{\Delta; \Gamma \vdash e.(T@U) : T@U}\end{array}}$$

T-CONST

$$\dfrac{\Delta \vdash T \checkmark_{\mathrm{b}} \qquad \mathit{under}_{\Delta}(T) = \mathit{ctype}(c)}{\Delta; \Gamma \vdash T(c) : T@T}$$

T-OP-STATIC

$$\dfrac{\Delta; \Gamma \vdash \overline{e} : T@T \qquad \mathit{under}_{\Delta}(T) = B \qquad B \in \mathit{dom}(\circ)}{\Delta; \Gamma \vdash \circ\#T(\overline{e}) : T@T}$$

T-OP-DYN

$$\dfrac{\Delta; \Gamma \vdash \overline{e} : \alpha@T \qquad (\alpha : T) \in \Delta \qquad \{\mathit{under}_{\Delta}(V) \mid V \in \mathit{types}(T)\} \subseteq \mathit{dom}(\circ)}{\Delta; \Gamma \vdash \circ(\overline{e}) : \alpha@T}$$

T-OP-RUNTIME

$$\dfrac{\Delta; \Gamma \vdash \overline{e} : T@U \qquad \mathit{under}_{\Delta}(T) = B \qquad B \in \{\mathit{under}_{\Delta}(V) \mid V \in \mathit{types}(T)\} \subseteq \mathit{dom}(\circ)}{\Delta; \Gamma \vdash \circ(\overline{e}) : T@U}$$

Fig. 12. LWG: additional typing rules

values. Static-change, $e.(T \twoheadrightarrow U)$, performs conversions between struct or base types without any boxing operations, hence rule T-STATIC-CHANGE applies only when both T and U are concrete type. Type assertions and constants (Rules T-ASSERT$_I$ and T-ASSERT$_S$, and T-CONST) are straightforward. We write $\mathit{ctype}(c)$ for the type of constant c, e.g., $\mathit{ctype}(42) = \textbf{int}$. Rule T-STUPID is a standard rule [Griesemer et al. 2020; Igarashi et al. 2001] to avoid that expressions become ill-typed during reduction. It is used for runtime terms only.

The last three rules deal with static and dynamic form of operators. Rule T-OP-STATIC handles operators on concrete types that are resolved statically. In this case, the operator must be available on the underlying type of T. Rule T-OP-DYN handles operators for type parameters, in which case the operator must be available for all underlying types of types listed in α's type bound (T). Rule

Compilation: types $\boxed{\langle T \rangle}$ $\boxed{mM \mapsto mM'}$ $\boxed{\Phi \mapsto \Phi'}$ $\boxed{C \mapsto C}$ $\boxed{T \mapsto T'}$ $\boxed{\lfloor T \rfloor}$ $\boxed{\lfloor \Phi \rfloor}$

$$\frac{\mathit{iface}(\Phi\lfloor_2)}{\langle \Phi \rangle = \Phi} \qquad \frac{\neg \mathit{iface}(T)}{\langle T@T \rangle = T@\mathsf{Any}}$$

$$\textsc{d-sig}\ \frac{\overline{S \mapsto S'} \qquad \overline{\Phi \mapsto \Phi'} \qquad \Psi \mapsto \Psi'}{m[\overline{\beta\ S}](\overline{y\ \Phi})\ \Psi \mapsto m_D[\overline{\beta\ S'}](\overline{y\ \langle \Phi' \rangle})\ \langle \Psi' \rangle} \qquad \textsc{d-type-b}\ \frac{}{B \mapsto B}$$

$$\textsc{d-tvar}\ \frac{}{\alpha \mapsto \alpha} \qquad \textsc{d-type-named}\ \frac{\overline{S \mapsto S'}}{t[\overline{S}] \mapsto t[\overline{S}']}$$

$$\textsc{d-type-struct}\ \frac{\overline{\Phi \mapsto \Phi'}}{\textbf{struct}\ \{\overline{f\ \Phi}\} \mapsto \textbf{struct}\ \{\overline{f\ \Phi'}\}} \qquad \textsc{d-type-iface}\ \frac{\overline{F \mapsto F'}}{\textbf{interface}\ \{\overline{F}\} \mapsto \textbf{interface}\ \{\overline{F'}\}}$$

$$\textsc{d-tilde}\ \frac{T \mapsto T'}{{\sim}T \mapsto {\sim}T'} \qquad \textsc{d-or}\ \frac{C \mapsto C' \qquad E \mapsto E'}{C|E \mapsto C'|E'} \qquad \textsc{d-pair}\ \frac{T \mapsto S \qquad V \mapsto U}{T@V \mapsto S@U}$$

$$\frac{T \mapsto S}{\lfloor T \rfloor = S} \qquad \frac{T \mapsto S \qquad V \mapsto U}{\lfloor T@V \rfloor = S@U}$$

Fig. 13. Auxiliary notations and compilation rules for types

t-op-runtime is necessary to handle runtime terms where a generic parameter bounded by V has been instantiated by a type T whose underlying type is a base type.

