School of Electronic Engineering
and Computer Science

Final Report

Programme of study:
BSc Computer Science

Project Title:
Generic array sizes in the Go programming language

Supervisor:
Dr. Raymond Hu

Student Name:
Dawid Lachowicz

Final Year
Undergraduate Project 2023/24

Date: April 29, 2024

Abstract

Go is a language developed by Google and the open-source community and is widely used in industry. Many have praised Go for its simplicity, while others criticised its lack of certain features, most notably generic programming. While the introduction of generics in Go 1.18 addressed the largest criticism, there are still areas where a more expressive version of the language would benefit users. One such area, which this work focuses on, is generically sized (static) arrays. To achieve this, a new kind of type parameter is introduced — one which can be instantiated with (compile-time) constant integers *values*. This work formalises a subset of Go and then extends the formalisation with type parameters that include the new numerical type parameters. A translation from the extended language to regular Go is also formalised (monomorphisation). To test the formalisations, two interpreters (with static type-checking) and a monomorphiser have been implemented, along with rigorous testing.

Contents

1 Introduction

Generics have been introduced in the Go programming language following the theoretical work by Griesemer et al. (2020) in *Featherweight Go*. The *Type Parameters Proposal* lists several generic programming constructs not supported by the initial implementation of generics (as of Go 1.18). Among them is "no parameterization on non-type values such as constants." (Taylor and Griesemer, 2021) The most notable use case of such type parameters would be for arrays. In Go, the size of an array is part of its type (*The Go Programming Language Specification*, 2023). As such, if the programmer wishes to write a function that operates on arrays or a data type that contains arrays, it is necessary to hard-code the size of the operated/contained array. This imposes a limitation on what abstraction may be introduced where arrays are concerned. Extending the generic type system in Go to support constants as type parameters aims to resolve this issue.

1.1 Background

Arrays are a primitive data structure found in many programming languages. However, various languages treat arrays differently. In Java, arrays are objects, and variables of array type are references to those objects. The size (or length) of an array is not part of its type. However, it is a property of the array object instance and cannot be changed after initialisation of the instance (Gosling et al., 2023). C# treats arrays analogously (B. Wagner, 2023). In languages like Go and Rust, arrays are value types, and the size of the array *is* part is of its type (*The Go Programming Language Specification*, 2023; *The Rust Reference*, 2020). In C, the size of an array is also part of its type. However, expressions of array type are converted to pointers to the first element of the array (ISO/IEC, 2018). Since dynamically typed languages do not have types associated with variables, those languages will not be discussed here.

1.2 Problem Statement

Arrays in the Go programming language have a number of use cases and situational benefits over slices — their dynamic counterparts. One practical scenario is when wanting to use a collection as a map key. Since the comparison operators are fully defined for arrays, they may be used as map keys, which is a very common data structure used in Go programs. The comparison operators are not fully defined for slices. Hence, they cannot be used as map keys in the same way. (*The Go Programming Language Specification*, 2023). Arrays may also be useful when value semantics are desired, i.e. assigning an array to another variable or passing it as an argument to a function makes a copy. For slices, which hold references to underlying arrays, copies need to be performed manually, which can be more verbose and error-prone. Since the size of an array is part of its type, it can also serve as documentation to the reader of the code.

In current day Go (version 1.21 as of the time of writing), there is no way of abstracting over arrays of any size, e.g. it is impossible to define a function that operates on an array of any size. While workarounds exist that make use of type set interfaces introduced in the *Type Parameters Proposal* (Taylor and Griesemer, 2021), this solution is not very elegant

since it requires manually enumerating all the array sizes the function can operate on, and also cannot be used in functions exposed as part of a library, as there is no way of knowing what array size the library consumer will use.

This work aims to address this problem by introducing constant (integer) type parameters that may be used to define generically-sized array types. A proposal exists to allow for generic parameterisation of array sizes in Go, demonstrating the demand for this feature (Werner, 2021). However, the design proposed in this work differs from the existing proposal.

1.3 Aim

The aim is to produce a set of formal rules specifying the syntax, reduction semantics and type system of a subset of the Go programming language, extended with support for generically sized arrays. The rules are to be verified by implementing an interpreter that includes the new language feature, as well as a monomorphiser that translates the extended language into regular Go.

As an extension to the project, a formal proof of correctness of the language extension may be carried out. Another extension would be to implement the proposed design into the mainstream Go compiler.

