Generic Go to Go

Dictionary-Passing, Monomorphisation, and Hybrid

STEPHEN ELLIS*, University of Oxford, United Kingdom
SHUOFEI ZHU*, The Pennsylvania State University, United States
NOBUKO YOSHIDA, University of Oxford, United Kingdom
LINHAI SONG, The Pennsylvania State University, United States

Go is a popular statically-typed industrial programming language. To aid the type safe reuse of code, the recent Go release (Go 1.18) published early 2022 includes *bounded parametric polymorphism* via *generic types*. Go 1.18 implements generic types using a combination of *monomorphisation* and *call-graph based dictionary-passing* called *hybrid*. This hybrid approach can be viewed as an optimised form of monomorphisation that statically generates specialised methods and types based on possible instantiations. A monolithic dictionary supplements information lost during monomorphisation, and is structured according to the program's call graph. Unfortunately, the hybrid approach still suffers from code bloat, poor compilation speed, and limited code coverage.

In this paper we propose and formalise a new *non-specialising call-site based dictionary-passing* translation. Our call-site based translation creates individual dictionaries for each type parameter, with dictionary construction occurring in place of instantiation, overcoming the limitations of hybrid. We prove it correct using a novel and general bisimulation up to technique. To better understand how different generics translation approaches work in practice, we benchmark five translators, Go 1.18, two existing monomorphisation translators, our dictionary-passing translator, and an erasure translator. Our findings reveal several suggestions for improvements for Go 1.18— specifically how to overcome the expressiveness limitations of generic Go and improve compile time and compiled code size performance of Go 1.18.

CCS Concepts: • **Software and its engineering → Polymorphism**; **Compilers**; • **Theory of computation → Semantics and reasoning**.

Additional Key Words and Phrases: Generic types, Dictionary-passing translation, Correctness

ACM Reference Format:
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, Article 168 (October 2022), 29 pages. https://doi.org/10.1145/3563331

1 INTRODUCTION

Since its creation in 2009, the Go programming language has placed a key emphasis on simplicity, safety, and efficiency. Based on the Stack Overflow [2021] survey, Go is the 5th most beloved language, and is used to build large systems, *e.g.,* Docker [2021], Kubernetes [2021], and gRPC [2021]. The recent Go release (Go 1.18 released on the 15th of March 2022) added *generics*, which

*Stephen Ellis and Shuofei Zhu contributed equally to this work.

Authors' addresses: Stephen Ellis, stephen.ellis@cs.ox.ac.uk, University of Oxford, Oxford, United Kingdom; Shuofei Zhu, sfzhu@psu.edu, The Pennsylvania State University, University Park, PA, United States; Nobuko Yoshida, nobuko.yoshida@cs.ox.ac.uk, University of Oxford, Oxford, United Kingdom; Linhai Song, songlh@ist.psu.edu, The Pennsylvania State University, University Park, PA, United States.

2475-1421/2022/10-ART168
https://doi.org/10.1145/3563331

has been considered Go's most critical missing and long awaited feature by Go programmers and developers [Merrick 2020]. The Go Team [2022], however, has posted that much work is still needed to ensure that generics in Go are well-implemented.

The work on implementing generics in Go began in earnest with Griesemer et al. [2020], in which they formalised two core calculi of (generic) Go; Featherweight Generic Go (FGG) and Featherweight Go (FG), as well as formalising a *monomorphisation translation* from FGG to FG. Monomorphisation statically explores a program's call graph and generates multiple implementations of each generic type and method according to each specialisation of that type, or method, required at runtime.

The Go team informally proposed three approaches; (1) Stencilling (monomorphisation) [Randall 2020c], (2) Call-graph dictionary-passing [Randall 2020a], and (3) GC shape stencilling (hybrid of (1) and (2)) [Randall 2020b]. A monomorphisation-based source-to-source prototype (Go prototype translator (go2go)) has been implemented by The Go Team [2021b], following the stencilling proposal (1) and [Griesemer et al. 2020]. The current Go 1.18 implementation extends (3) [Randall 2022]. Unlike more traditional *non-specialising* dictionary approaches (*e.g.,* dictionary-passing in Haskell and vtables in C++), Go 1.18 uses an optimised form of monomorphisation to allow types in the same GC shape group to share specialised method and type instances. In theory, all objects in a GC shape group have an equivalent memory footprint and layout, although currently, Go 1.18 only groups pointers. As multiple types may share the same GC shape group, their dictionaries provide information lost during monomorphisation, *e.g.,* concrete types and method pointers. Moreover, Go 1.18 builds a monolithic dictionary based on the program's *call-graph*. Monomorphisation has a number of well-known limitations; it can substantially increase code size, it can be prohibitively slow during compilation [Jones 1995; Stroustrup 1997], and it does not cover all programs [Griesemer et al. 2020]. Concretely, there are two core limitations with all the Go team proposals (1–3), the current Go 1.18 implementation, and the proposal of Griesemer et al. [2020].

1) Non-monomorphisable programs. All current implementations and proposals for generics in Go suffer from the inability to handle a class of programs that use recursive instantiations, *e.g.,* the list permutation example[1] provided in Figure 1. This program cannot be monomorphised, as a list of integers `List[int]` has a `permute` method which returns a list of type `List[List[int]]`, which in turn has a `permute` method that returns type `List[List[List[int]]]`, and on *ad infinitum*. Monomorphisation cannot explore this infinite set of types in finite time, and so cannot specialise a method for each instance.

```
type List[T Any] interface {
  permute() List[List[T]]; insert(v T, i int) List[T];
  map[R Any](func(T) R) List[R] ; len() int }
type Cons[T Any] struct { head T; tail List[T] }
type Nil[T Any] struct {}
func (this Cons[T]) permute() List[List[T]] {
  if this.len() == 1 { return Cons{this, Nil{}}
  } else {
    return flatten(this.tail.permute().Map(
      func(l List[T]) List[List[T]]{
        var l_new List[List[T]] = Nil[List[T]]{}
        for i := 0; i <= l.len(); i++ {
          l_new = Cons{l.insert(this.head, i), l_new}
        }
        return l_new
})))}
func (this Nil[T]) permute() List[List[T]] {
  return Nil[List[T]]{}
}
```

Fig. 1. List permutation example

2) Specialising translation. All currently realised approaches to generics in Go are based on method/type specialisation. This stands in contrast to the approaches taken by other languages with automatic memory management, such as Haskell, C#, and Java. Go uses garbage collection for automatic memory management. In the top 16 statically typed languages with generics [Spectrum 2022], we find a constant theme; languages with automatic memory management use non-specialising implementations such as dictionary-passing or erasure, and those without use monomorphisation (see the full version of this paper [Ellis et al. 2022d] for a breakdown of language implementations).

[1]See [gitchander 2021] for an efficient but type unsafe implementation of list permutation.

Challenges and contributions. We develop and implement a new non-specialising, call-site dictionary-passing translation from Go with generics (FGG) to Go (FG), and prove its correctness. We then create micro and real-world benchmarks for generic Go, and examine the trade-offs between the different translations to suggest improvements for Go 1.18.

1) The first challenge is to design and build a non-specialising call-site dictionary-passing translation for Go. Go's distinctive structural subtyping adds an extra level of complexity that requires careful consideration. Our first contribution in § 4 and § 6.1 is the formalisation and implementation of a new dictionary-passing translation that is specifically designed for the unique qualities of Go.

2) The second challenge is to overcome the non-monomorphisability limitation of the current implementations and translate previously untranslatable programs such as `permute`. A key aspect of our dictionary design is *call-site*—each polymorphic type parameter is represented by its own dictionary, which in turn is created at the call-site where that type parameter would have been instantiated. This allows any well-formed FGG program to be translated.

3) The third challenge we meet is to establish semantic correctness of our translation. Historically, dictionary-passing translations have been proven correct using value preservation [Sulzmann and Wehr 2021; Yu 2004; Yu et al. 2004], an approach that cannot ensure termination preservation or generalise to more advanced language features (*e.g.,* concurrency in Go). We instead use a fine-grained behavioural equivalence guided by the work of Igarashi et al. [1999]. Unfortunately, proving the *bisimulation* result in [Griesemer et al. 2020, Theorem 5.4] is insufficient due to intermediate states created by dictionary-passing. We propose a novel *bisimulation up to dictionary resolution* reduction, and use this relation to prove that the translation preserves essential properties of the source language (§ 5). This proof technique is general and translation-agnostic, and is useful in other contexts where a standard bisimulation is inadequate.

4) The fourth challenge is to find an effective evaluation for implementations of generics in Go. We compare the five implementations— (1) our call-site, non-specialising dictionary-passing translation; (2) an erasure translation built by us for empirical evaluation; (3) a monomorphisation translation by Griesemer et al. [2020]; (4) the initial source-to-source monomorphisation prototype translation go2go by the Go team; and (5) Go 1.18 —along three dimensionalities; (1) compilation time, (2) translated code size, and (3) performance of compiled executables. As Go 1.18 was just released, *there currently exists no real-world Go program with generics.* In § 6.2, we contribute a number of benchmarks to overcome this deficit: we construct micro benchmarks to examine the effect of different forms of complexity in generic programs; and reimplement the real-world benchmarks from [Odersky et al. 2000; Ureche et al. 2013] in Go.

5) The final challenge is to examine the trade-offs between the different translations, which suggest future improvements of Go 1.18. We observe, in general, that monomorphisation leads to better execution performance, while non-specialisation (dictionary-passing) produces smaller executables in less compilation time. We also observe that on the micro benchmarks our dictionary-passing translation can generate programs that are comparable in efficiency to Go 1.18. Overall, our results show that Go 1.18 has much scope for improvement and the usefulness of non-specialised call-site dictionary-passing translations for languages such as Go. We provide concrete suggestions in § 6.4.

Outline. § 2 and § 3 summarise FG and FGG; § 4 proposes a new dictionary-passing translation; § 5 proves its semantic correctness; § 6 describes our implementations and measures the trade-offs between the five translators; § 7 gives related work; and § 8 concludes. Proofs and omitted definitions can be found in the full version of the paper [Ellis et al. 2022d]. The dictionary-passing/erasure translators and benchmarks are available in the artifact to this paper [Ellis et al. 2022a]. Source code is available on GitHub [Ellis et al. 2022b] and Software Heritage [Ellis et al. 2022c].

```
1  type Any interface {}
2  type Function interface { Apply(x Any) Any }
3  type Ord interface { Gt(x Ord) bool }
4  type List interface { Map(f Function) List }
5  type Nil struct {}
6  type Cons struct { head Any ; tail List }
7  func main() {
8      _ = Cons{1, Cons{7, Cons{3, Nil{}}}}
9      .Map(GtFunc{5})
10     .Map(GtFunc{5}) // PANIC
11 } // Unable to assert bool as type Ord
12 type GtFunc struct { val Ord }
13 func (this GtFunc) Apply(x Any) Any {
14     return this.val.Gt(x.(Ord)) //Gt needs an Ord arg
15 }
16 func (this int) Gt(x Ord) bool {
17     return x.(int) < this // < needs an int value
18 }
19 func (this Nil) Map(f Function) List { return Nil{} }
20 func (this Cons) Map(f Function) List {
21     return Cons{ f.Apply(this.head), this.tail.Map(f) }
22 }
```

Fig. 2. FG List example adapted from [Griesemer et al. 2020, Figures 1 & 3]

2 FEATHERWEIGHT GO

We briefly summarise the Featherweight Go (FG) language [Griesemer et al. 2020, § 3]; specifically highlighting the key points related to dictionary translation.

2.1 Featherweight Go by Examples

FG is a core subset of the (non-generic) Go 1.16 language containing *structures*, *interfaces*, *methods*, and *type assertions*. In FG, there are two kinds of named types; *Interfaces* (**`interface`**) specify a collection of methods which any implementing type must also possess, and *structures* (**`struct`**) which are data objects containing a fixed collection of typed fields. *Methods* are functions that apply to a specific structure, called the method's *receiver*. Finally, *type assertions* ask whether a structure can be used as a specific type. If it cannot, then FG will produce a *type assertion error*.

In contrast to nominally typed languages, Go uses *structural subtyping*. As we shall see in § 4, it is this distinctive feature that makes our dictionary-passing translation challenging and non-trivial. In a nominally typed language, such as Java, one type implements (subtypes) another when it explicitly declares such. In Go, we do not declare that one type implements another. Rather, one type implements another precisely when it implements (at least) all of the prescribed methods.

Consider the example Go code in Figure 2, which simulates higher order functions, lists, and mapping. For simplicity of presentation, we assume that there are primitive `int` and `bool` types along with a < operation. The `Any` interface does not specify any methods; as such, all other types are its subtypes, meaning that any object may be used when an `Any` is expected, but also that we cannot apply any methods to an `Any` object without first asserting it to some more specific type – an action which may fail at runtime. The `Function` interface specifies a single method, which is given by the *method signature* `Apply(x Any) Any`. Any structure implementing an `Apply` method that takes an argument of type `Any` and returns a value, also of type `Any`, is said to implement the `Function` interface. Our example code simulates the *greater than* function as a structure (`GtFunc`) containing a single `Ord` field. Its `Apply` method then calls the `Gt` method provided by struct's field. The `Ord` interface, however, specifies that `Gt` should accept a single argument of type `Ord`. Before the `Apply` method of `GtFunc` can call `Gt` it must, then, assert its argument to type `Ord`. If the argument does not implement `Ord`, then a *type assertion error* occurs. We assume that only one implementation of `Ord` exists, that being `int`, which itself uses a risky type assertion.

The example also includes a `List` interface specifying a `Map` method. We provide a cons list implementation of `List`. In FG, there is a single top-level `main` function that acts as the program's entrance. Our program initially builds a simple three value `int` list on line 8, and then uses the simulated greater than function (`GtFunc`) to map the list to a `bool` list. When, however, we attempt to map this `bool` list using the same function, we encounter a runtime type assertion error on line 10. While we could catch this error at compile time by increasing the specificity of the `Apply`, `Gt`, and `Map` functions using `int` and `bool` instead of `Any`, this would severely limit code reusability.

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

Fig. 3. Syntax of Featherweight Go

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

$$\text{[r-fields]}\ \frac{(\overline{f\ t}) = fields(t_S)}{t_S\{\overline{v}\}.f_i \longrightarrow v_i} \qquad \text{[r-call]}\ \frac{(x : t_S, \overline{x : t}).e = body(type(v).m)}{v.m(\overline{v}) \longrightarrow e[x := v, \overline{x := v}]}$$

$$\text{[r-assert]}\ \frac{type(v) <: t}{v.(t) \longrightarrow v} \qquad \text{[r-context]}\ \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

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

Fig. 4. Reduction semantics of Featherweight Go

2.2 Featherweight Go Syntax and Semantics

Figure 3 presents the syntax of FG from [Griesemer et al. 2020]. We use the $\overline{x}$ notation for a sequences of x, namely $x_0, x_1, \ldots, x_n$. A program (P) is given by a sequence of declarations ($\overline{D}$) along with a **main** function which acts as the top-level expression (e). Shortened as $P = \overline{D} \triangleright e$.