4.2 Metatheory of LWG

We prove LWG's type safety through progress and preservation arguments. Unlike WG, type preservation does not require the implements relation because interface and struct values are distinct at runtime, and typing requires exact type matches rather than assignability. Though this seems restrictive, our compilation procedure (Section 5) shows that all well-typed WG programs can be compiled to LWG using suitable make and change type primitives.

Theorem 4.1 (Type Preservation). *If* $\Delta; \Gamma \vdash e : T$ *and* $e \longrightarrow e'$ *then* $\Delta; \Gamma \vdash e' : T$.

We define panics as in WG, noting that progress implies that only type assertions may effectively fail at runtime, not make or change type.

Theorem 4.2 (Progress). *If* $\emptyset; \emptyset \vdash e : T$ *then either* e *is a value,* $e \longrightarrow e'$ *for some* e'*, or* e *panics.*

5 Compilation

Monomorphisation tackles the problem of data layouts for generic code by generating all possibly needed specialisations, which typically requires a whole program analysis. An alternative approach more compatible with separate compilation is a uniform (or *boxed*) representation using pointer indirection, which requires runtime boxing/unboxing actions.

Our compilation strategy is based on the latter approach. The key points and challenges are how we (i) reuse Go's pre-existing runtime infrastructure for interfaces to perform the type-directed static compilation and the runtime boxing, and (ii) achieve a uniform compilation strategy given Go's structural typing, as we discuss in Section 5.2.

5.1 Repurposing Go's Runtime Infrastructure

We repurpose interface values as boxes that represent values of *generic* type, with boxing performed by LWG's make and change type operations. This is natural since interface values are already boxed in the Go runtime, and generic values have interface-like status with bounds that constrain their operations. Unboxing is performed by type assertions. We present our compilation strategy as a translation from well-typed WG programs to LWG programs using a $\mapsto$ arrow.

Figure 13 gives the compilation of types to prepare the program with uniform handling of dynamic methods. Each method m occurring in interfaces is renamed to m_D (D for dynamic) and its

Compilation: synthetic casts $\boxed{\Delta; \Gamma \vdash e : U \mapsto_{\Phi} e'}$

MAKE-IFACE

$$\frac{\mathit{iface}(V) \qquad \neg\mathit{iface}(S) \qquad \mathit{iface}(T) \qquad \Delta; \Gamma \vdash e \mapsto e' \qquad \rho = \mathsf{mkTable}_{\Delta}(\lfloor S \rfloor, \lfloor V \rfloor)}{\Delta; \Gamma \vdash e : S \mapsto_{T@V} e'.(\lfloor S \rfloor \to_{\rho} \lfloor T@V \rfloor)}$$

MAKE-BS

$$\frac{\mathit{iface}(V) \qquad \neg\mathit{iface}(S) \qquad \neg\mathit{iface}(T) \qquad \Delta; \Gamma \vdash e \mapsto e' \qquad \rho = \mathsf{mkTable}_{\Delta}(\lfloor T \rfloor, \lfloor V \rfloor)}{\Delta; \Gamma \vdash e : S \mapsto_{T@V} e'.(\lfloor S \rfloor \twoheadrightarrow \lfloor T \rfloor).(\lfloor T \rfloor \to_{\rho} \lfloor T@V \rfloor)}$$

CHANGE

$$\frac{\mathit{iface}(U) \qquad \mathit{iface}(V) \qquad \Delta; \Gamma \vdash e \mapsto e'}{\Delta; \Gamma \vdash e : U \mapsto_{T@V} e'.(\lfloor U \rfloor \to \lfloor T@V \rfloor)}$$

STATIC-CHANGE

$$\frac{\neg\mathit{iface}(U) \qquad \neg\mathit{iface}(T) \qquad \Delta; \Gamma \vdash e \mapsto e'}{\Delta; \Gamma \vdash e : U \mapsto_{T@T} e'.(\lfloor U \rfloor \twoheadrightarrow \lfloor T \rfloor)}$$

Fig. 14. Compilation rules: generate synthetic casts

signature is adapted so that all argument and return types are boxed (using $\langle T \rangle$). The rules ensure these changes are made consistently, including in literal interface types in struct definitions. We introduce $\lfloor T \rfloor$ (resp. $\lfloor \Phi \rfloor$) as a shorthand for a compiled type.

To compile expressions and method declarations, we rely on an auxiliary judgement given in Figure 14. In $\Delta; \Gamma \vdash e : U \mapsto_{\Phi} e'$, e is a well-typed WG expression, e' is an LWG expression that e compiles into, and Φ is the target LWG type of e'. This judgement determines which synthetic casts (make, change or static-change) are needed in the resulting expression. Interface values are created in two situations. Rule MAKE-IFACE handles non-interface expressions used at interface types (standard Go interface creation), computing method table ρ with methods of S listed in interface V. Rule MAKE-BS handles generic value boxing: non-interface source type S with non-interface target type T but interface bound V, creating an interface value with method table from T for V.