1.4 Objectives

- Investigate how the problem of abstracting over arrays of any size has been solved in other statically-typed programming languages, and what research has been done in this area.
- Formalise the syntax, reduction and typing rules of arrays in Go, based on *Featherweight Go*, referred to as FGA going forward (Griesemer et al., 2020).
- Design and formalise the syntax, reduction and typing rules of generically sized arrays, based on *Featherweight Generic Go*, updated with the existing state of generics in Go, referred to as FGGA going forward (Griesemer et al., 2020).
- Implement and test a type checker and interpreter for FGA extended with support for arrays.
- Implement and test a type checker and interpreter for FGGA extended with support for generically sized arrays.
- Implement and test a monomorphiser that translates code written in the extended FGGA into regular Go code.
- Submit a proposal for adding the language extension to the open source community, addressing the challenges previously discussed by the community, and address any feedback for my design.

1.5 Research Questions

- How have other statically typed languages where the size of an array is part of its type tackled this issue, if at all?
- How can the Go type system be extended to support generically sized arrays, compatible with the existing implementation of generics in Go?
- How can the proposed design be verified to be correct?

1.6 Report Structure

The second chapter explores how other languages have tackled the problem of abstracting over arrays of any size, as well as past research in the area of programming languages, in particular relating to the Go programming language. The third chapter presents a rationale for introducing generically sized arrays in Go through examples. The fourth chapter goes into detail about the proposed language extension, and explores certain design considerations. Chapter five introduces the formal rules for "Featherweight Go with Arrays" — a subset of Go containing arrays, based on the paper going by the name "Featherweight Go" (Griesemer et al., 2020). Chapter six extends the rules with generics as found in today's version of Go and the proposed design for generically sized arrays. Chapter seven formally presents a way of translating FGGA to regular Go via monomorphisation. Finally, the eighth chapter gives an overview of the implementation of two interpreters and a monomorphiser, corresponding to the rules found in chapters five, six and seven.

2 Background Research

2.1 Generic array sizes in other languages

Languages where the size of an array is not part of its type (e.g. Java, C#), automatically abstract over arrays of all sizes, as there is no way of expressing that a variable holds an array of a specific size.

In C, macros can be used to achieve a form of generic programming, which includes the ability to parameterise the size of an array forming part of its type (see code example in appendix D.1).

2.1.1 Const generics in Rust

In Rust, arrays are treated very similarly to how they are treated in Go: arrays are value types, and the size is part of the array's type. As such, the development of arrays in Rust is of particular interest when designing the generic extension in Go. In particular, *const generics* have already been introduced to Rust, facilitating parameterisation over array sizes (The const generics project group, 2021).

The syntax introduces the `const` keyword in the type parameter list, followed by the type parameter name and its bound (same as for non-constant type parameters). The current implementation only permits integral types for the bounds of const type parameters, which is reasonable given that the main rationale for const generics is to be able to parameterise over array sizes. The authors could have limited const generics to the `usize` type (the type used for array sizes in Rust). However, they claim it would not have made the implementation simpler (Aronson, 2017). Since Go puts a strong emphasis on language simplicity (Pike, 2015), this work proposes to limit constant generics in Go to only accept integral type parameters of the same type that can be used as array sizes, i.e. "a non-negative constant representable by a value of type `int`" (*The Go Programming Language Specification*, 2023). The implication of this design decision is that the syntax can be simplified, as `const` can be made to imply the array-length type described by the Go specification.

Rust imposes limitations on constant type arguments for ease of implementation. In particular, expressions including a const parameter (with the exception of a lone const parameter) are not permitted as type arguments. In the monomorphisation model of generics, recursive definitions with const type argument expressions containing const type parameters could lead to extreme code bloat in the compiled binary. Another issue would be the potential for the const type argument to go out of bounds (e.g. less than 0 as a result of a subtraction operation on the type parameter). The simple solution is to not permit such type arguments. This insight can be carried over to the Go design. The simplicity of the feature with this constraint also aligns with the ethos of Go (Pike, 2015).