FG is statically typed: all FG typing rules follow the Go 1.16 specification. If, in the variable-type environment Γ, an expression e is of type t, then it satisfies the judgement $\Gamma \vdash e : t$. We assume that all programs P are *well-formed*, written $P\ ok$. Since the rules/notations are identical to those in [Griesemer et al. 2020], we omit them here, but provide definitions and details in the full version of this paper [Ellis et al. 2022d].

Figure 4 presents the FG semantics with values and evaluation contexts. ***Evaluation context*** E defines the left-to-right call-by-value semantics for expressions. ***Reductions*** are defined by the field selection rule [r-fields], type assertion rule [r-assert], and the method invocation [r-call], with [r-context] for the context evaluation. We use $\longrightarrow^*$ to denote a multi-step reduction. FG satisfies type preservation and progress properties (see [Griesemer et al. 2020, Theorems 3.3 and 3.4]).

3 FEATHERWEIGHT GENERIC GO AND THE LIMITATIONS OF MONOMORPHISATION AND GO 1.18

As with § 2, we briefly summarise the Featherweight Generic Go (FGG) language [Griesemer et al. 2020, § 4]. This section concludes with a discussion of limitations in existing generic Go translations and Go 1.18.

3.1 Featherweight Generic Go by Example

Figure 5 extends Figure 2 with generics. As we saw in § 2.1, there was a critical flaw in the original, non-generic, FG code. One part of the logic was polymorphic (*i.e.,* `Map` is a natural transformation) while the other was not (*i.e.,* `Gt`). We concluded that section by observing the two options; either we cater to the strict type discipline demanded by `Gt`, reducing reusability, or force an excessively permissive polymorphism on `Gt` and risk runtime type assertion errors.

```
1  type Any interface {}
2  type Function[T Any, R Any] interface {
3      Apply(x T) R
4  }
5  type Ord[T Ord[T]] interface { Gt(x T) bool }
6  type List[T Any] interface {
7      Map[R Any](f Function[T, R]) List[R]
8  }
9  type Nil[T Any] struct {}
10 type Cons[T Any] struct { head T;tail List[T]}
11 func main() {
12  _ = Cons[int]{1, Cons[int]{7, Cons[int]{3, Nil[int]{}}}} // : List[int]
13         .Map[bool](GtFunc[int]{5}) // : List[bool]
14         .Map[bool](GtFunc[int]{5}) // This line doesn't pass type checking since
15 } // GtFunc does not implement the Function[bool, bool] interface
16 type GtFunc[T Ord[T]] struct { val T }
17 func (this GtFunc[T]) Apply(x T) bool {
18     return this.val.Gt(x)
19 }
20 func (this int) Gt(x int) bool { return x < this }
21 func (this Nil[T]) Map[R Any](f Function[T, R]) List[R] {
22     return Nil[R]{}
23 }
24 func (this Cons[T]) Map[R Any](f Function[T, R]) List[R] {
25     return Cons[R]{ f.Apply(this.head), this.tail.Map[R](f)}
26 }
```

Fig. 5. FGG List example adapted from [Griesemer et al. 2020, Figures 4 & 6]

Generics, or bounded parametric polymorphism, provide us with a third solution via the precise definition and tracking of polymorphic types in structures, interfaces, and methods. As we shall see momentarily, in FGG, each of these constructs may now accept any number of type variables (type parameters) as a type formal, which must then be instantiated upon use. Each type variable has a bound, an interface, that any instantiating type must satisfy, *i.e.,* be an instance of. Type formal `[T Any]` is read as type parameter `T` is bound by type `Any`. Objects with a generic type can use all methods specified by the type variable's bound. Type variables can be bound by any interface type, and may be mutually recursive within a type formal. Take, for example, the type bound of `Ord` in Figure 5. `Ord` is bound by `Ord` itself and is used recursively in the type bound for `GtFunc`. For a type (*e.g.,* `int`) to instantiate type variable `T` in `[T Ord[T]]`, its `Gt` method must not only take an argument of `Ord`, but must be precisely the same `Ord`-implementing type. This kind of self-referential type bound is known as *F-bounded polymorphism* [Canning et al. 1989].

The interface `Function` is now defined over two type variables (`T` and `R`, both bounded by `Any`), which are used by the specified `Apply` method to type the simulated function's domain and codomain, respectively, e.g., a type implementing `Function[int, bool]` must implement the method `Apply(x int) bool`. Unlike the original FG code, we do not need `GtFunc` to simulate any arbitrary function, but rather just functions from some generic `Ord` type to `bool`. Instantiating `GtFunc` with `int`, written `GtFunc[int]`, gives an implementation of `Function[int,bool]`.

A type bound not only limits which types may specialise a type parameter, but also what methods are available to polymorphic values, *i.e.,* given that all valid specialisations of `T` in `GtFunc[T]` must implement `Ord[T]`, we know that the `val` field must always possess the `Gt` method, allowing us to call to `Gt` on line 18 without a type assertion.

By accepting a type parameter argument, the `Map` method specifies both the codomain of the `Function` argument, and the type specialisation of the returned `List`. Line 14 thus fails during type checking because `GtFunc` does not implement `Function[bool, bool]`.

3.2 Featherweight Generic Go Syntax and Semantics

Figure 6 presents the syntax of FGG. The key differences from FG are the addition of types formal (Ψ, Φ) for method signatures and declarations. A type formal $(\overline{\alpha\ \tau_I})$ is a sequence of pairs, each of which contains a type parameter (α) and parameter bound (τ_I). Type bounds are interface types that can be mutually recursive, in that any bound in a type formal may depend upon any type parameter in that type formal, including itself. Type parameters are instantiated by a type actual (ψ, ϕ) – a sequence of types that satisfy the requirements imposed by the type formal. A type (τ) in FGG is either a type parameter or a declared type that has been instantiated $(t[\phi])$. We simplify method declaration from FGG [Griesemer et al. 2020], following the Go 1.18 syntax.

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

Fig. 6. Syntax of Featherweight Generic Go

Value	$v ::= \tau_S\{\overline{v}\}$				
Evaluation context	$E ::=$				
Hole	$\square$	Structure	$\tau_S\{\overline{v}, E, \overline{e}\}$	Call receiver	$E.m[\psi](\overline{e})$
Select	$E.f$	Type assertion	$E.(\tau)$	Call arguments	$v.m[\psi](\overline{v}, E, \overline{e})$

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

$$\mathit{type}(\tau_S\{\overline{v}\}) = \tau_S \qquad \frac{(\mathbf{func}\ (\mathtt{this}\ t_S[\overline{\alpha}])\ m[\Psi](\overline{x\ \tau})\ \tau\ \{\mathbf{return}\ e\}) \in \overline{D} \quad \theta = (\overline{\alpha}, \Psi := \phi, \psi)}{\mathit{body}(t_S[\phi].m[\psi]) = (x : t_S[\phi], \overline{x : \tau}).e[\theta]}$$

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

Fig. 7. Reduction semantics of Featherweight Generic Go

The type system in FGG extends FG with the addition of a new type variable context Δ mapping type variable to its bound. Expression e of type τ is now given by the judgement $\Delta; \Gamma \vdash e : \tau$. Program well-formedness is given by $P\ ok$. The typing rules follow those given in [Griesemer et al. 2020, Figure 15], which can be found in the full version of this paper [Ellis et al. 2022d].

The reduction semantics of FGG are defined in Figure 7. They extend those of FG; notably, [r-call] (via the *body* auxiliary function) specialises generic types in the resolved method body. FGG satisfies type preservation and progress properties (see [Griesemer et al. 2020, Theorems 4.3 and 4.4]).

3.3 The Limitation of Monomorphisation

Griesemer et al. [2020] define a class of programs that their monomorphisation approach cannot translate. This limitation also applies to the Go 1.18 call-graph based dictionary implementation for the same rationale. Consider the model non-monomorphisable program in Figure 8.

```
1 type Box[α Any] struct { value α }
2 func (b Box[α]) Nest(n int) Any {
3   if (n > 0) {
4     return Box[Box[α]]{b}.Nest(n-1)
5   } else { return b }
6 }
```

Fig. 8. Box example [Griesemer et al. 2020, Figure 10]

Intuitively, the fundamental issue with this deceptively simple program is that *instance set discovery* is non-terminating. To monomorphise a program, we first need to discover all possible type instantiations used in said program. Perfectly well-behaved programs may however produce infinitely many type instantiations. This occurs when an instance of a (mutually) recursive method eventually depends upon a greater instantiation of itself, which in turn depends on an even greater instantiation of itself *ad infinitum*, *e.g.*, `Box[int].Nest()` depends upon the specialisation `Box[Box[int]].Nest()`. In [Griesemer et al. 2020], such programs are called *nomono*.

3.4 Go 1.18 Implementation

The official release of Go 1.18 uses an optimised version of monomorphisation called *dictionaries and GC shape stenciling* [Randall 2022]. When possible, their implementation reuses monomorphised functions to reduce code size. Two objects may share the same specialised method implementation when they have the same GC shape. In the current implementation, the criteria of having the same GC shape means they are of the same data type, or both are pointers. Each function therefore must to have a dictionary to differentiate concrete types at runtime. A dictionary contains (1) the runtime type information of generic type parameters, as well as (2) their derived types used in the function. In the function body, each generic function call that depends on the generic type parameters also needs a dictionary; (3) these sub-dictionaries required by the method calls are also provided in the dictionary. Additionally, the dictionary provides each generic object with (4) the data structure that Go runtime uses to conduct method calls.

Go 1.18 would also need to create an infinite call-graph dictionary for the `Box` example in Figure 8, as well as for the `permute` example in Figure 1. Hence, Go 1.18 cannot handle either example. Our call-site dictionary-passing approach does not suffer this limitation.

4 CALL-SITE, NON-SPECIALISING DICTIONARY-PASSING TRANSLATION

This section presents our new dictionary-passing translation from FGG to FG.

High level overview. Our call-site, non-specialising dictionary-passing translation can be split into a number of parts, each tackling a different challenge. Specifically, we consider: the preservation of typeability, the use of dictionaries to resolve generic method implementations, the creation of dictionaries, and the preservation of type assertion behaviour. These challenges may have been discussed in other works, yet the structural type system of Go serves to hinder any existing solutions. We explain the key ideas and challenges in § 4.1, and detail the formal translation rules in § 4.2.

4.1 Dictionary-Passing by Example

4.1.1 Structural Subtyping and Type Erasure. The first challenge we encounter is that subtypes must be preserved. If, in the source program, expression e can be used as an argument to `Foo`, then the translation of e should likewise be usable as an argument to the translation of `Foo`. We should also desire that any non-subtypes are preserved, we leave this challenge to § 4.1.3.

As a first naive attempt at removing polymorphic types, we might observe that regardless of the value we pass to a polymorphic argument, it must implement the `Any` type. From this, we could – again, naively – conclude that lifting all polymorphic arguments to the `Any` type solves our problem. Unfortunately, such a solution fails upon closer inspection. Consider the code in Figure 5. By erasing the polymorphic types in `Function`, we lose the subtype `GtFunc[int]` <: `Function[int, bool]` (The naively erased `GtFunc` implements `Apply(in Any) bool`, while the erased `Function` demands `Apply(in Any) Any`). This issue is noted in Igarashi et al. [1999, § 4.4].

Their solution, however, is inappropriate in a structurally typed language such as Go. In nominally typed languages like Java, it is clear that one type subtypes another. One need only inspect the implementing type's declaration, as a subtype exists only when it is explicitly declared. Igarashi et al. [1999] insert *bridge methods* to handle cases such as the `GtFunc`-`Function` example. A bridge method is an overloaded method added to the subtype whose type matches the erased method as specified by the supertype, *i.e.,* adding an overloaded method of type `Apply(in Any) Any` to `GtFunc`. This method is immediately inappropriate as Go does not allow method overloading.

The bridge method solution would still be inappropriate were we to simulate overloading using name mangling. To add bridge methods, we need to know – statically – that a subtype exists. In FGG, we need to know how two types are instantiated before we can conclude that a subtype

```
1  type Ord[T Ord[T]] interface {
2      Gt[](that T) bool
3  }
4  type GtFunc[T Ord[T]] struct { val T }
5  func (this GtFunc[T]) Apply(in T) bool {
6      return this.val.Gt[](in)
7  }
8  type Max struct {}
9  func (this Max) Of[T Ord[T]](l T, r T) T {
10     ··· l.Gt(r) ···
11 }
12 func main() { GtFunc[int]{5}.Apply(7) }
```

```
1  type Ord interface{ Gt(that Any) Any }
2  type OrdDict struct {
3      Gt func(rec Any, in Any) Any ; //Gt method pointer
4      /*Simulated type*/ }
5  type GtFunc struct { val Any ; dict OrdDict }
6  func (this GtFunc) Apply(in Any) Any {
7      return this.dict.Gt(this.val /*Receiver*/, in)
8  }
9  func (this Max) Of(dict OrdDict, l Any, r Any) Any {
10     ··· dict.Gt(l, r) ··· }
11 func main() {
12   od := OrdDict{Gt: func(rec Any, in Any) Any { rec.(int).Gt(in)}}
13   GtFunc{5, od}.Apply(7) }
```

Fig. 9. Dictionary-passing translation example extending Figure 2. FGG source (Left), FG translation (Right)

relation exists. This requires the kind of potentially infinite whole program analysis (§ 3.3) that we wished to avoid in our dictionary-passing translation. Instead, we ensure that subtypes are preserved by erasing *all* method types, rather than just polymorphic types. As with `GtFunc`'s `Apply` method in Figure 2, when a variable of a known type is used, we assert it to that type; although unlike Figure 2, the FGG type checker has already ensured the safety of these synthetic assertions.

4.1.2 Dictionaries. We are now confronted with the primary challenge of concern to dictionary-passing translations; how do we resolve generic method calls without polymorphic type information? A dictionary is, at its simplest, a map from method names to their specific implementation for some type. A dictionary-passing translation, then, is one which substitutes the specialisation of type parameters with the passing of dictionaries as supplementary value-argument. One may then resolve a method call on a generic value by performing a dictionary lookup.

Presently, we consider the structure and usage of dictionaries while delaying our discussion of call-site dictionary construction and type simulation until § 4.1.4 and § 4.1.3, *resp.* Consider Figure 9 (left) extending a fragment of Figure 2 with a `Max.Of` method. For us to call `Gt` in `GtFunc[T].Apply` or `Max.Of[T]`, we need to know the concrete type of `T`. This information is lost during erasure.

The translation (right) includes a fresh struct `OrdDict` which is, quite naturally, the dictionary for `Ord` bounded type parameters. Dictionaries contain a method pointer field for each method in the original interface, along with a *type-rep* which shall be discussed in § 4.1.3. FG does not include method pointers; instead, we must simulate them using higher order functions with the first argument being the receiver. While this adds a small amount of complexity to the final correctness proofs, we see this as a worthwhile compromise, as it allows us to focus on the translation of generics alone, rather than on generics *and* on a translation to some low level language. By containing each method specified by the FGG bounding interface, dictionaries have a fixed internal representation. This reflects real-world dictionary-passing implementations and allows entries to be accessed efficiently [Driesen and Hölzle 1996].