For example, consider the WG declaration **type** `Cell[T` **any**`]` **struct**`{ val T@any }`. Initialising `Cell[`**int**`]{`**int**`(42)}` requires boxing the integer for uniform representation. The integer field compiles to `int`(42).(`int` $\to_{\rho}$ $\lfloor$`int@any`$\rfloor$) via Rule MAKE-BS. Field access generates unboxing via Rule SELECT-BOXED and the bound any allows us to identify a boxed **int**.

Rule CHANGE handles interface type U to interface type V conversion via Rule CHANGE-TYPE (Figure 10). Unlike MAKE, the method table must be computed at runtime when the runtime type becomes known. This showcases reuse of Go's interface infrastructure: existing interface values instantiating type parameters are adapted to the interface specified by the type parameter bound.

Rule STATIC-CHANGE applies when neither source type U nor target bound T are interfaces. Such transformations convert between assignable WG types, similar to type conversions in WG's operational semantics. Consider **type** `StringerCell` **struct**`{ val Stringer@Stringer }`. Assume **int** $<:_{\emptyset}$ `Stringer`, in `StringerCell{`**int**`(42)}`, the field **int**`(42)` compiles to `int`(42).(`int` $\to_{\rho}$ $\lfloor$`Stringer@Stringer`$\rfloor$) by rule MAKE-IFACE. Rules BOX and NBOX are variants to ensure all values are boxed, this is used in dynamic method calls and matches the $\langle \Phi \rangle$ function introduced earlier.

The compilation of expressions is formalised in Figures 15 and 16. In judgement $\Delta; \Gamma \vdash e \mapsto e'$, e is a well-typed WG expression (under suitable contexts Δ and Γ) and e' is an LWG expression that e compiles into. Rules VAR, CONST, and ASSERT are straightforward. Rule CONVERSION essentially replaces a WG static conversion with the appropriate synthetic cast.

The compilation of field selection is given by Rules SELECT-BOXED and SELECT-UNBOXED, the selected rule depends on whether the struct field needs to be unboxed or not. For Rule SELECT-BOXED, the type of field f_i is $S_i@V_i$, where S_i is not a type variable and V_i is an interface. This means that either the field denotes a generic type that has been instantiated or it is a value of interface type

Compilation: expressions $\boxed{\Delta; \Gamma \vdash e \mapsto e'}$

Var

$$\frac{}{\Delta; \Gamma \vdash x \mapsto x}$$

Const

$$\frac{}{\Delta; \Gamma \vdash T(c) \mapsto T(c)}$$

Assert

$$\frac{\Delta; \Gamma \vdash e \mapsto e'}{\Delta; \Gamma \vdash e.(T) \mapsto e'.(\lfloor T \rfloor @ \lfloor \mathit{bounds}_\Delta(T) \rfloor)}$$

Conversion

$$\frac{\Delta; \Gamma \vdash e : S \qquad \Delta; \Gamma \vdash e : S \mapsto_{T @ \mathit{bounds}_\Delta(T)} e'}{\Gamma \vdash T(e) \mapsto e'}$$

Struct

$$\frac{\overline{\_\;\Phi} = \mathit{fields}_\Delta(T) \qquad \Delta; \Gamma \vdash \overline{e : U} \qquad \overline{\Delta; \Gamma \vdash e : U \mapsto_\Phi e'}}{\Delta; \Gamma \vdash T\{\overline{e}\} \mapsto \lfloor T \rfloor \{\overline{e'}\}}$$

Select-boxed

$$\frac{\Delta; \Gamma \vdash e \mapsto e' \qquad \Delta; \Gamma \vdash e : T \qquad \overline{f\ S @ V} = \mathit{fields}_\Delta(T) \qquad \neg \mathit{tyvar}_\Delta(S_i) \qquad \mathit{iface}(V_i) \qquad f_i = f}{\Delta; \Gamma \vdash e.f \mapsto e'.i.(\lfloor S_i \rfloor @ \lfloor S_i \rfloor)}$$

Select-unboxed

$$\frac{\Delta; \Gamma \vdash e \mapsto e' \qquad \Delta; \Gamma \vdash e : T \qquad \overline{f\ S @ V} = \mathit{fields}_\Delta(T) \qquad (\mathit{tyvar}_\Delta(S_i) \vee \neg \mathit{iface}(V_i)) \qquad f_i = f}{\Delta; \Gamma \vdash e.f \mapsto e'.i}$$

op-static

$$\frac{\Delta; \Gamma \vdash \overline{e} : T \qquad \neg \mathit{tyvar}_\Delta(T) \qquad \overline{\Delta; \Gamma \vdash e \mapsto e'}}{\Delta; \Gamma \vdash \circ(\overline{e}) \mapsto \circ \# \lfloor T \rfloor (\overline{e'})}$$

op-dyn

$$\frac{\Delta; \Gamma \vdash \overline{e} : \alpha \qquad \overline{\Delta; \Gamma \vdash e \mapsto e'}}{\Delta; \Gamma \vdash \circ(\overline{e}) \mapsto \circ(\overline{e'})}$$