While Rust implements generics using monomorphisation (*Rust Compiler Development Guide*, 2023), Go uses a hybrid of monomorphisation and dictionary-passing (Scales and Randall, 2022). Further investigation needs to be undertaken to determine whether this difference affects how constant type parameters could be implemented in Go, such as potentially lifting the constraint on recursive constant type parameter expressions.

```
struct ArrayPair<T, const N: usize> {
    left: [T; N],
    right: [T; N],
}
```

Figure 1: Example of const generics for arrays in Rust (The const generics project group, 2021)

2.2 Type systems and programming language theory

B.C. Pierce (2002) covers core topics in programming languages and type theory including how to precisely describe the syntax, evaluation and static type system of programming languages, and techniques for proving properties of those languages. The book formally presents the notion of language type safety consisting of *progress* and *preservation*.

The progress theorem says that well-typed terms (i.e. those that are in the typing relation defined by the typing rules) do not get "stuck", i.e. they are either terminal values or can take a reduction step, as defined by the reduction (evaluation) rules.

The preservation theorem says that if a term is well-typed, then it will continue to be well-typed after a reduction step. Depending on the language, the type system may impose further restrictions on this theorem, e.g. that the type of the term after taking a step is exactly the same type as before the step, or that it is a subtype of the type before the step.

To summarise, if we have a term that is well-typed, then by the preservation theorem, it will continue to be well-typed no matter how many reduction steps are taken. Together with the progress theorem, we know that the term will always be able to take a reduction step (because it is well-typed after each one) until it reaches a terminal value. In other words, the well-typed term will never get "stuck" during execution. These theorems can be used to prove the soundness of a type system, which is fundamental for any well-designed type system, including the one proposed in this work.

The book also presents how to conduct such proofs, namely using a technique called *structural induction*. Structural induction is analogous to induction on natural numbers, except that it works on recursively defined structures, such as those found in the formal rules of programming languages.

2.3 Dependent and refinement types

Both dependent and refinement types have the goal of being able to assign more precise types in programs, meaning more invariants can be checked at compile time, leading to more bugs being caught early in software development (Xi and Pfenning, 1999; Jhala and Vazou, 2021).

A dependent type is a type depending on a *value*, i.e. an element of another type. The most basic example, and the one we are most concerned with in this work, is the type "array of size n" where n is an element of the integer type. When n is a parameter as opposed to a concrete value, this is known as a family of types. This form of dependent typing can be traced as far back as FORTRAN. Dependent types can be more complex than simple array

sizes, e.g. we can express balanced trees (by depending on the values of their heights) or sorted lists (Bove and Dybjer, 2009).

Dependent typing can be found in research functional languages such as Idris and Agda (Brady, 2011; Norell, 2007). Dependent types need not depend on constants (i.e. concrete values known at compile time, such as the literal `1`) — they may depend on arbitrary terms (e.g. variables). Mainstream languages including C++ and Rust, as well as the Go extension proposed in this work, do restrict the dependency to constants only.

Dependent types are the more general form, whereas refinement types impose a restriction that the type must depend on a *predicate* (which itself involves values). These predicates are said to refine (narrow down) the set of values described by the base type (Jhala and Vazou, 2021). Moreover, unlike dependent types, these predicates cannot be arbitrary expressions but are rather formed from a more restricted language (than the language they are used in). The exact restrictions depend on the specifics of the refinement type system. One concrete example is liquid types which enforce decidability of the predicates. Liquid types have been implemented for several major languages, including recently for Java, which shows active development in this area of research. (Vazou, 2015; Gamboa et al., 2023).

Since this work intends to lay the foundation for generically sized arrays, we are not directly concerned with refinement types. However, in section 4.3 we explore how in the future we can make the usage of generically sized arrays more expressive through refinements of the array length, i.e. from arbitrary to a specific range.

2.4 Featherweight Go

For many years, the biggest criticism against the Go language was the lack of generics (Merrick, 2022; 2021; Kulesza, 2020). The Go team recognised the importance of solving this problem "right" and consequently reached out to the world of academics for a collaboration, the result of which was a paper named *Featherweight Go* (Griesemer et al., 2020). The work was inspired by *Featherweight Java*, an effort two decades prior aimed at formalising Java and its generic extension (Igarashi, B. Pierce, and Wadler, 1999). The common theme in the two papers is the reduction of the programming language into a small core "featherweight" subset, making it easier to prove properties about the language and, subsequently, any proposed extensions. Both papers also extended their language subsets into variants with generics (parametric polymorphism), and showed how the generic variant can be translated into the non-generic base language.