Dictionaries are passed to methods via two mechanisms, namely the method's receiver, and as regular value-arguments. Generic structures, *e.g.*, `GtFunc`, possess a dictionary for each type parameter. When used as a receiver, these dictionaries can be accessed using standard field destructuring. Method dispatch then takes the form of a dictionary lookup and method invocation as seen on lines 7 and 10 (right).

```
1  type Foo[α Any] interface {
2      do[β Any](a β, b bool) α
3  }
4  type Bar[α Any] struct {}
5  func (x Bar[α]) do[β Any](a β, b α) int {···}
6  func main() {
7      Bar[bool]{}.(Foo[int]);
8      Bar[bool]{}.(Foo[bool])
9  }
```

Fig. 10. Type-rep example. FGG source

```
1 type _type_metadata interface { tryCast (in Any) Any }
2 type AnyDict struct {_type _type_metadata}
3 type Foo interface { do(dict0 Anydict, in Any) Any ; spec_do() spec_metadata4 }
4 type Foo_meta struct { _type0 _type_metadata }
5 func (this Foo_meta) tryCast(x Any) Any {  // Type formal, Parametrised arg, Literal arg, return type α
6   if (x.(Foo).spec_do() != spec_metadata4{Any_meta{}, param_index0{}, Bool_meta{}, this._type0 }) { panic }
7   return x }
8 type Bar struct {dict0 AnyDict}
9 func (this Bar) spec_do() spec_metadata4 { // Type formal, Parametrised arg, Arg type α, return type literal
10   return spec_metadata4{Any_meta{}, param_index0{}, this.dict0._type, Int_meta{}}}
11 func main() {
12   Foo_meta{Int_meta{}}.tryCast(Bar{AnyDict{Bool_meta{}}})
13   Foo_meta{Bool_meta{}}.tryCast(Bar{AnyDict{Bool_meta{}}}) }
```

Fig. 11. Type-rep example. FG translation

4.1.3 Type Collision. Here we consider the challenge of ensuring that type assertion behaviour is preserved by our translation. Erasing type parameters may introduce new subtypes which did not exist in the source program. Consider the expression `GtFunc[int]{5}.(Function[bool, bool])` where `GtFunc` and `Function` are defined in Figure 5. Upon evaluation, this expression produces a runtime type assertion error as `GtFunc[int]{5}` is not a subtype of `Function[bool, bool]`. Their erased types (§ 4.1.1), however, form a subtype relation, meaning the error will not occur in the translated code. This behaviour would be incorrect. To ensure that type assertion errors are correctly preserved we simulate the FGG type assertion system inside the translated FG code via type-reps [Crary et al. 1998]. A simulated FGG type implements `_type_metadata` by specifying a method, `tryCast`, which throws an error if and only if the FGG assertion would have failed.

Consider the code in Figure 10. The source FGG code contains two assertions; the one on line 7 passes, while line 8 produces a type assertion error.

A struct implements an interface when it correctly implements each method specified by the interface. This means that not only does the struct define a method of the same name, but also of precisely the same type. Assertion to an interface, then, need only ensure that each method is correctly implemented. Assertion to a structure is a simple type equality check.

The translated interface, Figure 11, now includes the meta method `spec_do`, returning simulated FGG type information for a struct's `do` implementation.

The $\texttt{spec\_metadata}_4$ object returned by `spec_do` on line 10 of the target code is a four-element tuple containing: type parameter bounds, argument types, and the return type. This object simulates the FGG method type for `do` on `Bar[`τ`]` for some τ, *i.e.,* `do[`β `Any](a` β`, b` τ`) Int[]`. The first entry `Any_meta{}` gives the simulated type bound of the source method's type parameter β. The next gives the type of argument `a`, namely β. As there is no suitable concrete metadata type for β, we use an index $\texttt{param\_index}_0$ to indicate that `a`'s type is the method's first type parameter. The third, that of `b`, is not known at compile time, but is rather given by the type parameter of the receiver. Finally, the return type is given by the constant `Int_metadata`.

The type assertion on line 13 uses the `Foo_meta`'s `tryCast` method defined on line 5. This method first checks that the erased types are compatible, *i.e.,* that `Bar` implements all erased methods in `Foo`. The `spec_do` method is then used to check the simulated method type matches the interface specification. If any of these checks is failed then the assertion fails and a panic is thrown.

4.1.4 Call-Site Dictionary Creation. As discussed in § 3.3, the approach taken by Go 1.18 is fundamentally limited by its use of call-graph based dictionary construction. In contrast we consider the challenge of the call-site construction of dictionaries in a structurally typed language. Our approach overcomes the aforementioned limitation of [Griesemer et al. 2020] and Go 1.18.

```
type Eq[α Eq[α]] interface {
    Equal(that α) bool
}
type Ord[α Ord[α]] interface {
    Gt(that α) bool;
    Equal(that α) bool
}
func Foo[β Ord[β]](val β) Any {
    return Bar[β](val)
}
func Bar[β Eq[β]](val β) Any {···}
func main() { Foo[int](5) }
```

```
type EqDict struct { Equal func(rec Any, that Any) Any }
type OrdDict struct {
  Equal func(rec Any, that Any) Any ;
  Gt func(rec Any, that Any) Any
}
func Foo(dict OrdDict, val Any) Any {return Bar(EqDict{dict.Equal}, val)}
func Bar(dict EqDict, val Any) Any { ··· }
func main() {
  old_dict := OrdDict{
    Equal : func(rec Any, that Any) Any { return rec.(int).Equal(that) }
    Gt : func(rec Any, that Any) Any { return rec.(int).Gt(that) } }
  Foo(old_dict, 5) }
```

Fig. 12. Call-site dictionary creation example. FGG source (Left). FG translation (Right)

We note a few key facts. A τ-dictionary provides all the methods specified by the type bound τ, and we may build a dictionary for any specialising type which is a subtype of τ. We can also use a type variable to specialise some other type variable as long as the bound of the later is a supertype of the former. In a translation this *dictionary-supertyping* involves using a τ-dictionary to build a potentially different σ-dictionary. In a nominally typed language the explicit, and fixed, hierarchy allows a dictionary-passing translation to easily structure and construct dictionaries according to the subtype hierarchy. Dictionary-supertyping in nominally typed languages is generally a matter of extracting the appropriate sub-dictionary [Bottu et al. 2019].

In a structurally typed language, however, there is not a fixed subtype hierarchy. Recall that in order to infer subtype relationships, we first need the specific type instances. We have two choices: either explore the entire call graph to discover all type instantiations and construct our dictionaries according to the call-graph, or construct/supertype our dictionaries at the call-site where specialisation would have happened. The former approach was taken by Go 1.18 and beyond the significant static analysis required, this approach also suffers from the same finiteness limitation encountered by monomorphisation approaches [Griesemer et al. 2020].

We demonstrate our call-site approach in Figure 12. This example consists of two interfaces, `Eq` and `Ord`, which form a subtype relation along with a method `Foo` which uses a type parameter bounded by `Ord` to instantiate a type parameter bounded by `Eq`.

If, in the source program, there are two types σ and τ where there exists an instantiation creating a subtype relation, then the two erased types form a subtype relation. This is precisely the result discussed in § 4.1.1. When initially creating a dictionary, we populate it with the required method pointers for the known instantiating type. If, however, we are creating a τ-dictionary for type parameter β bounded by σ, then the method contained by the supertyping τ-dictionary (`Eq`) is a subset of the σ-dictionary (`Ord`) for type parameter α. Dictionary-supertyping then consists of destructuring the subtype's dictionary and – along with the type-rep – adding all required method pointers to a new supertype-dictionary.

While conceptually simple, our call-site approach directly addresses the unique issues raised by structural typing systems and allows us to overcome the limitation discussed in § 3.3 that afflicts both monomorphisation [Griesemer et al. 2020] and Go 1.18.

4.2 Dictionary-Passing Judgement

This subsection is technical: readers who are not interested in the formal translation rules can safely skip this subsection.

We define the judgement $\vdash P \Longmapsto P^{\ddagger}$ as the dictionary-passing translation from P in FGG to $P^{\ddagger}$ in FG. The expression judgement $\Delta; \eta; \Gamma \vdash e \Longmapsto e^{\ddagger}$ is parametrised by variable and type variable

$$arity(\overline{D}) = \bigcup \overline{arity(D)} \quad arity([\Psi](\overline{x\ \tau})\ \tau) = |\Psi| + |\overline{x}| \quad arity(\textbf{type}\ \tau_I\ \textbf{interface}\ \{\overline{mM}\}) = \overline{arity(M)}$$

$$arity(\textbf{type}\ \tau_S\ \textbf{struct}\ \{\overline{f\ \tau}\}) = \emptyset \quad arity(\textbf{func}\ (\texttt{this}\ \tau_S)\ mM\ \{\textbf{return}\ e\}) = \{arity(M)\}$$

$$maxFormal(\textbf{func}\ (\texttt{this}\ t_S[\overline{\alpha}])\ m[\Psi](\overline{x\ \tau})\ \tau\ \{\textbf{return}\ e\}) = |\Psi|$$

$$maxFormal(\textbf{type}\ t_I[\Phi]\ \textbf{interface}\ \{\overline{m[\Psi](\overline{x\ \tau})\ \tau}\}) = \max(|\overline{\Psi}|, |\Phi|) \quad maxFormal(\textbf{type}\ t_S[\Phi]\ \textbf{struct}\ \{\overline{f\ \tau}\}) = |\Phi|$$

$$type\_meta_{\zeta}(\alpha) = \zeta(\alpha) \quad type\_meta_{\zeta}(t[\overline{\tau}]) = mdata\_name(t)\{\overline{type\_meta_{\zeta}(\tau)}\} \quad asParam(\alpha\ t_I[\phi]) = \overline{\texttt{dict}\ typeDict(t_I)}$$

$$\frac{n = arity(M) \quad m^{\ddagger} = spec\_name(m)}{spec\_mdata(mM) = m^{\ddagger}()\ \texttt{spec\_mdata}_n}$$

$$\frac{\zeta' = \zeta, \{\beta_i \mapsto \texttt{param\_index}_i\{\}\}_i \quad n = arity([\beta\ \sigma_I](\overline{x\ \tau})\ \tau)}{sig\_mdata_{\zeta}([\beta\ \sigma_I](\overline{x\ \tau})\ \tau) = \texttt{spec\_mdata}_n\{\overline{type\_meta_{\zeta'}(\sigma_I)}, \overline{type\_meta_{\zeta'}(\tau)}, type\_meta_{\zeta'}(\tau)\}}$$

$$\frac{\overline{\texttt{dict}\ u} = asParam(\Psi)}{meth\_ptr(t, m[\Psi](\overline{x\ \tau})\ \tau) = \left\{ \begin{array}{l} \textbf{type}\ mName(t,m)\ \textbf{struct}\{\}; \\ \textbf{func}(x\ \ mName(t,m))\ \texttt{Apply(rec Any,}\ \overline{\texttt{dict Any}}, \overline{x\ \texttt{Any}})\ \texttt{Any}\{\textbf{return}\ \texttt{rec.}(t).m(\overline{\texttt{dict.}(u)},\ \overline{x})\ \} \end{array} \right\}$$

$$\frac{\Delta \vdash t_I[\phi] <: \alpha \quad \overline{mM} = methods_{\Delta}(t[\phi]) \quad t_S = typeDict(t_I)}{makeDict_{\eta,\Delta}(\alpha, t_I[\phi]) = t_S\{\overline{\eta(\alpha).m},\ \eta(\alpha).\_type\}} \qquad \frac{\Delta(\alpha) = \tau_I}{makeDict_{\eta,\Delta}(\alpha, \tau_I) = \eta(\alpha)}$$

$$makeDict_{\eta,\Delta}(\overline{\tau}, \overline{\alpha\ \sigma}) = \overline{makeDict_{\eta,\Delta}(\tau, \sigma)}$$

$$\frac{\overline{m[\Psi](\overline{x\ \tau})\ \sigma} = methods_{\Delta}(u_I[\psi]) \quad t_S = typeDict(u_I) \quad \zeta = (-.\_type) \circ \eta \quad meta = type\_meta_{\zeta}(t[\phi])}{makeDict_{\eta,\Delta}(t[\phi], u_I[\psi]) = t_S\{\overline{mName(t,m)},\ meta\}}$$

Fig. 13. Dictionary-passing auxiliary function

environments (Γ and Δ *resp.*) as well as a dictionary map η from type variable names to dictionary variables. We provide auxiliary functions in Figure 13 and translation rules in Figure 14.

Name constants. We introduce a set of maps from name constants in FGG to unique FG names which are assumed to never produce a collision, (1) $typeDict(t_I)$ — from a type bound (interface) to the dictionary struct name for that bound; (2) $mdata\_name(t)$ — from a type name to a simulated type name; (3) $spec\_name(m)$ — from a method name to a method producing simulated specification; and (4) $mName(t, m)$ — the method applicator (pointer) for method m on type t.

Auxilary functions. Figure 13 provides a number of auxiliary functions used in the dictionary-passing translation. The overloaded $arity()$ function computes the number of type and value parameters required by each method signature, including method signatures in an interface's specifications. Function $maxFormal(D)$ computes the number of type parameters expected by the largest type formal. Function $asParam(\Phi)$ converts a type formal into dictionary arguments. The function $meth\_ptr(t, mM)$ constructs the simulated method pointer struct and implementation – called the *abstractor/applicator pair* – for method m on type t.

To build a type simulation of type τ we call $type\_meta_{\zeta}(\tau)$ where ζ is a map from type variables to existing simulated types. When simulating the type assertion to an interface in § 4.1.3, we used the $spec\_name(m)$ method `spec_do` to produce the instantiated simulated signature for method m. The $spec\_mdata(mM)$ function takes an interface's method specification and produces the specification for the $spec\_name(m)$ method. Simulated method signatures are built using $sig\_mdata_{\zeta}(M)$. This function takes a map ζ from type variables to simulated types, and extends ζ with the indexing structs for the method's type formal.

A set of simulated method signature ($\texttt{spec\_mdata}_n$) and type parameter index ($\texttt{param\_index}_i$) structs are created by the program translation rule ([d-program]); $arity(\overline{D})$ and $maxFormal(D)$ are used to ensure that all needed structs are constructed. $\texttt{spec\_mdata}_n$ is an $n{+}1$ tuple used in interface assertion simulation, and describes a method signature of arity n and gives type parameter bounds,

value argument types, and the method's return type. To allow interface assertion simulation to reference type variables, we use $\texttt{param\_index}_i\{\}$ to reference a method's i^{th} type parameter.