Fig. 15. Compilation rules: expressions (part 1)

(in which case $S_i = V_i$). In the former case the value will be boxed at runtime but is treated in the remaining source WG program as if it were a plain value of type S_i. In both cases we insert a type assertion to $\lfloor S_i \rfloor$ which will unbox the value in the former case and leave it unchanged (as needed) in the latter. Rule Select-unboxed applies when S_i is a type variable or V_i is not an interface type. No unboxing is needed: if S_i is a type variable, the field must remain boxed to use operations from its bound; if V_i is not an interface, the field is already unboxed.

Rules op-static and op-dyn deal with operators. Recall that the WG type system enforces that the operands have all the same type, which is either a type whose underlying type is a base type, or a type variable whose bound supports $\circ$. The first rule applies when the type of the operand is not a type variable, hence it must be a static operator invocation. If the type of the operand is a type variable, then it is a dynamic invocation and the second rule applies.

5.2 Boxing and Structural Typing

We now move on to the compilation of methods and method calls. It turns out that Go's structural typing makes the treatment of methods especially subtle as we illustrate below.

Example 5.1. *Consider the following WG code:*

```
type Processor[T any] interface { process(input T) T }
type IntProcessor struct {} // IntProcessor <:∅ Processor[int]
func (p IntProcessor) process(input int) int { return input }
type GenericProcessor[T any] struct {} // GenericProcessor[T] <:{T:any} Processor[T]
func (p GenericProcessor[T any]) process(input T) T { return input }
type Client struct {}
func (c Client) useProcessor(processor Processor[int]) int { return processor.process(int(42)) }
```

The code above defines a generic interface `Processor[T]` with a method `process`. We then define a non-generic struct type `IntProcessor` with a `process` method from **`int`** to **`int`**. Notably, `IntProcessor`

Compilation: boxing casts $\boxed{\Delta; \Gamma \vdash d : U \Mapsto_{\Phi} d'}$

$$\textsc{box}\quad \frac{\neg \mathit{iface}(V) \qquad \Delta; \Gamma \vdash d : U \mapsto_{T@\mathsf{Any}} d'}{\Delta; \Gamma \vdash d : U \Mapsto_{T@V} d'} \qquad\qquad \textsc{nbox}\quad \frac{\mathit{iface}(V) \qquad \Delta; \Gamma \vdash d : U \mapsto_{T@V} d'}{\Delta; \Gamma \vdash d : U \Mapsto_{T@V} d'}$$

Compilation: method calls $\boxed{\Delta; \Gamma \vdash e \mapsto e'}$

$$\textsc{Call}_S\textsc{-boxed}\quad \frac{\begin{array}{c}\Delta; \Gamma \vdash e : R \qquad \neg \mathit{iface}(R) \qquad m[\overline{\beta\ S''}](\overline{y\ \Phi})\ T@V \in \mathit{methods}_\Delta(R) \qquad \neg \mathit{tyvar}_\Delta(T[\eta]) \\ \mathit{iface}(V[\eta]) \qquad \eta = (\overline{\beta := S}) \qquad \Delta; \Gamma \vdash e \mapsto e' \qquad \Delta; \Gamma \vdash \overline{d : U} \qquad \Delta; \Gamma \vdash \overline{d : U \mapsto_{\Phi[\eta]} d'}\end{array}}{\Delta; \Gamma \vdash e.m[\overline{S}](\overline{d}) \mapsto e'\#\lfloor R \rfloor.m[\overline{\lfloor S \rfloor}](\overline{d'}).(\lfloor T[\eta] \rfloor @ \lfloor T[\eta] \rfloor)}$$

$$\textsc{Call}_S\textsc{-unboxed}\quad \frac{\begin{array}{c}\Delta; \Gamma \vdash e : R \qquad \neg \mathit{iface}(R) \qquad m[\overline{\beta\ S''}](\overline{y\ \Phi})\ T@V \in \mathit{methods}_\Delta(R) \qquad \eta = (\overline{\beta := S}) \\ (\neg \mathit{iface}(V[\eta]) \vee \mathit{tyvar}_\Delta(T[\eta])) \qquad \Delta; \Gamma \vdash e \mapsto e' \qquad \Delta; \Gamma \vdash \overline{d : U} \qquad \Delta; \Gamma \vdash \overline{d : U \mapsto_{\Phi[\eta]} d'}\end{array}}{\Delta; \Gamma \vdash e.m[\overline{S}](\overline{d}) \mapsto e'\#\lfloor R \rfloor.m[\overline{\lfloor S \rfloor}](\overline{d'})}$$