Featherweight Go formalises a subset of Go through syntax, reduction and typing rules, as well as an extended variant with generics. They are used as the starting point for the formal rules in this work. In their work, Griesemer et al. prove the soundness of the FG and FGG type systems using the progress and progress theorems described in the previous section.

Griesemer et al. argue that both "featherweight" and "complete" descriptions of languages have value. This work will follow the featherweight approach, since it was a successful strategy in the case of *Featherweight Go*, aimed at introducing a new feature to the language, and keeps the focus on the parts of the language that matter most when making the addition.

Because the *Type Parameters Proposal* (implemented as of Go 1.18) only includes a

subset of the generic extension features described in *Featherweight Go*, this work will use that reduced description, both for simplicity and to better mirror the state of generics in current-day Go (Taylor and Griesemer, 2021).

Featherweight Go also fully implemented the described languages as interpreters to test that all presented examples work as expected. This work intends to follow in the footsteps of Griesemer et al. in this regard.

3 Arrays by example

Most of the time, Go programmers will use slices over arrays due to their dynamic nature. However, when the size of a collection of elements can be determined at compile time, there are certain benefits to using arrays. This section intends to identify use cases where using arrays in Go programs may be more beneficial than using slices, and what are the benefits and the performance implications.

3.1 Gentle introduction

The first function we'll examine creates a reversed copy of an array/slice. Below is a side-by-side view of the two functions: the first operating on arrays, and the second operating on slices.

```
func reversedArray(arr [N]int) [N]int {
    n := len(arr)
    for i := 0; i < n/2; i++ {
        arr[i], arr[n-i-1] = arr[n-i-1], arr[i]
    }
    return arr
}

func reversedSlice(s []int) []int {
    n := len(s)
    newS := make([]int, n)
    for i := 0; i < n/2; i++ {
        newS[i], newS[n-i-1] = s[n-i-1], s[i]
    }
    return newS
}
```

The slice function is almost identical to the array function, except that we need to explicitly allocate a new slice for the result. Arrays are value types, so simply passing an array into a function (and similarly returning an array from a function) creates a copy. For relatively small arrays (8MB in size or less), the memory for the array copy is allocated on the stack. For slices, however, even for the smallest slices (of length 1 or more), the `reversed` function allocates the new slice on the heap. This operation is more expensive, and so, as the benchmarks show, the array variant of the `reversed` functions performs on average around 50% faster for arrays of size 8MB or less. For very small arrays (64 `int`s or less), the performance benefits or arrays for this operation are even more apparent. Once the array reaches 16MB or more, the memory for the copy gets allocated onto the heap and becomes slower than using slices. The benchmarks are limited to 256MB, since at array sizes of 512MB the compiler rejects the program since it could potentially use up more than 1GB of stack space (we need to multiply 512MB by two since we are creating a copy of the array). So if collections of such large sizes are necessary, slices are the only option. For smaller sizes, however, arrays perform better.