Given a type environment Δ and a map η from type variables to existing dictionaries, we build a τ_I-dictionary for type σ using the $makeDict_{\eta,\Delta}(\sigma, \tau_I)$ function. In the case that σ is already a type variable α, then the map η must contain a dictionary for α. When α is bounded by τ_I in δ, we are done, whereas if τ_I is a subtype of α, but not $\tau_I = \alpha$, then we need copy method pointers required by the new (and smaller) τ_I-dictionary. A new dictionary is built for a constant type σ by providing a method pointer (abstractor) for each method specified by τ_I and the simulated type of σ.

$$\text{[d-program]}\quad \frac{\begin{array}{c} fns = \{\textbf{type}\ \texttt{Function}_n\ \textbf{interface}\ \{\texttt{Apply(rec Any, }\{x_i\ \texttt{Any}\}_{i<n}\texttt{) Any}\}\}_{n\in\overline{n}} \\ \overline{n} = arity(\overline{D}) \quad metas = \{\textbf{type}\ \texttt{spec\_mdata}_n\ \textbf{struct}\ \{\{\texttt{\_type}_i\ \texttt{\_type\_mdata}\}_{i\le n}\}\}_{n\in\overline{n}} \quad m = \max \bigcup \overline{maxFormal(D)} \\ params = \{\textbf{type}\ \texttt{param\_index}_i\ \textbf{struct}\ \{\}\}_{i<m} \quad typeMeta = \{\textbf{type}\ \texttt{\_type\_mdata}\ \textbf{interface}\ \{\texttt{tryCast}(x\ \texttt{Any})\ \texttt{Any}\}\} \\ \vdash \overline{D} \Rightarrow \overline{\mathcal{D}} \quad \overline{D^\ddagger} = \{\textbf{type}\ \texttt{Any}\ \textbf{interface}\ \{\}\} \cup params \cup typeMeta \cup metas \cup fns \cup \bigcup \overline{\mathcal{D}} \quad \emptyset;\emptyset;\emptyset \vdash e \Rightarrow e^\ddagger \end{array}}{\vdash \overline{D} \triangleright e \Rightarrow \overline{D^\ddagger} \triangleright e^\ddagger}$$

$$\text{[d-interface]}\quad \frac{\begin{array}{c} \vdash \overline{S \Rightarrow S^\ddagger} \quad \vdash \overline{S \Rightarrow_{\text{dict}} S_{\text{dict}}} \quad \zeta = \{\overline{\alpha \mapsto \texttt{this.\_type}}\} \\ \overline{assertions} = \{\textbf{if}(x.(t_I).\ spec\_name(m)()\ !=\ sig\_mdata_{\zeta}(M))\ \{\ \textbf{panic}\ \}\ |\ mM \in \overline{S}\} \end{array}}{\vdash \textbf{type}\ t_I\,[\overline{\alpha\ \tau_I}]\ \textbf{interface}\ \{\overline{S}\} \Rightarrow \left\{\begin{array}{l} \textbf{type}\ t_I\ \textbf{interface}\ \{\overline{S^\ddagger},\ \overline{spec\_mdata(S)}\} \\ \textbf{type}\ typeDict(t_I)\ \textbf{struct}\ \{\overline{S_{\text{dict}}},\ \texttt{\_type \_type\_mdata}\} \\ \textbf{type}\ mdata\_name(t_I)\ \textbf{struct}\ \{\{\texttt{\_type}_i\ \texttt{\_type\_mdata}\}_{i<|\alpha|}\} \\ \overline{\textbf{func}\ (\texttt{this}\ mdata\_name(t_I))\ \texttt{tryCast}(x\ \texttt{Any})\ \texttt{Any}\{\overline{assertions}\ ;\ \textbf{return}\ x\}} \\ \overline{meth\_ptr(t, S)} \end{array}\right\}}$$

$$\text{[d-meth]}\quad \frac{\begin{array}{c} (\textbf{type}\ \ t_S\,[\overline{\alpha\ \tau_I}]\ T) \in \overline{D} \quad \eta = \overline{\alpha \mapsto \texttt{this.dict}}, \overline{\beta \mapsto \texttt{dict}} \quad \Psi^\ddagger = asParam(\beta\ \sigma_I) \\ \overline{\alpha\ \tau_I}, \overline{\beta\ \sigma_I}; \eta; \texttt{this} : t_S\,[\overline{\alpha}], \overline{x : \tau} \vdash e \Rightarrow e^\ddagger \quad \zeta = \{\alpha_i \mapsto \texttt{this.dict}_i\texttt{.\_type}\}_i \quad m^\ddagger M^\ddagger = spec\_mdata(m[\beta\ \sigma_I](\overline{x\ \tau})\ \tau) \end{array}}{\vdash \textbf{func}\ (\texttt{this}\ t_S[\overline{\alpha}])\ m[\beta\ \sigma_I](\overline{x\ \tau})\ \tau\ \{\textbf{return}\ e\} \Rightarrow \left\{\begin{array}{l} \textbf{func}\ (\texttt{this}\ t_S)\ m(\Psi^\ddagger, \overline{x\ \texttt{Any}})\ \texttt{Any}\ \{\textbf{return}\ e^\ddagger\} \\ \textbf{func}\ (\texttt{this}\ t_S)\ m^\ddagger M^\ddagger\ \{\textbf{return}\ sig\_mdata_{\zeta}([\beta\ \sigma_I](\overline{x\ \tau})\ \tau) \\ meth\_ptr(t, m[\beta\ \sigma_I](\overline{x\ \tau})\tau) \end{array}\right\}}$$

$$\text{[d-struct]}\quad \frac{\begin{array}{c} \overline{\texttt{dict}\ u} = asParam(\Phi) \quad n = |\Phi| \quad \overline{assertions} = \{\textbf{if}\ \texttt{this.\_type}_i\ !=\ x.(t_S)\texttt{.dict}_i\texttt{.\_type}\ \{\ \textbf{panic}\ \}\}_{i<n} \end{array}}{\vdash \textbf{type}\ t_S\,[\Phi]\ \textbf{struct}\,\{\overline{f\ \tau}\} \Rightarrow \left\{\begin{array}{l} \textbf{type}\ t_S\ \textbf{struct}\,\{\overline{f\ \texttt{Any}}, \overline{\texttt{dict}\ u}\} \\ \textbf{type}\ \ mdata\_name(t_S)\ \textbf{struct}\,\{\{\texttt{\_type}_i\ \texttt{\_type\_mdata}\}_{i<n}\} \\ \textbf{func}\ (\texttt{this}\ \ mdata\_name(t_S))\ \texttt{tryCast}(x\ \texttt{Any})\ \texttt{Any}\{x.(t_S)\ ;\ \overline{assertions}\ ;\ \textbf{return}\ x\} \end{array}\right\}}$$

$$\text{[d-field]}\quad \frac{\Delta;\eta;\Gamma \vdash e \Rightarrow e^\ddagger \quad \Delta;\Gamma \vdash e : t_S[\phi]}{\Delta;\eta;\Gamma \vdash e.f \Rightarrow e^\ddagger.(t_S).f}$$

$$\text{[d-value]}\quad \frac{(\textbf{type}\ t_S\,[\Phi]\ T) \in \overline{D} \quad \phi^\ddagger = makeDict_{\eta,\Delta}(\phi, \Phi) \quad \Delta;\eta;\Gamma \vdash \overline{e \Rightarrow e^\ddagger}}{\Delta;\eta;\Gamma \vdash t_S[\phi]\{\overline{e}\} \Rightarrow t_S\{\overline{e^\ddagger}, \phi^\ddagger\}}$$

$$\text{[d-assert]}\quad \frac{\Delta;\eta;\Gamma \vdash e \Rightarrow e^\ddagger \quad \zeta = (\texttt{-.\_type}) \circ \eta}{\Delta;\eta;\Gamma \vdash e.(\tau) \Rightarrow type\_meta_{\zeta}(\tau).\texttt{tryCast}(e^\ddagger)}$$

$$\text{[d-dictcall]}\quad \frac{\Delta;\Gamma \vdash e : \alpha \quad (m[\Psi](\overline{x\ \tau})\ \tau) \in methods_\Delta(\alpha) \quad \psi^\ddagger = makeDict_{\eta,\Delta}(\psi, \Psi) \quad \Delta;\eta;\Gamma \vdash e \Rightarrow e^\ddagger \quad \Delta;\eta;\Gamma \vdash \overline{e \Rightarrow e^\ddagger}}{\Delta;\eta;\Gamma \vdash e.m[\psi](\overline{e}) \Rightarrow \eta(\alpha).m.\texttt{Apply}(e^\ddagger, \psi^\ddagger, \overline{e^\ddagger})}$$

$$\text{[d-call]}\quad \frac{\Delta;\Gamma \vdash e : t[\phi] \quad (m[\Psi](\overline{x\ \tau})\ \tau) \in methods_\Delta(t[\phi]) \quad \psi^\ddagger = makeDict_{\eta,\Delta}(\psi, \Psi) \quad \Delta;\eta;\Gamma \vdash e \Rightarrow e^\ddagger \quad \Delta;\eta;\Gamma \vdash \overline{e \Rightarrow e^\ddagger}}{\Delta;\eta;\Gamma \vdash e.m[\psi](\overline{e}) \Rightarrow e^\ddagger.(t).m(\psi^\ddagger, \overline{e^\ddagger})}$$

$$\text{[d-spec]}\quad \frac{\Psi^\ddagger = asParam(\Psi)}{\vdash m[\Psi](\overline{x\ \tau})\ \tau \Rightarrow \quad m(\Psi^\ddagger, \overline{x\ \texttt{Any}})\ \texttt{Any}}$$

$$\text{[d-dict]}\quad \frac{n = |\Phi| + |\overline{x}|}{\vdash m[\Psi](\overline{x\ \tau})\ \tau \Rightarrow_{\text{dict}} m\ \texttt{Function}_n}$$

$$\text{[d-var]}\quad \frac{}{\Delta;\eta;\Gamma \vdash x \Rightarrow x}$$

Fig. 14. Dictionary-passing translation

Program translation. Rule [d-program] introduces new declarations required for method pointers and type simulations as described in § 4.1, and the `Any` interface to provide a uniform, erased, type representation. Each method applicator must implement an n-arity function interface $\texttt{Function}_n$, that accepts the receiver and the n arguments for the desired method call. The arity of a method includes both the regular value arguments as well as the dictionary arguments. A simulated type implements the `_type_mdata` interface by providing an assertion simulation method (`tryCast`), which panics if the assertion is invalid. The $\texttt{spec\_mdata}_n$ and $\texttt{param\_index}_i$ structs are created as

required by the $arity(\overline{D})$ and $maxFormal(D)$ functions, respectively. Each declaration is translated to multiple declarations; we use $\mathcal{D}$ to indicate this.

Interface and dictionary construction. The translation of interfaces produces a number of FG declarations ([d-interface]). They are (1) an FG interface, and (2) a dictionary for that interface.

The interface $t_I[\Phi]$ becomes the erased type t_I (1). For each method specification S defined by the source interface, we produce two specifications in the target; the first is defined by [d-spec] and replaces types formal with appropriate dictionaries while erasing all other types, and the second defines a method producing the simulated FGG method specification. Since [d-meth] produces such a simulated specification method for each method, it is guaranteed that any type which implements the former will implement the latter.

The dictionary (2) for an interface t_I is given by a new struct $typeDict(t_I)$, which contains a method pointer (abstractor) for each specified method and the simulated type (`_type`) for the type parameter's specialising type. Type simulation is also defined here. For type $t_I[\Phi]$, the simulation struct ($mdata\_name(t_I)$) contains a field for each type parameter in Φ. The `tryCast` method checks that each specified method is implemented correctly by the target of the assertion (See § 4.1.3). For clarity of presentation, we assume a number of extra language features that can be easily implemented in FG, including; if-statement, struct inequality, explicit panic, and sequencing [Griesemer et al. 2020].

Struct declaration. To translate $t_S[\Phi]$, we erase all field types and add a new dictionary field for each type parameter in Φ. The simulated type $mdata\_name(t_S)$ is constructed with a variable for each type parameter, and `tryCast` checks that the target value is exactly the assertion type.

Method declaration. Judgement on method $m[\Psi](\overline{x\ \tau})\ \tau$ ([d-meth]) produces a primary method, a method returning the simulated method type, and an abstractor/applicator pair. The primary method and simulation method's types match those from [d-interface]. The body of the implementing method is translated in the $\Delta; \eta; \Gamma$ environments, where Δ and Γ are built according to the typing system. There are two locations for type variables – and thus dictionaries – to be passed into a method, namely in the receiver or as an argument; consequently, η may map into either a dictionary argument ($\texttt{dict}_i$) or a receiver's dictionary field ($\texttt{this.dict}_i$).

Expressions. The struct literal $(t_S[\phi]\{\overline{e}\})$ is translated by first translating each field assignment and then building an appropriate dictionary for each type in ϕ using *makeDict* ([d-value]).

Method calls are translated in one of two ways. The first ([d-call]) is the immediate structural translation of sub terms and creation of appropriate dictionaries; this translation is only possible if the type of the receiver is not a type variable, although it does not need to be a closed type. The second ([d-dictcall]) translates arguments and creates dictionaries in the same way as the former, but needs to resolve the method implementation using a dictionary lookup.

5 CORRECTNESS OF DICTIONARY-PASSING TRANSLATION

In this section, we define, justify, and prove the correctness of our dictionary-passing translation using a behavioural equivalence. We first introduce a general *correctness criteria* which good translations should satisfy. We then propose a novel *bisimulation up to* technique to prove that translated programs are behaviourally equivalent to their source program. We use this result to prove the correctness of our dictionary-passing translation. Full proofs can be found in the full version of this paper [Ellis et al. 2022d].

5.1 Correctness Criteria

The correctness criteria is defined using a number of preliminary predicates provided below.

Definition 5.1 (Type assertion errors). We say expression e in FG is a *type assertion error* (*panic* in [Griesemer et al. 2020]) if there exists an evaluation context E, value v, and type t such that $e = E[v.(t)]$ and $type(v) \not<: t$. We say expression e gets a *type assertion error* (denoted by $e \Downarrow_{\text{panic}}$) if it reduces to an expression that contains a type assertion error, *i.e.,* $e \longrightarrow^* e'$ and e' is a type assertion error. We write $P \Downarrow_{\text{panic}}$ when $P = \overline{D} \triangleright e$ and $e \Downarrow_{\text{panic}}$. Similarly, we define $e \Downarrow_{\text{panic}}$ and $P \Downarrow_{\text{panic}}$ for FGG.

We write $e \Downarrow v$ if there exists v such that $e \longrightarrow^* v$ and extend this predicate to P. We abbreviate $\emptyset;\emptyset;\emptyset \vdash e \Longrightarrow e^\ddagger$ to $\vdash e \Longrightarrow e^\ddagger$.

We define the following general correctness criteria related to typability, error correctness, and preservation of a program's final result.

Definition 5.2 (Preservation properties). Let P *ok* in FGG, and let there exist $P^\ddagger$ such that $\vdash P \Longrightarrow P^\ddagger$. A translation is:

(1) ***type preserving***: if P *ok*, then $P^\ddagger$ *ok*.
(2) ***type assertion error preserving***: $P \Downarrow_{\text{panic}}$ iff $P^\ddagger \Downarrow_{\text{panic}}$.
(3) ***value preserving***: $P \Downarrow v$ iff $P^\ddagger \Downarrow v^\ddagger$ with $\vdash v \Longrightarrow v^\ddagger$.