$$\textsc{Call}_I\quad \frac{\begin{array}{c}\Delta; \Gamma \vdash e : R \qquad \mathit{iface}(R) \qquad m[\overline{\beta\ S''}](\overline{y\ \Phi})\ T@V \in \mathit{methods}_\Delta(R) \qquad \neg \mathit{tyvar}_\Delta(T[\eta]) \\ \eta = (\overline{\beta := S}) \qquad \Delta; \Gamma \vdash e \mapsto e' \qquad \Delta; \Gamma \vdash \overline{d : U} \qquad \Delta; \Gamma \vdash \overline{d : U \Mapsto_{\Phi[\eta]} d'}\end{array}}{\Delta; \Gamma \vdash e.m[\overline{S}](\overline{d}) \mapsto e'.m_D[\overline{\lfloor S \rfloor}](\overline{d'}).(\lfloor T[\eta] \rfloor @ \lfloor T[\eta] \rfloor)}$$

$$\textsc{Call}_{I\alpha}\quad \frac{\begin{array}{c}\Delta; \Gamma \vdash e : R \qquad \mathit{iface}(R) \qquad m[\overline{\beta\ S''}](\overline{y\ \Phi})\ T@V \in \mathit{methods}_\Delta(R) \qquad \mathit{tyvar}_\Delta(T[\eta]) \\ \eta = (\overline{\beta := S}) \qquad \Delta; \Gamma \vdash e \mapsto e' \qquad \Delta; \Gamma \vdash \overline{d : U} \qquad \Delta; \Gamma \vdash \overline{d : U \Mapsto_{\Phi[\eta]} d'}\end{array}}{\Delta; \Gamma \vdash e.m[\overline{S}](\overline{d}) \mapsto e'.m_D[\overline{\lfloor S \rfloor}](\overline{d'}).(\lfloor T[\eta] \rfloor \to \lfloor T[\eta] @ V[\eta] \rfloor)}$$

Fig. 16. Compilation rules: expressions (part 2)

implements `Processor[int]`. Struct type `GenericProcessor` is a generic version `IntProcessor`, it implements `Processor[T]` for any `T`. We further define a `useProcessor` method on a client struct, which takes an argument of type `Processor[int]` and invokes the method `process` on it.

Following our uniform compilation strategy and considering interface `Processor`, it would be expected that its `process` method would take a boxed argument and return a result. Thus, when compiling the body of `useProcessor` the compiler would box the argument to `process` and then unbox the return value, in order for it to match the signature of `useProcessor` which has a non-generic return. The attentive reader might have already identified the problem with this naive approach: while `IntProcessor` implements `Processor[int]`, its `process` method is a plain `int` to `int` method with no reason for it to receive or return boxed values. Thus if we pass a value of type `IntProcessor` as an argument to `useProcessor`, the box and unbox operations would be incorrect. If instead we were to pass a value of type `GenericProcessor[int]`, which also implements `Processor[int]`, the boxing strategy described above would be correct.

This example reveals a fundamental challenge in the uniform compilation of methods in the presence of structural typing: When compiling a method body we cannot *a priori* predict which interfaces the type will implement and so we cannot locally determine what arguments must be boxed or not (e.g. method `process` in `IntProcessor` and in `GenericProcessor`). Dually, when considering the call site to an interface method (e.g., the call to `process` in `useProcessor`) we also cannot statically determine the type of the method receiver and so we cannot predict whether arguments and return values need to be boxed by inspecting the signature of the interface. We can manifest the issue further by considering an additional interface `type SimpleProcessor interface {process(x int) int}`. We have that both `IntProcessor` and `GenericProcessor[int]` implement `SimpleProcessor` and any context performing a call to `process` on a value of type `SimpleProcessor` has no way of determining if the receiver expects boxing (i.e., `GenericProcessor`) or not (i.e. `IntProcessor`).

Adaptor Methods. Our solution to this issue is to compile *two* versions of each method m: One whose signature (and name) is identical to the source WG signature and whose body is compiled accordingly; and a *dynamic adaptor* version m_D of the method, that expects *all* arguments and return value to be boxed. The body of m_D redirects to m with the appropriate unboxing operations.

We first address the compilation of method calls. The rules are given in Figure 16. The rules are split into two categories, depending on whether the call is on an interface (i.e., a *dynamic* call) or on a non-interface (i.e., a *static* call): Rule $\textsc{Call}_I$ and Rule $\textsc{Call}_{I\alpha}$ deal with the former, and the two $\textsc{Call}_S$ rules deal with the latter. Two rules per category are required to deal with potential unboxing of the value returned by the call.

In the static call cases, the arguments are compiled according to their WG signature, via the synthetic cast judgement. If the return type of m is boxed then we unbox it ($\textsc{Call}_S$-boxed) since the calling context expects an unboxed result; otherwise, we leave it unchanged ($\textsc{Call}_S$-unboxed).