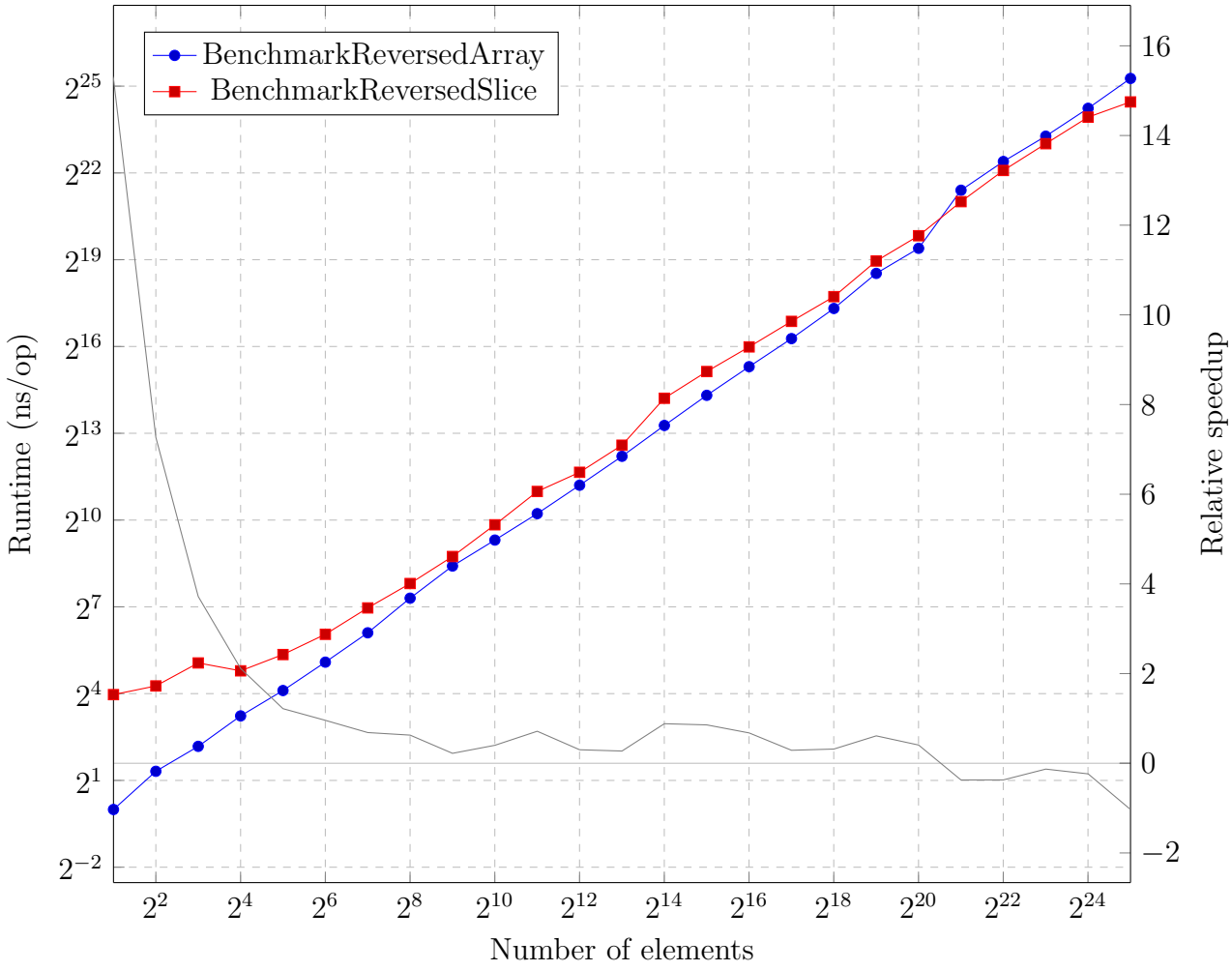

Figure 2: Comparison of `reversed` function benchmarks captured on Apple M1 Pro

The benchmarks were run on a single core, using `GOMAXPROCS=1`, and the results of the functions were written to global variables, in an attempt to prevent the compiler optimiser from eliminating the benchmarked code (Cheney, 2013).

The literal `N` in the array type refers to a constant, defined elsewhere in the program. The function signature only accepts arrays of length `N`, despite the function body being generic enough that it could work on arrays of any length. To illustrate the point, the next example will inline `N` with an integer literal. The function also happens to only accept `int` arrays, but this can easily be fixed using generic type parameters, introduced in Go 1.18 (unfortunately, at the cost of some performance):

```
func reversedArray[T any](arr [5]T) [5]T {
    n := len(arr)
    for i := 0; i < n/2; i++ {
        arr[i], arr[n-i-1] = arr[n-i-1], arr[i]
    }
    return arr
}
```

The implication of this is that for each different array length we want to use the `reversed` operation on, we would have to write a new function with the exact same code, or use a code generation tool to do this for us.

As of Go 1.18, there is a workaround that partially solves the above problem. Interface types are now defined in terms of the more general notion of type sets, as opposed to method sets pre Go 1.18 (*The Go Programming Language Specification*, 2023; *The Go Programming Language Specification*, 2021). "General interfaces" were introduced, that can only be (as of Go 1.21) used as type parameter constraints, and among its features is the ability to specify a union of types. With this, we can define an interface in terms of the union of differently sized arrays that we wish to use with our `reversed` function (Werner, 2021):

```
type array[T any] interface {
    [2]T | [3]T | [4]T | [5]T
}

func reversedArray[T any, A array[T]](arr A) A {
    n := len(arr)
    for i := 0; i < n/2; i++ {
        arr[i], arr[n-i-1] = arr[n-i-1], arr[i]
    }
    return arr
}
```

This approach still has limitations. Apart from the obvious burden of having to update the `array` interface every time we use an array of a new size, this model breaks down as soon as we wish to expose such a function as part of a public API. There is no way of knowing ahead of time what array sizes a user may wish to use, and enumerating them all is infeasible. Ideally, we'd want a way to abstract over arrays of any size.