We only require the left-to-right direction for type preservation, as due to type erasure (§ 4.1.1), we cannot obtain the right-to-left direction for dictionary-passing. Our type preservation criteria matches that defined in Griesemer et al. [2020, Theorem 5.3]. We can, however, show that type assertions are precisely simulated (§ 4.1.3).

5.2 Behavioural Equivalence – Bisimulation up to Dictionary Resolution

Griesemer et al. [2020, Theorem 5.4] prove the correctness of the monomorphism translation using a simple (strong) bisimulation: the binary relation $\mathfrak{R}$ is a *bisimulation* iff for every pair of $\langle e, d\rangle$ in $\mathfrak{R}$, where e is a FGG expression and d is a FG expression: (1) if $e \longrightarrow e'$, then $d \longrightarrow d'$ such that $\langle e', d'\rangle \in \mathfrak{R}$; and (2) if $d \longrightarrow d'$, then $e \longrightarrow e'$ such that $\langle e', d'\rangle \in \mathfrak{R}$. This strong bisimulation suffices for translations that preserve a simple one-to-one reduction-step correspondence.

Unlike monomorphisation, dictionary-passing relies on runtime computation, which prevents such a simple correspondence. We can, however, distinguish between reductions introduced by dictionary-passing and those inherited from the source program. This distinction allows us to construct a one-to-one correspondence relation *up to* dictionary resolution. The formulation is non-trivial since, in FG, dictionary resolution can occur at any point in a subterm. We demonstrate this issue by evaluating the example in Figure 15. Importantly, the translated function `foo` cannot resolve the generic `Add` method from dictionary `dict` until *after* expression `bar(···)` is fully evaluated.

```
1 func foo[α Num](a α) α {
2   return a.Add(bar(···))
3 }
4 func main() {
5   foo[Int](Zero{})
6 }
```

```
1 func foo(dict NumDict, a Any) Any {
2   return dict.Add.Apply(a, bar(···)) }
3 type Int_Add struct {} // method pointer
4 func (i Int_Add) Apply(this Any, a Any) Any {
5   return this.(Int).Add(a) }
6 func main() {
7   foo(NumDict{Int_Add{}}, Zero{}) }
```

Fig. 15. Non-trivial dictionary example. Source (Left). Translation (Right)

After one step, the FGG program (left) is `Zero{}.Add(bar(···))`. If we translate the afore reduced term, we get `Zero{}.(Zero).Add(bar(···))` ($Q_0^\ddagger$). But reducing the translated FG program (right), we obtain the `NumDict{Int_Add{}}.Add.Apply(Zero{}, bar(···))` ($Q_1^\ddagger$).

To show $Q_0^\ddagger$ equivalent to $Q_1^\ddagger$ using the standard FG reduction, we would first have to fully resolve `bar(···)` before we could start to the resolve dictionary access in $Q_1^\ddagger$. We might attempt to show that the translation in Figure 15 is correct using a many-to-many reduction-step relation, *i.e.,* some binary relation $\mathfrak{R}$ where for every pair of $\langle e, d\rangle$ in $\mathfrak{R}$ it holds that (1) if $e \longrightarrow^* e'$, then

$d \longrightarrow^* d'$ such that $\langle e', d' \rangle \in \mathfrak{R}$; and (2) if $d \longrightarrow^* d'$, then $e \longrightarrow^* e'$ such that $\langle e', d' \rangle \in \mathfrak{R}$. This approach is both complicated by the presence of non-termination, *e.g.,* if `bar(···)` does not return a value, then we could never show that $Q_0^\ddagger$ and $Q_1^\ddagger$ are related. And more importantly, many-to-many relationships give less information about the nature of a translation than one-to-one relationships.

Were we to consider just the `NumDict{Int_Add{}}.Add.Apply(···)` portion of $Q_1^\ddagger$ we observe that using a pre-congruence reduction $Q_1^\ddagger$ resolves to `Zero{}.(Int).Add(bar(···))`. We may then safely increase the accuracy of the assertion `Zero{}.(Int)` to `Zero{}.(Zero)` without altering the semantics of the term. The later step is required because while the dictionary stored the information that `Zero{}` was passed to `foo` as type `Int`, the reduction of the FGG term forgot this information. We call these two steps *dictionary resolution*, as they resolve only those computations introduced by the use of dictionaries for method resolution. $Q_0^\ddagger$ is equivalent to $Q_1^\ddagger$ *up to dictionary resolution.* Our translation also adds type simulation computations and type assertions. Unlike dictionary resolution, these extra computation steps are subsumed by the standard FG reduction.

Definition 5.3 (Dictionary resolution). We define three pattern sets in FG: ρ_{erase} (type assertions as a result of erasure), ρ_{sim} (type assertion simulation), and ρ_{dict} (dictionary resolution):

$$\rho_{\text{erase}} ::= \{\ v.(t)\ \}$$
$$\rho_{\text{sim}} ::= \{\ v.\_\texttt{type}_i,\ v.\_\texttt{type},\ v.(t),\ v.\,spec\_name(m)(),\ v.\texttt{dict}_i,\ \textbf{if}\ v\,!=v\ \{\,\textbf{panic}\,\},\ \textbf{return}\ v\ \}$$
$$\rho_{\text{dict}} ::= \left\{ \begin{array}{l} typeDict(t)\{\overline{v}\}.f,\ v.\texttt{dict}_i,\ v.\_\texttt{type}_i,\ v.\_\texttt{type}, \\ mName(t,m)\{\}.\texttt{Apply}(\overline{e}),\ typeDict(t)\{\overline{v}\}.(typeDict(t)) \end{array} \right\}$$

From these patterns, we define a number of reductions. We define the first of these as $E[e] \longrightarrow_{\text{e}} E[e']$ if $e \longrightarrow e'$ with $e \in \rho_{\text{erase}}$; and $E[e] \longrightarrow_{\text{s}} E[e']$ if $e \longrightarrow e'$ with $e \in \rho_{\text{sim}}$. We write $d \Longrightarrow d'$ if $d \longrightarrow_{\text{e}}^* \longrightarrow \longrightarrow_{\text{s}}^* d' \not\longrightarrow_{\text{s}}$.

Let C be the context: $C ::= \square \mid C.f \mid C.(t) \mid t_S\{\overline{e}, C, \overline{e}'\} \mid C.m(\overline{e}) \mid e.m(\overline{e}, C, \overline{e}')$. We define the dictionary resolution reduction $\twoheadrightarrow$ as (1) $C[e] \twoheadrightarrow C[e']$ if $e \longrightarrow e'$ where $e \in \rho_{\text{dict}}$; and (2) $C[e.(t)] \twoheadrightarrow C[e.(u)]$ if $\vdash e : u$ and $u <: t$.

Notice that if $e \Longrightarrow e'$, then $e \longrightarrow^+ e'$; and that $\Longrightarrow$ can be viewed as a one-step reduction which corresponds to a one-step of the source language. Reduction $\longrightarrow_{\text{s}}$ only occurs following a call to `tryCast`, and simulates whether or not the source FGG assertion is a type assertion error (See § 4.1.3). The reduction $\longrightarrow_{\text{e}}$ resolves only the assertions introduced during the type erasure step (See § 4.1.1). The dictionary resolution reduction ρ_{dict} will occur following a method call [r-call] and simulates the type parameter specialisation. As demonstrated in the above example, the $\twoheadrightarrow$ reduction may reduce any subterm matching ρ_{dict} or refine any type assertion.

Lemma 5.1. *Let e be an FG expression. Assume $\emptyset \vdash e : u$.*

(1) $\Longrightarrow$ is deterministic, i.e., *if $e \Longrightarrow e_1$ and $e \Longrightarrow e_2$, then $e_1 = e_2$.*
(2) $\twoheadrightarrow$ is confluent, i.e., *if $e \twoheadrightarrow e_1$ and $e \twoheadrightarrow e_2$, then there exists e' such that $e_1 \twoheadrightarrow e'$ and $e_2 \twoheadrightarrow e'$.*

We now extend the bisimulation relation to bisimulation up to dictionary resolution.

Definition 5.4 (Bisimulation up to dictionary resolution).

The relation $\mathfrak{R}$ is a *bisimulation up to dictionary resolution* if $\mathfrak{R} \cdot (\twoheadleftarrow)^*$ is a bisimulation, *i.e.,* if P *ok* in FGG and $\vdash P \rightrightarrows P^\ddagger$ where $P = \overline{D} \triangleright e$ and $P^\ddagger = \overline{D^\ddagger} \triangleright e^\ddagger$ then the diagram (right) commutes.

$$\begin{array}{ccccc} e & \overset{\pi_1}{\cdots\!\!\prec} & \langle e, e^\ddagger \rangle \in \mathfrak{R} & \overset{\pi_2}{\cdots\!\!\succ} & e^\ddagger \\ \downarrow & & & & \Downarrow \\ d & \overset{\pi_1}{\cdots\!\!\prec} & \langle d, d^\ddagger \rangle \in \mathfrak{R} & \overset{\pi_2}{\cdots\!\!\succ} & d^\ddagger \overset{*}{\longleftarrow} d' \end{array}$$

Intuitively, our translation forms a bisimulation up to dictionary resolution if (1) each step that the source program takes can be mimicked by the translated program; and (2) conversely, that if the translated program reduces, then the source program must have been able to make an equivalent

step – albeit with the translated program still needing to evaluate the added dictionary resolution computations at some future point during computation.

By considering the observable behaviour of a program to be non-dictionary resolution reduction steps, type assertion errors, and termination (value production), we ensure that the translated program is behaviourally equivalent to that of the source program. Note that this formulation may be extended to a concurrent or effectful fragment of Go with the standard addition of *barbs* [Milner and Sangiorgi 1992] or transition labels.

Finally, we arrive at our main theorem — that the translation satisfies the correctness criteria.

Theorem 5.1 (Correctness of dictionary-passing). *Let P ok in FGG and $\vdash P \Longmapsto P^\ddagger$ with $P = \overline{D} \triangleright e$ and $P^\ddagger = \overline{D^\ddagger} \triangleright e^\ddagger$.(1) Dictionary-passing translation $(-)^\ddagger$ is type preserving; (2) e and $e^\ddagger$ are bisimilar up to dictionary resolution; (3) $(-)^\ddagger$ is type assertion error preserving; and (4) $(-)^\ddagger$ is value preserving.*

Theorem 5.1 states that our translation is correct, as translated programs behave exactly as the source program would have behaved, and that any extra computations are accounted for by machinery introduced for dictionary-passing.

It is worth stressing that our statement is *stronger* than the various definitions of dictionary-passing translation correctness considered in the literature (see § 7), which limit themselves to non-termination preserving versions of value preservation. By providing an account of intermediate state equivalence, Theorem 5.1(2) not only gives a meaningful equivalence for non-terminating programs, but may also be extended to languages with non-determinism or concurrency.

5.3 Proof of Theorem 5.1

We provide the key lemmata, theorems, and corollaries used in the proof of Theorem 5.1. All omitted proofs may be found in the full version of this paper [Ellis et al. 2022d].

Type preservation. The type preservation criteria given in Definition 5.2 only considers whole programs. We must first show that the dictionary-passing translation is type preserving for expressions. Note that the translation of structure literals is the only non-Any typed expression.

Lemma 5.2 (Type preservation of expressions). *Let $\Delta; \eta; \Gamma \vdash e \Longmapsto e^\ddagger$ and $[\![\Gamma]\!]_{\Delta;\eta;\Gamma}$ be the FG environment where all variables in Γ are erased (*Any*) and each dictionary in η is appropriately typed according to the bound in Δ. If $\Delta; \Gamma \vdash e : \tau$ then (1) If $\tau = \alpha$ or τ_I, then $[\![\Gamma]\!]_{\Delta;\eta;\Gamma} \vdash e^\ddagger :$ Any. (2) If $\tau = t_S[\phi]$, then either $[\![\Gamma]\!]_{\Delta;\eta;\Gamma} \vdash e^\ddagger :$ Any or $[\![\Gamma]\!]_{\Delta;\eta;\Gamma} \vdash e^\ddagger : t_S$.*

Corollary 5.1 (Type preservation (Theorem 5.1 (1))). *If P ok, then $P^\ddagger$ ok.*

Proof. By the assumption that name constant functions are distinct and Lemma 5.2. □

Bisimulation and error preservation. The operational correspondence theorem described the behaviour of a source program and its translation as four non-overlapping cases. Note that $e^\ddagger \Longrightarrow e'$ is the maximum reduction without another type assertion simulation reduction ($e' \not\longrightarrow_s$).

Theorem 5.2 (Operational correspondence). *Let P ok where $P = \overline{D} \triangleright e$ and let $\vdash \overline{D} \triangleright e \Longmapsto \overline{D^\ddagger} \triangleright e^\ddagger$.*

(a) If $e \longrightarrow d$, then there exists $d^\ddagger$ such that $\emptyset; \emptyset; \emptyset \vdash d \Longmapsto d^\ddagger$ and $e^\ddagger \Longrightarrow \rightarrow^* d^\ddagger$.

(b) If $e^\ddagger \Longrightarrow e'$ where e is not a type assertion error, then there exists d such that $e \longrightarrow d$ and there exists $d^\ddagger$ such that $\emptyset; \emptyset; \emptyset \vdash d \Longmapsto d^\ddagger$ and $e' \rightarrow^* d^\ddagger$.

(c) If $e^\ddagger \Longrightarrow e'$ where e is a type assertion error, then e' is a type assertion error.

(d) If e is a type assertion error, then there exists an e' such that $e^\ddagger \Longrightarrow e'$ and e' is a type assertion error.

Proof. By induction over the assumed reduction. Full proof is provided in [Ellis et al. 2022d]. □

Corollary 5.2 (Bisimulation up to dictionary resolution (Theorem 5.1 (2))). *Let P ok and $\vdash P \Vdash P^\ddagger$ with $P = \overline{D} \triangleright e$ and $P^\ddagger = \overline{D^\ddagger} \triangleright e^\ddagger$. Then e and $e^\ddagger$ are bisimilar up to dictionary resolution.*

Proof. By Theorem 5.2. Let $\mathfrak{R}$ be the least relation such that all source expressions are paired with their translation. $\mathfrak{R}$ is a bisimulation up to dictionary resolution. Namely, for each element $\langle e, e^\ddagger \rangle \in \mathfrak{R}$, we have that:

(1) If $e \longrightarrow e'$, then by Theorem 5.2 (a) there exists a $\langle e', d \rangle \in \mathfrak{R}$ such that $e^\ddagger \Longrightarrow \twoheadrightarrow^* d$.
(2) If $e^\ddagger \Longrightarrow \twoheadrightarrow^* d$, then by Theorem 5.2 (b) there exists a $\langle e', d \rangle \in \mathfrak{R}$ such that $e \longrightarrow e'$.

□

Corollary 5.3 (Type error preservation (Theorem 5.1 (1))). *Let $\overline{D} \triangleright$ ok and $\vdash P \Vdash P^\ddagger$. $P \Downarrow_{\mathsf{panic}}$ iff $P^\ddagger \Downarrow_{\mathsf{panic}}$.*

Proof. By induction on the reductions in $\Downarrow$. □

Value preservation. Finally, the value preservation property follows dictionary-passing being a bisimulation up to dictionary resolution, as the dictionary resolution steps are eager reductions that can equivalently be delayed until they become standard reductions.