The compilation of dynamic calls boxes all arguments, as needed by the adaptor (the $\Mapsto$ judgement boxes all values, see Figure 16). As for the return type, we check whether it is a type variable in WG. If so, then the method call happens in a context where the return type is still parametric and must therefore be boxed according to the bound of the type variable. We will see that the adaptor method m_D returns a boxed value with an *empty* method table (the bound information is only determined at the call site). Thus, we change the type accordingly, so that the appropriate method table is generated (rule $\textsc{Call}_{I\alpha}$). If the return type is *not* a type variable, then we type assert the result to the return type. This has the effect of unboxing the result if needed by the calling context.

Revisiting our initial example, the call to `process` in the body of `useProcessor` compiles to a dynamic call to the adaptor for `process`, boxing the integer argument and unboxing the return value, since the return of `process` in `Processor[int]` is not a type variable, Rule $\textsc{Call}_I$ applies, we obtain: $\texttt{processor.process}_D\texttt{(int(42).(int} \to \texttt{int@any)).(int@int)}$.

The compilation of method declarations is given in Figure 17. Rule d-func generates two methods per WG method m. Method m is the compiled version of the original method, with an identical signature (modulo type compilation) and with method body obtained by compiling the source body. Adaptor method m_D bridges dynamic method call destined to m. Its signature is obtained by forcing the types of arguments and the return type to be boxed—achieved by the auxiliary function $\langle\Phi\rangle$.

The adaptor method body consists of a (static) call to method m, generated by an auxiliary *Adapt* function that inspects the WG signature of m to determine which arguments need to be unboxed and whether the return value of m needs to be boxed. To ensure the adaptor returns a boxed result, we insert a make instruction with an empty method table via the synthetic cast ($\lfloor V \rfloor \to_{\emptyset} \lfloor V\texttt{@Any} \rfloor$) when required. Finally, Rule d-type compiles type declarations. Its sole purpose is to apply the $\lfloor\Phi\rfloor$ transformation on all (anonymous) interfaces it may contain.

5.3 Properties of Compilation

Compilation of a well-typed WG program results in a well-typed LWG program that preserves the behaviour of the original WG program. The former property is embodied in Theorem 5.2, which states that a well-typed WG expression of type T compiles to a well-typed LWG expression of type $\lfloor T \rfloor @ \lfloor \mathit{bounds}_{\Delta}(T) \rfloor$, presupposing all declarations in the ambient program are compiled. The

Compilation: declarations $\boxed{\Delta; \Gamma \vdash D \mapsto \overline{D}}$

D-FUNC

$$\frac{\begin{array}{c} mM = m[\overline{\beta\ S}](\overline{y\ \Phi})\ \Psi \qquad \overline{\Phi} = \overline{T@T'} \qquad \Delta = \overline{\alpha : R}, \overline{\beta : S} \qquad \Gamma = x : t[\overline{\alpha}], \overline{y : T} \\ \overline{R \mapsto R'} \qquad \overline{S \mapsto S'} \qquad \Delta; \Gamma \vdash e : V \qquad \Delta; \Gamma \vdash e : V \mapsto_{\Psi} e' \\ D = \mathbf{func}\ (x\ t[\overline{\alpha\ R'}])\ m_D[\overline{\beta\ S'}](\overline{y\ \lfloor\langle\Phi\rangle\rfloor})\ \lfloor\langle\Psi\rangle\rfloor\ \{\ \mathbf{return}\ \mathit{Adapt}((x\ t[\overline{\alpha\ R}])\ mM)\ \} \end{array}}{\mathbf{func}\ (x\ t[\overline{\alpha\ R}])\ mM\ \{\mathbf{return}\ e\} \mapsto \{D,\ \mathbf{func}\ (x\ t[\overline{\alpha\ R'}])\ m[\overline{\beta\ S'}](\overline{y\ \lfloor\Phi\rfloor})\ \lfloor\Psi\rfloor\ \{\mathbf{return}\ e'\}\}}$$

$$\frac{\mathit{iface}(V') \qquad \neg\mathit{iface}(T_i') \implies d_i = y_i.(\lfloor T_i@T_i'\rfloor) \qquad \mathit{iface}(T_i') \implies d_i = y_i.(T_i \to T_i@T_i')}{\mathit{Adapt}((x\ t[\overline{\alpha\ R}])\ m[\overline{\beta\ S}](\overline{y\ T@T'})\ V@V') = x.t[\overline{\alpha}]\#m[\overline{\beta}](\overline{d})}$$

$$\frac{\neg\mathit{iface}(V) \qquad \neg\mathit{iface}(T_i') \implies d_i = y_i.(\lfloor T_i@T_i'\rfloor) \qquad \mathit{iface}(T_i') \implies d_i = y_i.(T_i \to T_i@T_i')}{\mathit{Adapt}((x\ t[\overline{\alpha\ R}])\ m[\overline{\beta\ S}](\overline{y\ T@T'})\ V@V) = x.t[\overline{\alpha}]\#m[\overline{\beta}](\overline{d}).(\lfloor V\rfloor \to_{\emptyset} \lfloor V@\mathsf{Any}\rfloor)}$$

D-TYPE

$$\frac{\overline{R \mapsto R'} \qquad T \mapsto T'}{\mathbf{type}\ t[\overline{\alpha\ R}]\ T \mapsto \mathbf{type}\ t[\overline{\alpha\ R'}]\ T'}$$

Fig. 17. Compilation rules: method declarations

resulting type captures the essence of our typing scheme for LWG, where expressions are assigned a (logical) type but also track the type bound.