3.2 Array semantics

We've looked at an example where using arrays is faster than slices. In general, it is difficult to reliably build array-based data structures that offer better performance than slices, since all the operations need to be handwritten and optimised. One rule of thumb is that heap allocations are expensive, and it is easier to avoid heap allocations (e.g. when making copies) for arrays than for slices. Figures 3 and 4 compare the semantics of arrays versus slices, and an accompanying description can be found in appendix C.

```
import "fmt"

type Outer struct {
    inner   Inner
    someVal int
}

type Inner struct {
    matrix [2][2]int
}

func main() {
    x := Outer{Inner{[2][2]int{{1, 2}, {3, 4}}}, 5}
    y := x                              // trivial deep-copy
    fmt.Println(x == y)                 // deep-comparison: prints "true"

    myMap := map[Outer]int{x: 42}       // structure can be used as map key
    fmt.Println(myMap[x])               // prints: 42
    fmt.Println(myMap[y])               // prints: 42 (y has same value as x)

    y.inner.matrix[0][1] = 10           // update y
    fmt.Println(x.inner.matrix[0][1]) // x remains unchanged: prints "2"
    fmt.Println(x == y)                 // deep-comparison: prints "false"
    fmt.Println(myMap[x])               // prints: 42 (x remains unchanged)
    fmt.Println(myMap[y])               // prints: 0 (new y value not in map)
}
```

Figure 3: Array semantics allow for trivial deep-copying and comparison

```
import "fmt"

type Outer struct {
    inner   Inner
    someVal int
}

type Inner struct {
    matrix [][]int
}

func main() {
    x := Outer{Inner{[][]int{{1, 2}, {3, 4}}}, 5}
    y := x                              // non-deep copy: matrix is shared
    y.inner.matrix[0][1] = 10           // update y
    fmt.Println(x.inner.matrix[0][1]) // x also changed: prints "10"

    fmt.Println(x == y)                 // invalid operation: uncomparable type
    _ = map[Outer]int{x: 42}            // invalid map key type Outer
}
```

Figure 4: Slices hold references to underlying data, and cannot be compared

3.2.1 Limitations

The drawback of defining array-based data structures and operations on them is that the array sizes must be fixed at compile time. By making array sizes generic, we can parameterize array-based data structures over many sizes known at compile time, making them much more versatile. The following subsection outlines some examples of array-based data structures, and then we proceed to describe how numerical type parameters can resolve this limitation and propose a design for them.

3.3 Data structure examples

Array-based data structures can enjoy the benefits of being easily copyable and comparable, even as part of nested data structures. The figures present two popular data types: a "resizable" array with a fixed maximum capacity, and a double-ended queue with an underlying fixed-size ring buffer (circular array)[1].

The two example data structures can be fully implemented in FGA, with minor adjustments. Most notably, FGA does not support mutation via pointers (or pointers at all), so functions that "mutate" the data structure return updated copies in FGA instead. Additionally, user-raised panics (via calls to the Go `panic`) function are not supported, so some fallback (such as returning the zero-value, or performing a no-op) is chosen instead.

[1]The "wasted" slot strategy was used to differentiate between "empty" and "full" states (Johnston, 2017)

```
type Array struct {
    arr [N]int
    len int
}

func (a *Array) Push(el int) {
    if a.len >= N {
        panic("array is full")
    }
    a.arr[a.len] = el
    a.len++
}

func (a *Array) Pop() int {
    if a.len == 0 {
        panic("array is empty")
    }
    a.len--
    return a.arr[a.len]
}

func (a *Array) Get(i int) int {
    if i >= a.len {
        panic("index out of bounds")
    }
    return a.arr[i]
}

func (a *Array) Len() int {
    return a.len
}
```

```
type Array struct {
    arr Arr
    len Nat
}

func (a Array) Push(el int) Array {
    return a.Cap().ifLessEqA(a.len,
        ArrayFunc{a},
        PushFunc{a, el})
}
func (f PushFunc) call() Array {
    return Array{
        f.a.arr.set(
            f.a.len.val(), f.el),
        Succ{f.a.len}}
}

func (a Array) Pop() Array {
    return Array{a.arr, a.len.pred()}
}

func (a Array) Get(i Nat) int {
    return a.len.ifLessEq(i,
        IntFunc{0},
        ArrGetFunc{a.arr, i.val()})
}
func (f ArrGetFunc) call() int {
    return f.arr[f.i]
}

func (a Array) Len() Nat {
    return a.len
}
```