Lemma 5.3 (Reduction rewrite). *Let $e_1 \twoheadrightarrow e_2 \longrightarrow e_3$ where $e_1 = C[d_1]$, $e_2 = C[d_2]$, and $d_1 \longrightarrow d_2$.*

(1) If there exists an E such that $C = E$ then $e_1 \longrightarrow^2 e_3$
(2) If there does not exists an E such that $C = E$ then $e_1 \longrightarrow \twoheadrightarrow e_3$

Lemma 5.4 (Resolution to value). *If $e \twoheadrightarrow v$ then $e \longrightarrow v$.*

Corollary 5.4 (Value preservation (Theorem 5.1 (4))). *Let $\overline{D} \triangleright$ ok and $\vdash P \Vdash P^\ddagger$. $P \Downarrow v$ iff $P^\ddagger \Downarrow v^\ddagger$ where $\vdash v \Vdash v^\ddagger$.*

Proof. By Corollary 5.2 we have the following diagram (where $\mathfrak{R}$ is created by $\Vdash$)

$$\begin{array}{ccccccc} e_1 & \longrightarrow & e_2 & \longrightarrow & e_3 & \longrightarrow \cdots \longrightarrow & v \\ \Downarrow & & \Downarrow & & \Downarrow & & \Downarrow \\ e_1^\ddagger & \Longrightarrow \twoheadrightarrow^* & e_2^\ddagger & \Longrightarrow \twoheadrightarrow^* & e_3^\ddagger & \Longrightarrow \twoheadrightarrow^* \cdots \Longrightarrow \twoheadrightarrow^* & v^\ddagger \end{array}$$

By Lemma 5.3 and 5.4 each dictionary resolution reduction $\twoheadrightarrow$ is either subsumed by $\longrightarrow$ or may be delayed using reduction rewriting until it becomes a $\longrightarrow$ reduction. In other words, since $e_1 \longrightarrow e_2 \longrightarrow \cdots \longrightarrow v$ iff $e_1^\ddagger \Longrightarrow \twoheadrightarrow e_2^\ddagger \Longrightarrow \twoheadrightarrow \cdots \Longrightarrow \twoheadrightarrow v^\ddagger$. We use that $\twoheadrightarrow$ can be delayed ($d \twoheadrightarrow\!\!\Longrightarrow d'$ implies $d \Longrightarrow \twoheadrightarrow d$ or $d \longrightarrow\!\!\Longrightarrow d$), hence $e_1^\ddagger \Longrightarrow^+ \twoheadrightarrow^+ v^\ddagger$. Finally, from $e \twoheadrightarrow^+ v$ implies $e \longrightarrow^+ v$, we have that $e_1 \Downarrow v$ iff $e_1^\ddagger \Downarrow v^\ddagger$.

□

Proof of Theorem 5.1 is given by Corollary 5.1, 5.2, 5.3, and 5.4.

6 IMPLEMENTATION AND EVALUATION

Beside the Dictionary-Passing Translation (`dict`), we also implement an Erasure Translation (`erasure`). We compare the two implementations with three existing translators: Monomorphisation Translation (`mono`) [Griesemer et al. 2020], `go2go` (the initial prototype based on a source-to-source monomorphisation), and Go 1.18 [Randall 2022] (the official generic type implementation released on 15th March 2022). This section first discusses the two implementations, then describes the evaluation methodology, before finally presenting the evaluation results.

6.1 Implementation of Dictionary-Passing Translation

We implement the dictionary-passing translator (`dict`) and the erasure-based translator (`erasure`) based on the FGG artifact [Hu 2021] in Go 1.16. We have fully tested the implementations using designed unit tests. Figure 16 shows the code coverage difference across the five translators.

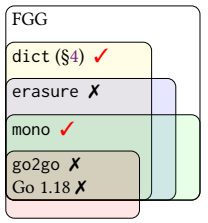

Fig. 16. Relationship of implementations. ✓ denotes a translation proven correct (Theorem 5.1); ✗ denotes a translation that has not.

FGG is the calculus presented in [Griesemer et al. 2020]; `dict` does not cover receiver type formal subtyping; `erasure` does not cover FGG type assertions; `mono` does not cover a class of recursive (*nomono*) programs [Griesemer et al. 2020]; `go2go` is a source-to-source monomorphisation translator implemented by the Go Team, and does not cover F-bounded polymorphism, method parametrisation, receiver type formal subtyping, or recursive (*nomono*) programs; and Go 1.18 is the official release with generics and has the same limitations as `go2go`. Both Go 1.18 and `go2go` target the full Go language, including features not considered by FGG.

We implement `dict` following the rules in § 4. Rather than strictly follow the formalisations of FG and `dict` translation, we leverage the first-class functions support in Go and use function types [The Go Team 2021a] as dictionary fields, similar to using function pointers in C/C++. We also ignore unnecessary type assertions in [d-field] and [d-call] when the translation is not on an interface. We memorise expression typing results to accelerate compilation. We exclude type simulation (§ 4.1.3) of non-generic types (*i.e.,* the size of the type formal is zero), and directly use type assertion for [d-assert] for better runtime performance. We also find if those type metadata are used, and remove them when possible. Users can also disable all type metadata copies if there are no type assertions in the input program. In total, `dict` contains 1160 lines of Go code.

`erasure` is an alternative homogeneous translation implementation from FGG. This implementation erases generic type information and uses the underlying interface type, similar to the erasure implementations for Java [Igarashi et al. 1999; Odersky et al. 2000]. When calling a method, the erased object is directly used as the receiver (If $\Delta, \alpha{:}t_I[\phi]; \Gamma \vdash e : \alpha$ then $\Delta, \alpha{:}t_I[\phi]; \Gamma \vdash e.m[\psi](\overline{e}) \Longrightarrow e^{\ddagger}.(t_I).m(\overline{e^{\ddagger}})$), in contrast to `dict`'s dictionary lookup ([d-dictcall]). For example, `func f[a Foo](x a) {x.Bar()}` translates to `func f(x Any) {x.(Foo).Bar()}`, while `dict` calls the corresponding function in a dictionary field. As in § 4.1.1, naively erasing type parameters breaks type assertion preservation (Definition 5.2). An example of `erasure` is provided in the full version of this paper [Ellis et al. 2022d]. Compared with `dict`, `erasure` provides a concise translation of generics that is fully based on Go's existing dynamic dispatch mechanism. When calling a method of a generic object as though it were an interface, the Go runtime looks up the actual method to call from a list of methods [Clement Rey 2018; Russ Cox 2009], while `dict` finds the actual method from the dictionary. The implementation of `erasure` contains 765 lines of Go code.

6.2 Evaluation Methodology

Benchmarks. We build two benchmark suites to conduct head-to-head comparisons for the five translators. *1) Micro Benchmarks:* we design five micro benchmark sets. Each has a configuration parameter to demonstrate how the translated code scales with a particular aspect of FGG/Go programs. *2) Real-World Benchmarks:* we reimplement all benchmarks in previous papers about generics in Java and Scala [Odersky et al. 2000; Ureche et al. 2013]. Go 1.18 officially released generics on March 15th, 2022, and it is infeasible for us to find usage of generics in real Go programs. The

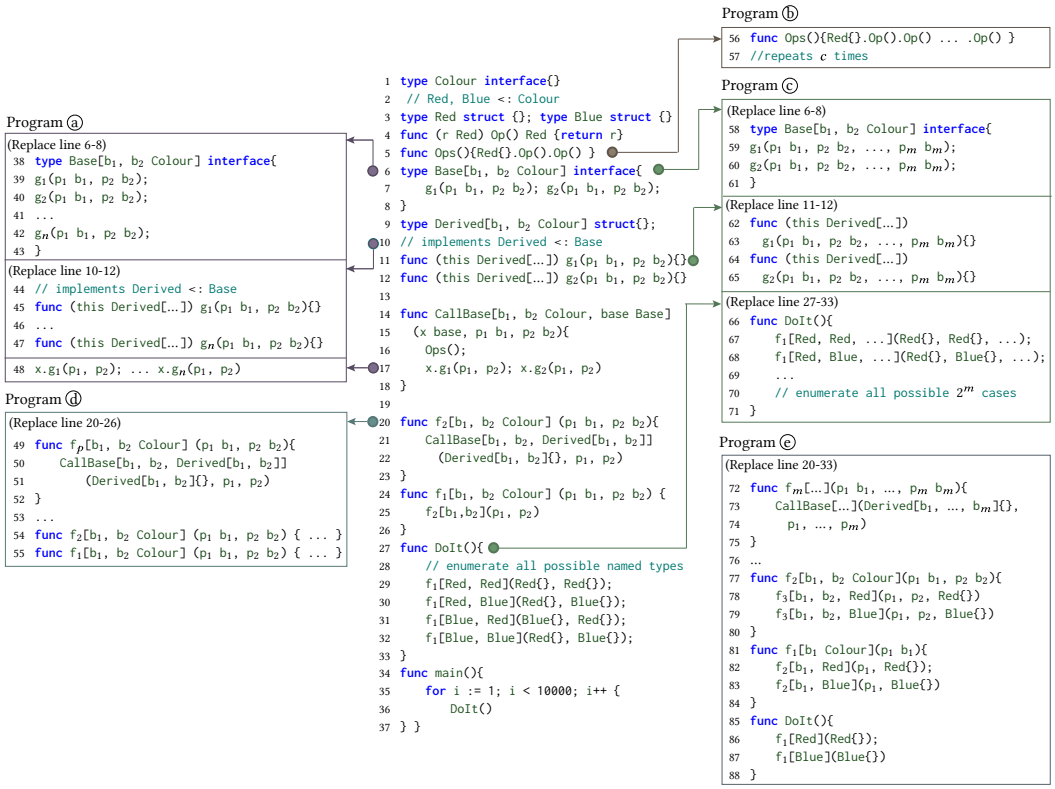

Fig. 17. The base program and its five variations in the micro benchmarks.

second benchmark suite is a reasonable substitute to reveal how the five translators behave in reality.

Micro Benchmarks. The five sets of micro benchmarks, `Program` ⓐ-ⓔ, are all derived from a base program. Figure 17 shows the base program and how the five benchmark sets are derived from it. In the base program, lines 29–32 enumerate all possible combinations of types actual for `f1()`. Function `f1()` takes two parameters and uses them to call `f2()` on line 20, which in turn calls `CallBase()` on line 14. Function `CallBase()` calls `Ops()` on line 5, which further calls `Op()` twice to represent two non-generic operations. All methods of interface `Base` (`g1()` and `g2()`) are implemented by struct `Derived`, and called on line 17, from receiver variable `x` with generic type `base`. Function `main()` calls `DoIt()` 10,000 times (line 36) to provide stable performance results.

The set of `Program` ⓐ extends the number of methods of `Base` (lines 39–42) and `Derived` (lines 45–47) in the base program, from 2 to n. `Program` ⓑ repeats the non-generic operation c times on line 56, instead of two. In `Program` ⓒ, we increase the number of type parameters from 2 to m (lines 59, 60, 63, and 65), and enumerate all 2^m type actual combinations (lines 67–70). `Program` ⓓ increases the length of the call chain between `Doit()` and `CallBase()` from 2 to p (lines 49–55). `Program` ⓔ is particularly designed to expose the exponential complexity of monomorphisation (lines 72–88). Its configuration parameter m controls both the type parameter number of `Base` (and `Derived`) and the number of functions called in between `DoIt()` and `BaseCall()` along the call chain. For the m functions in between `DoIt()` and `BaseCall()`, we further configure each caller to

call its callee twice, and each callee to have one more parameter than its caller (e.g., function body of f_1 and f_2 on lines 77–84).

Real-World Benchmarks. We reimplement the Java and Scala programs using `go2go`, Go 1.18, and FGG for our evaluation. Since FGG does not support all syntax in the programs, we first use FGG to reimplement as many functionalities as possible. Then, we translate the FGG code to Go and manually insert the missed non-generic functionalities. On the other hand, `go2go` and Go 1.18 support all required syntax, so we use them to reimplement each whole program. We manually test the reimplementations with designed testing inputs and compare their outputs with the original versions in Java or Scala. Our tests achieve 100% code coverage.

The benchmarks' functionalities are explained as follows. `List` [Ureche et al. 2013] is an implementation of a linked list. It supports insert and search operations on the linked list. `ResizableArray` [Ureche et al. 2013] implements a resizable array. It inserts elements into the array, reverses the array, and searches elements in the array. `ListReverse` [Odersky et al. 2000] constructs a linked list and reverses it. It contains two reversing implementations. `VectorReverse` [Odersky et al. 2000] is to reverse an array. Similarly, it implements the reversing functionality in two different ways. `Cell` [Odersky et al. 2000] implements a generic container. `Hashtable` [Odersky et al. 2000] accesses elements in a hash table.

Metrics. We consider *code size*, *execution time*, and *compilation time* as our metrics. For code size, we compile each translated benchmark program into a binary executable and disassemble the executable using objdump [Foundation 2021]. Next, we count the number of assembly instructions compiled from the benchmark program as its code size, while excluding the assembly instructions of linked libraries. To measure execution time, we compile each translated FG program using the Go compiler and compute the average execution time over *ten* runs. We consider the time spent on the source-to-source translation and the compilation from a FG program to an executable as the compilation time for the four source-to-source translators. For Go 1.18, we measure its compilation time directly. We compile each benchmark program with each translator *ten* times and report the average compilation time.

Platform & Configurations. All our experiments are conducted on a desktop machine, with AMD Ryzen 5 2600 CPU, 32GB RAM, and Ubuntu-18.04. To focus more on the impact of different translations for generics, we disable garbage collection and compiler optimisations for all translators. No benchmark requires type simulation. Thus, we disable this option in `dict`, allowing us to better understand the impact of method translation and dispatch.

6.3 Evaluation Results

6.3.1 *Micro benchmarks.*

Program @. We change n from 2 to 40 to analyse how the method number of a generic interface impacts the five translators. As shown in Figure 18a, the code size (number of assembly instructions) of translated FG programs has a linear relationship with n for all five translators. However, different translators have different coefficients. The coefficients of `mono` (328.8), `go2go` (300.8), and Go 1.18 (297.8) are much larger than the coefficients of `dict` (117.9) and `erasure` (103.8).

Figure 18b shows the execution time of translated programs. The programs translated by `dict` and Go 1.18 have a similar performance. They are slower than the corresponding programs translated by `mono` and `go2go`. This is largely due to the usage of dictionaries. The programs generated by `erasure` have the worst performance, since the structural typing conducted by `erasure` when it translates generic method calls to polymorphic method calls is very slow [Clement Rey 2018; Russ Cox 2009].