THEOREM 5.2. *If* $\Delta; \Gamma \vdash e : T$ *and* $\Delta; \Gamma \vdash e \mapsto e'$ *then* $\lfloor\Delta\rfloor; \lfloor\Gamma\rfloor_{\Delta} \vdash e' : \lfloor T\rfloor@\lfloor \mathit{bounds}_{\Delta}(T)\rfloor$.

Before stating our behavioural correspondence result, we define how WG and LWG programs should be related. All definitions introduced here are given formally in the online appendix.

There is a lowering simulation between a WG program $\overline{D} \triangleright e$ and an LWG program $\overline{D'} \triangleright e'$, if whenever $e \longrightarrow d$ then $e' \longrightarrow^* d'$, and e' and d' are in a lowering simulation. Also, if e is a value, then d is an *equivalent* value; and if e is stuck, then so is d. A lifting simulation is the reverse, e.g., if d can make a move, then e should be able to match it, etc. The theorem below relies on relation $\doteq$ (a weak bisimulation) on LWG expressions and which essentially adds or removes synthetic casts.

THEOREM 5.3. *Suppose* $\overline{D} \triangleright e \checkmark$, *then:* $\mathcal{R} \overset{def}{=} \{\, (e, d') \mid \exists d.\ \emptyset; \emptyset \vdash e \mapsto d \text{ and } d \doteq d' \,\}$ *is a lowering simulation; and* $\mathcal{R} \overset{def}{=} \{\, (d, e) \mid \exists d'.\ \emptyset; \emptyset \vdash e \mapsto d' \text{ and } d \doteq d' \,\}$ *is a lifting simulation.*

6 Related Work

This work fits within the established tradition of programming language research that studies formal models of real-world languages, e.g., [Amin and Rompf 2017; Griesemer et al. 2020; Grigore 2017; Igarashi et al. 2001; Jung et al. 2018; Kennedy and Syme 2001; Park et al. 2015].

Formal investigations of Go. Griesemer et al. [2020] presented the first core formalism of Go in FGG (Featherweight Generic Go), modelling generic types and structural subtyping. WG was initially motivated by an investigation into the type safety of the features of type unions and type sets. This quickly led to the observation that FGG's core subset fails to characterise significant semantic features of Go. By relying on nominal typing of structs and omitting underlying and anonymous types, FGG does not accurately model Go's structural typing that allows methods to accept differently-named structs with the same underlying structure. FGG relies solely on interface-driven subtyping, which does not capture Go's full structural typing nor expose the subtleties in different type coercions: assignability (a non-transitive superset of interface subtyping), static conversions (safe no-ops), and dynamic assertions (requiring RTTI).

Various approaches to implementing Go have been studied based on FGG. Griesemer et al. [2020] formalised monomorphisation as a translation from FGG to non-generic FG. Ellis et al. [2022] presented an approach based on translating FGG to non-generic FG using structs to implement RTTI dictionaries for generic type arguments. Sulzmann and Wehr [2023] presented an approach of translating FGG sans dynamic type assertions (and RTTI) to an *untyped* λ-calculus similar to that known from Haskell type classes. All these remain within FGG, and thus do not consider the features that we have introduced in WG. None consider the underlying mechanics of the Go runtime, such as interface values, RTTI, runtime type conversions and static versus dynamic resolution of operations, as in LWG. In LWG, we distinguish concrete runtime items (e.g., RTTI) from ghost entities used by our metatheory to establish our correctness properties. Note that LWG actions and rules that operate solely on ghost entities correspond to no-ops in an actual implementation.

Regarding correctness, Griesemer et al. [2020] shows FGG programs and their FG monomorphisations are bisimilar. Ellis et al. [2022] show their FGG to FG translation is a bisimulation up to dictionary resolution. Sulzmann and Wehr [2023] show their (untyped) λ-calculus translation is value and divergence preserving. By contrast, we establish a behavioural equivalence between high level WG and compilations into low level LWG. The challenges we address in this paper relate to establishing safety for LWG and the correspondence between high and low level behaviours.

Implementations of generics. Various approaches have been used to implement generics in OO languages with differing implications and trade-offs. Java employs an erasure approach [Bracha et al. 1998] where generic type information is discarded by the compiler, which restricts programs from performing operations that may dynamically depend on such. C# employs a mixed approach [Kennedy and Syme 2001; Yu et al. 2004] where generic code for object types is generated once statically, supported by RTTI at runtime, but generic code for primitives types is specialised dynamically by JIT compilation. The type theoretic study of RTTI for polymorphism dates back to Harper and Morrisett [1995] and Crary et al. [1998]. Rust employs static monomorphisation [The Rust Team 2025; Turon 2015], enabling zero-cost abstractions at the cost of output code size and compilation time. All use nominal subtyping, and do not tackle the interplay between structural subtyping, generics and non-uniform representations (interfaces vs. constants/structs) as in Go.