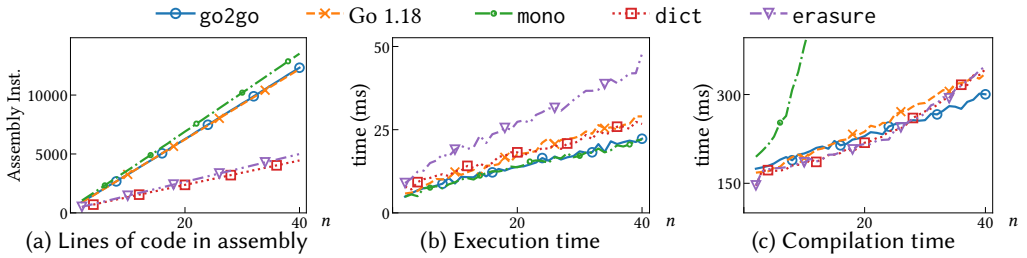

Fig. 18. Evaluation results of `Program` ⓐ

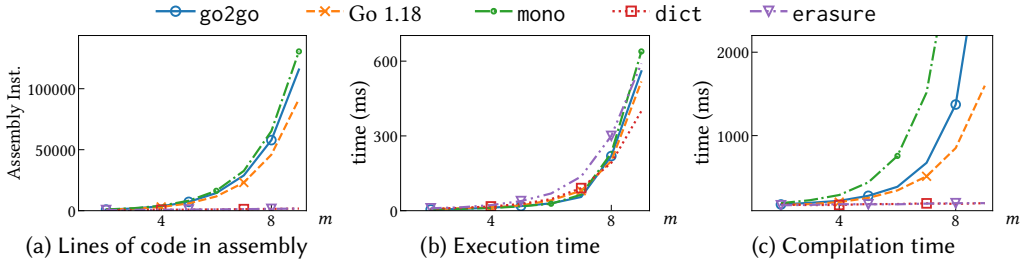

Fig. 19. Evaluation results of `Program` ⓔ

Figure 18c shows the compilation time. `mono` is significantly slower than the other four translators, and its compilation time is even not in a linear relationship with n. The compilation times of the other four translators are similar to each other.

Programs ⓑ and ⓓ. How the number of non-generic operations and the length of the call chain impact the three metrics is quite similar to the method number of generic interface `Base` in ⓐ. In particular, the code size, execution time, and compilation time are all in a linear relationship with the two configuration parameters, except for the compilation time of `mono`. Comparing ⓑ with ⓐ, one important difference to note is that for ⓑ, the programs translated by `dict` spend a similar execution time to that of the corresponding programs translated by `erasure`, and the execution time is larger than the execution time of the programs translated by Go 1.18. However, in Figure 18b for ⓐ, the line of `dict` is almost identical to the line of Go 1.18, indicating that their execution times are similar, and the line of `dict` is lower than the line of `erasure`. The reason is that when `dict` translates FGG to FG, it also synthesises type assertions for the non-generic operations in FGG (line 56 in Figure 17). The type assertions slow down the translated FG programs.

Program ⓒ. The code size, execution time, and compilation time all scale exponentially with m for the five translators. The underlying reason is that function `DoIt()` calls $\texttt{f}_1\texttt{()}$ 2^m times in each input FGG program. After normalising the three metrics with the number of characters in the FGG programs, we find that the three metrics are in a linear relationship with m. Among the five translators, `erasure`'s translated programs have the longest execution time. `dict` and `erasure` spend a similar compilation time, which is much shorter than `mono`, `go2go`, and Go 1.18. `dict`'s translated programs are similar in size to `erasure`'s translated programs, but they are smaller compared with the programs translated by `mono`, `go2go`, and Go 1.18.

Program ⓔ. As shown in Figures 19a and 19c, both the code size of the translated programs and the compilation time scale exponentially with m for `mono`, `go2go`, and Go 1.18. The reason is that $\texttt{f}_m\texttt{()}$ essentially calls `CallBase()` 2^m times with 2^m distinct parameter combinations, because

Table 1. Results of real-world benchmarks.

Name	Instruction Count					Execution Time (s)					Compilation Time (s)				
	go2go	Go 1.18	mono	dict	erasure	go2go	Go 1.18	mono	dict	erasure	go2go	Go 1.18	mono	dict	erasure
List	1201	1419	1736	862	840	23.2	22.4	24.4	37.69	30.89	0.24	0.22	0.23	0.22	0.22
ResizableArray	1976	2281	1882	867	841	7.0	6.8	6.9	14.90	15.00	0.24	0.22	0.23	0.23	0.22
ListReverse	1546	1818	1753	1115	1204	37.0	35.6	36.0	42.89	41.76	0.26	0.25	0.26	0.24	0.25
VectorReverse	985	1072	1047	921	914	2.99	2.9	3.12	2.66	2.68	0.25	0.24	0.26	0.24	0.25
Cell	112	104	75	196	151	.006	.009	.006	0.007	.007	0.24	0.23	0.16	0.24	0.24
Hashtable	188	184	209	245	249	0.24	0.21	0.46	0.46	0.45	0.19	0.17	0.19	0.19	0.19
Geometric mean	651.0	703.0	674.3	576.9	555.0	1.71	1.72	1.91	2.45	2.35	0.24	0.22	0.22	0.23	0.23

for $i \in [2, m)$, `f`$_i$`()` calls `f`$_{i+1}$`()` twice, with its input parameters plus `Red` for the first time and its parameters plus `Blue` for the second time, leading the three translators to copy `CallBase()` 2^m times. However, neither `dict` nor `erasure` makes any copy of `CallBase()`, and the code size of their translated programs is in a polynomial relationship with m (e.g., for `dict`'s translated programs, $size = 12.8m^2 + 34.5m + 381$, $p < 0.001$).

Contrary to the intuition, as shown in Figure 19b, the programs translated by `mono` have a worse execution performance compared with the corresponding programs translated by `dict`, when m is larger than 7. The reason is that when m is large, a program synthesised by `mono` has a large code size, and thus many cache misses occur during its execution. For example, when m is 9, the size of the executable file translated by `mono` is 6.3MB, and the executable triggers 6,058,156 cache misses in one run, while the program translated by `dict` only causes 93,695 cache misses.

Type simulation. As we discussed earlier, we disable the metadata copy of type simulation. If we enable the copy, then the translated programs become slower (e.g., 10% slower for @ when configuring n equal to 2). The slowdown becomes negligible when n is equal to 40.

6.3.2 Real-World Benchmarks. The evaluation results of real-world benchmarks are shown in Table 1. Overall, the translated programs of `dict` and `erasure` have a smaller code size, but a longer execution time, compared with the corresponding programs translated by `go2go`, `mono`, and Go 1.18, which is consistent with the results on the micro benchmarks. However, the compilation time does not change significantly across different translators, because all real-world benchmarks are small and do not have many usages of generics.

6.4 Discussion and Limitations

Our experimental results largely reflect the common intuition that monomorphisation translators (`mono`, `go2go`, and Go 1.18) generate programs with a better runtime performance, while non-specialising translators (`dict` and `erasure`) synthesise programs in a smaller code size. However, our evaluation also pinpoints cases where monomorphisation generates programs in an extremely large size. The programs trigger excessive cache misses during execution and have a very bad runtime performance. On the other hand, our experimental results motivate the introduction and usage of Go generics, since without generics, Go programmers have to implement polymorphism using interfaces, which is exactly the same as the programs translated by `erasure`, and our experimental results show that those programs are slow.

In practice, our dictionary-passing translator (`dict`) constantly generates programs in a smaller size and takes a smaller (or comparable) compilation time than all existing translators (including Go 1.18, the official generic type implementation). Thus, it provides an alternative for real-world users of Go generics to strike their desired tradeoff. Moreover, our implementation and evaluation

experience show that type simulation is an important component of `dict`, and that type metadata incurs extra runtime overhead. Thus, corresponding data structures and algorithms need to be carefully designed for better translated programs. For instance, link-time optimisation can be applied to remove unused type metadata.

Possible improvements for Go 1.18. First, Go 1.18 is very conservative in its support for GC shapes – only considering pointers to have the same GC shape. In our experiments, we do not observe the reuse of method implementations, or synthesis and use of dictionaries. Thus, to make full use of dictionaries and GC shape stenciling [Randall 2022], it is necessary for the Go team to improve the current implementation and support more GC shapes. Second, the Go team can consider dictionary-passing-based homogeneous compilation, as proposed in this paper, since it supports polymorphic recursion, provides a faster compilation speed, generates programs with a smaller code size, and enables separate compilation.

Limitations. Since the official generic type implementation released on March 15th, 2022, there does not yet exist generic Go code from large, production-run Go software (e.g., Docker, Kubernetes, etcd). We build the two benchmark suites to explore the translators' asymptotic behaviours and inspect how they perform on representative generic programs in other languages, which is our best effort in conducting the evaluation.

We formalise `dict` as a source-to-source translator to clarify design choices for future implementations and aid our proof of correctness (Theorem 5.1). However, this choice limits the performance of our implementation, and the evaluation results may not reflect the true capability of dictionary-passing translation for two reasons: first, we erase all types to `Any` to ensure type preservation, which is slow at runtime; and second, Go does not allow the creation of global constant dictionaries in source code, but those dictionaries can potentially be created by the Go compiler and leveraged by translated programs for a better runtime performance.

7 RELATED WORK

Implementation and benchmarks of generics.

To the best of our knowledge, there is no existing work comparing implementations of generics in Go. The closest ones target JVM languages [Odersky et al. 2000; Ureche et al. 2013], .NET common language runtime (CLR) [Kennedy and Syme 2001], and Haskell [Jones 1995]. Odersky et al. [2000] benchmark a homogeneous (similar to `erasure`) and a heterogeneous (similar to `mono`) translation for Pizza (an extension of Java with generics). They find that heterogeneity reduces execution time, but also increases code size. Jones [1995] gave a similar comparison for Haskell, reporting that monomorphisation produces a smaller code size; our work shows the opposite result. One major reason is that unnecessary dictionary fields and manipulation of dictionary parameters require more assembly instructions in Haskell than Go, as Go targets low-level efficiency.

Table 2. Implementations and benchmarks

	Language	Translation(s)	Optimal Exec. Time	Optimal Code Size
Our work	FGG (Go)	Dict/Mono/ Erasure†	Mono (1st) Dict (2nd)	Erasure† (1st) Dict (2nd)
Go team	Go	Mono/Hybrid	Mono	Mono
[Ureche et al. 2013]	Scala (JVM)	Hybrid	Hybrid	Hybrid
[Odersky et al. 2000]	Pizza (Java)	Mono/Erasure	Mono	Erasure
[Kennedy and Syme 2001]	.NET CLR	Hybrid	Hybrid	N/A
[Jones 1993]	Haskell	Dict/Mono	Mono	Mono

(†) FGG Erasure is not type preserving.

Kennedy and Syme [2001] apply a hybrid dictionary and monomorphisation approach targeting the Just-In-Time (JIT) .NET CLR compiler. Object instantiation is conducted lazily at runtime

according to an object's code and structure (*e.g.,* memory layout and garbage collection shape). Each object contains a pointer to a dictionary (vtable), which provides method entry points and type information. With the help of lazy instantiation during runtime, .NET CLR supports abundant language features, including but not limited to F-bounded polymorphism and polymorphic recursion. They compare their design with equivalent non-generic implementations using `Objects` and hand-specialised code. Their execution speed is close to that of the hand-specialised versions. The Go 1.18 approach is similar to .NET CLR, but unlike .NET CLR, its instantiation happens at compile time. Due to structural typing dictionaries are instantiated through an approach similar to instance discovery in monomorphisation. Hence, Go 1.18 suffers from an inability to support polymorphic recursion (*i.e.,* constrained by *nomono*, § 3.3) and the large code size of monomorphisation (§ 6).

Ureche et al. [2013] propose an optimised monomorphisation approach called miniboxing using one monomorphised instance on types with different sizes to reduce code size. Methods of different types are specialised at runtime using a custom classloader. They benchmark seven different settings, one achieving at most a 22 times speedup over the default generics translation in Scala. The main design goal of their benchmarks is the performance of reading and writing miniboxed objects allocated on heap by the JVM. They test the different combinations of concrete types for generics ("Multi Context"), which is similar to the scenario of `Program` ⓒ (in § 6.2), but their goal is to test the historical paths executed in the HotSpot JVM. They also test the speed of one method call `hashCode` from generics types. In comparison, our benchmarks test how various factors impact the performance (*e.g.,* the method number in an interface).

Formal translations of generics. Formal translations of generics can be split into three main techniques; *Erasure*, *dictionary-passing*, and *monomorphisation*. We consider the most relevant work, a breakdown of which is provided in Table 3. Where these works formally prove the correctness of their translation, we observe that they can be grouped as *behavioural equivalence* [Griesemer et al. 2020; Igarashi et al. 1999] and *value preservation* [Yu et al. 2004]. The former demands that during evaluation the source and target programs are still related, whereas the latter merely requires that the result of a productive program be preserved. In general behavioural equivalence is a more fine-grained equivalence, as it can be used to show value preservation. In this paper, we formalised and then proved our dictionary-passing translation correct using bisimulation up to dictionary-resolution, which is categorised as a behavioural equivalence.

Yu et al. [2004] formalise a hybrid dictionary and monomorphisation translation for the .NET CLR. They mostly follow the design of [Kennedy and Syme 2001]. They consider a target language which can, using an object's type, request the specific dictionary from an assumed infinite map. This is justified for the .NET CLR as method dictionaries are created on-demand using an object's type. Compare this to our translation in which we must eagerly construct dictionaries and pass them in addition to the objects that they describe. Yu et al. [2004, Theorem 5] show that their translation is value preserving; for expression e, and value v, if e evaluates to v ($e \Downarrow v$) then there is a reduction such that $(\!|e|\!) \longrightarrow^* (\!|v|\!)$ (where $(\!|-|\!)$ is their translation).

Table 3. Related Work: Theory

	Language	Approach	Translation(s)	Formalised
Our work	FGG (Go)	S-to-S	Dict	✔
[Griesemer et al. 2020]	FGG (Go)	S-to-S	Mono	✔
[Igarashi et al. 1999]	Java	S-to-S	Erasure	✔
[Yu et al. 2004]	.NET CLR	IR-to-IR	Hybrid	✔
[Bottu et al. 2019]	Haskell	S-to-IR	Dict	✔
[Odersky and Wadler 1997]	Pizza	S-to-S	Mono/Erasure	✗

S-to-S=Source to Source; IR=Intermediate representation

Bottu et al. [2019] formalise dictionary-passing in Haskell. Their work focuses on proving a *coherency theorem*. They motivate this work as nominally typed languages featuring multiple inheritance (*i.e.,* Haskell) suffer from an ambiguity in dictionary-resolution such that the translation of a single source program may *non-deterministically* produce different terms in the target language. A translation is coherent when these target terms are contextually equivalent. We need not consider this issue, as Go's structural typing system does not support the multiplicity of superclass implementations that causes incoherence. Bottu et al. [2019] do not prove the correctness of their dictionary-passing translation using an equivalence between the source and target language.

Griesemer et al. [2020] formalised the FG and FGG languages, as well as the `mono` translation used in § 6. This work defines a class of FGG programs that can be monomorphised, and proves that class membership is decidable. Finally, they prove that their translation forms a one-to-one bisimulation. Their behavioural equivalence is straightforward and does not require any up to techniques, as monomorphisation does not introduce runtime computations.