As discussed earlier, Go employs static monomorphisation [Griesemer et al. 2020] and runtime dictionary passing [Randall 2022]. The Go compiler monomorphises code up to GC shapes with RTTI for lost type information to balance specialisation against code size. This incurs compilation costs and restrictions such as disallowing polymorphic recursion. This paper instead combines Go's runtime infrastructure for type conversions with statically generated adapter code, incurring more runtime conversions but supporting all of WG, including proposed features like generic methods, while lifting restrictions and maintaining separate compilation compatibility.

Leroy [1992] presented an approach of boxing and unboxing generic function arguments and results for ML, with careful treatment of higher order functions. It does not consider subtyping and the language disallows polymorphic recursion. By contrast, our work tackles the problem of compiling generic code in the presence of structural subtyping between non-uniform representations.

In future work, we will consider techniques to optimise our approach, such as combining our adaptors with localised monomorphisation of package private code, and statically safe elimination of type conversions [Leroy 1992]. The overhead of runtime assembly of runtime 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 [Viroli and Natali 2000].

Acknowledgments

We would like to thank Ian Lance Taylor for valuable discussions, and the anonymous reviewers for their feedback. This work was supported by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects UID/50021/2025 and UID/PRR/50021/2025.

Data-Availability Statement

We have implemented a minimal prototype of WG and LWG in Go as interpreters, and our compilation approach as a translation from WG to LWG. The interpreters perform type checking of programs and reduction of their main expressions. Our software artifact is available on Zenodo [Hu et al. 2025]. It also includes the examples from this paper as tests.

References

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 666–679. doi:10.1145/3009837.3009866

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.

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. doi:10.1145/286936.286957

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. doi:10.1145/289423.289459

dominikh. 2022. Method sets section doesn't seem quite right for interfaces with type lists. https://github.com/golang/go/issues/51183. GitHub issue #51183, Go programming language repository.

Stephen Ellis, Shuofei Zhu, Nobuko Yoshida, and Linhai Song. 2022. Generic Go to Go: dictionary-passing, monomorphisation, and hybrid. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1207–1235. doi:10.1145/3563331

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 (2020), 149:1–149:29. doi:10.1145/3428217

Robert Griesemer and Ian Lance Taylor. 2022. An Introduction To Generics. https://go.dev/blog/intro-generics

Radu Grigore. 2017. Java generics are turing complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* ACM, 73–85. doi:10.1145/3009837

Robert Harper and J. Gregory Morrisett. 1995. Compiling Polymorphism Using Intensional Type Analysis. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995.* ACM Press, 130–141. doi:10.1145/199448.199475

Raymond Hu, Julien Lange, Bernardo Toninho, Philip Wadler, Robert Griesemer, and Keith Randall. 2025. Welterweight Go: Boxing, Structural Subtyping and Generics (Artifact). Zenodo. doi:10.5281/zenodo.17741038

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. doi:10.1145/503502.503505

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. doi:10.1145/3158154

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. doi:10.1145/378795.378797

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. doi:10.1145/143165.143205

Mario Macías. 2021. proposal: spec: allow type parameters in methods. GitHub Issue. https://github.com/golang/go/issues/49085 Issue #49085, golang/go repository.

Daejun Park, Andrei Stefanescu, and Grigore Rosu. 2015. KJS: a complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 346–356. doi:10.1145/2737924.2737991

Keith Randall. 2022. Go 1.18 Implementation of Generics via Dictionaries and Gcshape Stenciling. https://github.com/golang/proposal/blob/e9af402b19db4352e7831b33a3f47719e86a5267/design/generics-implementation-dictionaries-go1.18.md

Martin Sulzmann and Stefan Wehr. 2023. A type-directed, dictionary-passing translation of method overloading and structural subtyping in Featherweight Generic Go. *J. Funct. Program.* 33 (2023). doi:10.1017/S0956796823000047

Ian Lance Taylor and Robert Griesemer. 2021. Type Parameters Proposal: No parameterized methods. https://go.googlesource.com/proposal/+/refs/heads/master/design/43651-type-parameters.md#no-parameterized-methods

The Go Team. 2024. The Go Programming Language Specification: Method sets. https://go.dev/ref/spec#Method_sets

The Go Team. 2025. The Go Programming Language Specification. https://golang.org/ref/spec

The Rust Team. 2025. Generic Data Types. https://doc.rust-lang.org/book/ch10-01-syntax.html Chapter 10.1.

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. doi:10.1145/353171.353182

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. doi:10.1145/964001.964005

Received 2025-07-10; accepted 2025-11-06