Odersky and Wadler [1997] describe, but do not formalise, two alternative approaches – erasure and monomorphisation – to implementing generics in the Pizza language, a generic variant of Java. Igarashi et al. [1999] build on the erasure technique developed in [Odersky and Wadler 1997]. Their work formalises Featherweight Generic Java and proves a formal erasure translation to Featherweight Java. They prove the correctness of their erasure translation using a behavioural equivalence, although their translation introduces *synthetic casts* (assertions), which complicates the correctness theorems. To resolve this issue, they introduce a reduction for their proofs which freely adds, removes, or safely alters any required synthetic casts. Correctness of their translation is split into two directions, called *weak completeness* and *soundness* [Igarashi et al. 1999, Theorem 4.5.4 and Theorem 4.5.5], which uses a behavioural equivalence up to the cast reduction. As with our paper, they use these theorems to show a value preservation corollary. Igarashi et al. [1999, Corollary 4.5.6] also prove that their erasure translation is type assertion error preserving – in contrast to our `erasure` translation, since ours does not preserve type assertions. This disparity is due to a limitation on the expressivity of assertion in Generic Java. The inclusion of this limitation has been an area of contention, with other authors suggesting that it could be overcome with the use of type-reps [Agesen et al. 1997; Allen and Cartwright 2002; Crary et al. 1998; Solorzano and Alagić 1998; Viroli and Natali 2000].

Formal non-generics dictionary translation. Sulzmann and Wehr [2021] propose a dictionary-passing translation from the non-generic FG to an *untyped* variant of the λ-calculus with pattern matching. They use a dictionary-passing approach to investigate Go's resolution mechanism for overloaded methods and structural subtyping. Sulzmann and Wehr [2021] prove that their translation is value preserving using a step-indexed logical relation.

Intuitively, Sulzmann and Wehr [2021] use an inductive proof technique that, using two related values v and v' at type t, relates any terms (e and $(\!|e|\!)$) that can reduce to v and v' (*resp.*) within k reduction-steps. Step-indexed logical relations are a sophisticated extension to logical relations (*e.g.,* [Bottu et al. 2019]), and are applicable for languages with recursion. Sulzmann and Wehr [2021] left a type-preserving translation from FG and a translation from FGG as their future work. No implementation or evaluation of their translation is provided.

Alternatives to bisimulation up to. In our motivating example for *up to dictionary resolution* (Figure 15), we briefly discuss potential alternate many-to-many bisimulation approaches. One such approach is the *stuttering bisimulation* [Browne et al. 1988], which has been studied extensively in the domain of model checking [Baier and Katoen 2008]. The stutter bisimulation relates two terms when they both reduce to related terms in an unbounded, but finite, number of steps. Formally, e and $(\!|e|\!)$ are related by a *stutter bisimulation* when (1) $e \longrightarrow e'$ implies that there exists a finite

reduction $(\!|e|\!) \longrightarrow d_0 \longrightarrow \cdots \longrightarrow d_n$ ($n \geq 0$) where each intermediate state d_i is related to e'; and symmetrically, (2) $(\!|e|\!) \longrightarrow d$ implies that there is a finite reduction from e with each element being related to d. This approach works well for finite models, but becomes *undecidable* when applied to Turing complete languages such as FGG. To overcome this issue, the works in [Hur et al. 2014; Leroy 2009] consider restricted, decidable, variants of the stutter bisimulation to show the correctness of their translations. Leroy [2009] formulates the non-symmetric *"star"-simulation*, which requires a well-founded ordering on reducing terms to ensure that either (1) both source and target terms reduce infinitely; or (2) the source cannot reduce infinitely while the target is stuck. In practice, the well-founded ordering used in (2) is approximated using fixed parametric bounds. Hur et al. [2014] formulate this idea using *stuttering parametric bisimulation*, which bounds the number of steps that two related terms can take before their reductions are related. Such restricted variants of the stutter bisimulation cannot provide a sound and complete correctness proof for `dict`. More generally, our use of a fine-grained up to bisimulation not only develops on existing correctness theorems for the translation generics [Griesemer et al. 2020; Igarashi et al. 1999], but it can also be readily extended to include advanced language features such as concurrency and side effects in Go.

8 CONCLUSION

In this paper, we design and formalise a new source-to-source, non-specalised call-site dictionary-passing translation of Go, and prove essential correctness properties introducing a novel and general *bisimulation up to* technique. The theory guides a correct implementation of the translation, which we empirically compare along with the recently released Go 1.18, an erasure translator, and two existing monomorphisation translators [Griesemer et al. 2020; The Go Team 2021b], with micro and real-world benchmarks. We demonstrate that our dictionary-passing translator handles an important class of Go programs (*F*-bounded polymorphism and *nomono* programs) beyond the capability of Go 1.18 and existing translations [Griesemer et al. 2020; The Go Team 2021b], and provide several crucial findings and implications for future compiler developers to refer to. For instance, Go 1.18 requires more improvements on GC shapes in order to effectively generate small binary code (See 6.4 for a more detailed discussion).

Beyond Go language, many dynamically typed languages (such as Python, JavaScript, and Erlang) type-check at runtime, and their engines cannot easily decide an object's implemented methods nominally, similarly to Go. Consequently, many of their implementations [Castanos et al. 2012; Gal et al. 2009; Salib 2004] apply similar approaches to monomorphisation to optimise execution speed. Rust also supports generic via monomorphisation, yet this is considered a major reason for slow compilation. Our work can help in choosing alternative optimisations for these languages to reduce code size and compilation time.

In the future, we plan to inspect how other important Go language features (e.g., *reflection*, *packages*, *first-class*, *anonymous* functions) interact with generics by proving the correctness and examining the trade-offs among runtime performance, code sizes, and compilation times.

ACKNOWLEDGMENTS

The authors wish to thank Ziheng Liu and Zi Yang for benchmark collection and initial paper discussion, and the anonymous reviewers for their invaluable comments and suggestions on the paper. This work was partially supported by EPSRC (EP/T006544/1, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T006544/1, EP/T014709/1, EP/V000462/1 and EP/X015955/1), NCSS/EPSRC VeTSS, a Mozilla Research Award, and an Ethereum Grant.

REFERENCES

Ole Agesen, Stephen N. Freund, and John C. Mitchell. 1997. Adding Type Parameterization to the Java Language. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Atlanta, Georgia, USA) *(OOPSLA '97)*. Association for Computing Machinery, New York, NY, USA, 49–65. https://doi.org/10.1145/263698.263720

Eric Allen and Robert Cartwright. 2002. The Case for Run-Time Types in Generic Java. In *Proceedings of the Inaugural Conference on the Principles and Practice of Programming, 2002 and Proceedings of the Second Workshop on Intermediate Representation Engineering for Virtual Machines, 2002* (Dublin, Ireland) *(PPPJ '02/IRE '02)*. National University of Ireland, Maynooth, County Kildare, IRL, 19–24.

Christal Baier and Joost P. Katoen. 2008. *Principles of Model Checking*. MIT Press, United States.

Gert-Jan Bottu, Ningning Xie, Koar Marntirosian, and Tom Schrijvers. 2019. Coherence of Type Class Resolution. *Proc. ACM Program. Lang.* 3, ICFP, Article 91 (jul 2019), 28 pages. https://doi.org/10.1145/3341695

M.C. Browne, E.M. Clarke, and O. Grümberg. 1988. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science* 59, 1 (1988), 115–131. https://doi.org/10.1016/0304-3975(88)90098-9

Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. 1989. F-Bounded Polymorphism for Object-Oriented Programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*. Association for Computing Machinery, New York, NY, USA, 273–280. https://doi.org/10.1145/99370.99392

Jose Castanos, David Edelsohn, Kazuaki Ishizaki, Priya Nagpurkar, Toshio Nakatani, Takeshi Ogasawara, and Peng Wu. 2012. On the Benefits and Pitfalls of Extending a Statically Typed Language JIT Compiler for Dynamic Scripting Languages. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 195–212. https://doi.org/10.1145/2384616.2384631

Clement Rey. 2018. go-internals. https://github.com/teh-cmc/go-internals/blob/master/chapter2_interfaces/README.md#anatomy-of-an-interface. (Accessed on 04/15/2022).

Karl Crary, Stephanie Weirich, and Greg Morrisett. 1998. Intensional Polymorphism in Type-Erasure Semantics. *SIGPLAN Not.* 34, 1 (sep 1998), 301–312. https://doi.org/10.1145/291251.289459

Docker. 2021. Docker - Build, Ship, and Run Any App, Anywhere. https://www.docker.com/. (Accessed on 04/15/2022).

Karel Driesen and Urs Hölzle. 1996. The Direct Cost of Virtual Function Calls in C++. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Jose, California, USA) *(OOPSLA '96)*. Association for Computing Machinery, New York, NY, USA, 306–323. https://doi.org/10.1145/236337.236369

Stephen Ellis, Shuofei Zhu, Nobuko Yoshida, and Linhai Song. 2022a. Artifact of "Generic Go to Go: Dictionary-Passing, Monomorphisation, and Hybrid". https://doi.org/10.5281/zenodo.7067362

Stephen Ellis, Shuofei Zhu, Nobuko Yoshida, and Linhai Song. 2022b. fgg2go. https://github.com/sfzhu93/fgg2go.

Stephen Ellis, Shuofei Zhu, Nobuko Yoshida, and Linhai Song. 2022c. fgg2go. https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/sfzhu93/fgg2go.

Stephen Ellis, Shuofei Zhu, Nobuko Yoshida, and Linhai Song. 2022d. *Generic Go to Go: Dictionary-Passing, Monomorphisation, and Hybrid*. Technical Report. https://doi.org/10.48550/ARXIV.2208.06810

Free Software Foundation. 2021. objdump(1) - Linux manual page. https://man7.org/linux/man-pages/man1/objdump.1.html. (Accessed on 04/15/2022).

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-Based Just-in-Time Type Specialization for Dynamic Languages. *SIGPLAN Not.* 44, 6 (jun 2009), 465–478. https://doi.org/10.1145/1543135.1542528

gitchander. 2021. Permutation. https://github.com/gitchander/permutation. (Accessed on 04/15/2022).

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

gRPC. 2021. A high performance, open-source universal RPC framework. https://github.com/grpc/grpc-go.

Raymond Hu. 2021. *Mini prototype of FG/FGG/FGR in Go*. https://github.com/rhu1/fgg

Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2014. *A logical step forward in parametric bisimulations*. Technical Report. Technical Report MPI-SWS-2014-003, MPI-SWS.

Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. 1999. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA) *(OOPSLA '99)*. Association for Computing Machinery, New York, NY, USA, 132–146. https://doi.org/10.1145/320384.320395

Cliff B. Jones. 1993. Constraining Inference in an Object-Based Design Model. In *Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT '93)*. Springer-Verlag, Berlin, Heidelberg, 136–150. https://doi.org/10.1007/3-540-56610-4_61

Mark P. Jones. 1995. Dictionary-Free Overloading by Partial Evaluation. *Lisp Symb. Comput.* 8, 3 (sep 1995), 229–248. https://doi.org/10.1007/BF01019005

Andrew Kennedy and Don Syme. 2001. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/378795.378797

Kubernetes. 2021. Production-Grade Container Orchestration. https://kubernetes.io/. (Accessed on 04/15/2022).

Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (Nov. 2009), 363. https://doi.org/10.1007/s10817-009-9155-4

Alice Merrick. 2020. Go Developer Survey 2020 Results. https://blog.golang.org/survey2020-results (Accessed on 04/15/2022).

Robin Milner and Davide Sangiorgi. 1992. Barbed Bisimulation. In *19th ICALP (LNCS)*, W. Kuich (Ed.), Vol. 623. Springer, 685–695. https://doi.org/10.1007/3-540-55719-9_114

Martin Odersky, Enno Runne, and Philip Wadler. 2000. Two ways to bake your pizza—translating parameterised types into Java. In *Generic Programming*. Springer, 114–132. https://doi.org/10.1007/3-540-39953-4_10

Martin Odersky and Philip Wadler. 1997. Pizza into Java: Translating Theory into Practice. In *POPL '97* (Paris, France). ACM Press, 146–159. https://doi.org/10.1145/263699.263715

Keith Randall. 2020a. Generics implementation - Dictionaries. https://go.googlesource.com/proposal/+/refs/heads/master/design/generics-implementation-dictionaries.md. (Accessed on 04/15/2022).

Keith Randall. 2020b. Generics implementation - GC Shape Stenciling. https://go.googlesource.com/proposal/+/refs/heads/master/design/generics-implementation-gcshape.md (Accessed on 04/15/2022).

Keith Randall. 2020c. Generics implementation - Stenciling. https://go.googlesource.com/proposal/+/refs/heads/master/design/generics-implementation-stenciling.md (Accessed on 04/15/2022).

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 (Accessed on 04/15/2022).

Russ Cox. 2009. Go Data Structures: Interfaces. https://research.swtch.com/interfaces.

Michael Salib. 2004. *Starkiller: A static type inferencer and compiler for Python.* Ph.D. Dissertation. Massachusetts Institute of Technology.

Jose H. Solorzano and Suad Alagić. 1998. Parametric Polymorphism for Java: A Reflective Solution. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Canada) *(OOPSLA '98)*. Association for Computing Machinery, New York, NY, USA, 216–225. https://doi.org/10.1145/286936.286959

IEEE Spectrum. 2022. Top Programming Languages - IEEE Spectrum. https://spectrum.ieee.org/top-programming-languages/. (Accessed on 04/15/2022).

Stack Overflow. 2021. 2021 Developer Survey. https://insights.stackoverflow.com/survey/2021 (Accessed on 04/15/2022).

Bjarne Stroustrup. 1997. *The C++ Programming Language* (3rd ed.). Addison Wesley.

Martin Sulzmann and Stefan Wehr. 2021. A Dictionary-Passing Translation of Featherweight Go. In *Programming Languages and Systems: 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17–18, 2021, Proceedings* (Chicago, IL, USA). Springer-Verlag, Berlin, Heidelberg, 102–120. https://doi.org/10.1007/978-3-030-89051-3_7

The Go Team. 2021a. The Go Programming Language Specification - The Go Programming Language. https://https://golang.org/ref/spec#Function_types (Accessed on 04/15/2022).

The Go Team. 2021b. The go2go playground. https://go2goplay.golang.org/. (Accessed on 04/15/2022).

The Go Team. 2022. Go 1.18 Release Notes. https://tip.golang.org/doc/go1.18 (Accessed on 04/15/2022).

Vlad Ureche, Cristian Talau, and Martin Odersky. 2013. Miniboxing: improving the speed to code size tradeoff in parametric polymorphism translations. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 73–92. https://doi.org/10.1145/2509136.2509537

Mirko Viroli and Antonio Natali. 2000. Parametric Polymorphism in Java: An Approach to Translation Based on Reflective Features. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Minneapolis, Minnesota, USA) *(OOPSLA '00)*. Association for Computing Machinery, New York, NY, USA, 146–165. https://doi.org/10.1145/353171.353182

Dachuan Yu. 2004. *Safety Verification of Low-Level Code.* Ph.D. Dissertation.

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* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 39–51. https://doi.org/10.1145/964001.964005