Figure 5: "Resizeable" array with a fixed underlying buffer. The left-hand side shows idiomatic Go implementation, while the right-hand side shows FGA compatible implementation. Full code definitions (including `Nat`) can be found in the appendix.

```
type Deque struct {
    arr   [N + 1]int // "waste" a slot to detect fullness
    front int
    back  int
}

func (d *Deque) PushFront(el int) {
    if d.wrapped(d.front+1) == d.back { panic("deque is full") }
    d.arr[d.front] = el
    d.front = d.wrapped(d.front + 1)
}

func (d *Deque) PopFront() int {
    if d.front == d.back { panic("deque is empty") }
    d.front = d.wrapped(d.front - 1)
    return d.arr[d.front]
}

func (d *Deque) PushBack(element int) {
    if d.front == d.wrapped(d.back-1) { panic("deque is full") }
    d.back = d.wrapped(d.back - 1)
    d.arr[d.back] = element
}

func (d *Deque) PopBack() int {
    if d.front == d.back { panic("deque is empty") }
    el := d.arr[d.back]
    d.back = d.wrapped(d.back + 1)
    return el
}

func (d *Deque) wrapped(n int) int {
    if n < 0 { return N }
    if n > N { return 0 }
    return n
}
```

Figure 6: Double-ended queue (deque) with a fixed underlying ring buffer implemented in idiomatic Go

```
type Deque struct {
    arr   Arr
    front Nat
    back  Nat
}
func (d Deque) PushFront(el int) Deque {
    return d.succ(d.front).ifEqD(d.back, DequeFunc{d}, PushFrontFunc{d, el})
}
func (f PushFrontFunc) call() Deque {
    return Deque{
        f.d.arr.set(f.d.front.val(), f.el),
        f.d.succ(f.d.front), f.d.back}
}
func (d Deque) PopFront() Deque {
    return d.front.ifEqD(d.back, DequeFunc{d}, PopFrontFunc{d})
}
func (f PopFrontFunc) call() Deque {
    return Deque{f.d.arr, f.d.pred(f.d.front), f.d.back}
}
func (d Deque) GetFront() int {
    return d.front.ifEq(d.back,
        IntFunc{0}, ArrGetFunc{d.arr, d.pred(d.front).val()})
}
func (d Deque) PushBack(el int) Deque {
    return d.front.ifEqD(d.pred(d.back), DequeFunc{d}, PushBackFunc{d, el})
}
func (f PushBackFunc) call() Deque {
    return Deque{
        f.d.arr.set(f.d.pred(f.d.back).val(), f.el),
        f.d.front, f.d.pred(f.d.back)}
}
func (d Deque) PopBack() Deque {
    return d.front.ifEqD(d.back, DequeFunc{d}, PopBackFunc{d})
}
func (f PopBackFunc) call() Deque {
    return Deque{f.d.arr, f.d.front, f.d.succ(f.d.back)}
}
func (d Deque) GetBack() int {
    return d.front.ifEq(d.back, IntFunc{0}, ArrGetFunc{d.arr, d.back.val()})
}
```

Figure 7: Double-ended queue (deque) with a fixed underlying ring buffer implemented in FGA. Full code definitions can be found in appendix D.3.

4 Generically sized arrays

This section details a minimal design for the addition of generically sized arrays and numerical type parameters to Go. We also compare this design against the existing proposal by (Werner, 2021) and discuss future extensions to this feature. Many of the points discussed in this section were published as a proposal in the Go open source community, as an output of this work, in order to foster discussion and gather feedback (Lachowicz, 2024).

4.1 Type-set interfaces and the const interface

We've already looked at type set-based interfaces introduced in Go 1.18. If we treat array lengths as types, then we now have a conceptual set of types 0, 1, 2, 3, etc. Similarly, we can conceptually define the **const** type as the type set of all array lengths:

```
type const interface {
    0 | 1 | 2 | 3 | ...
}
```

Where the `...` means the pattern repeats for all the non-negative integers. In practice, **const** would be another "special" predeclared interface type, just like the existing `comparable` that "denotes the set of all non-interface types that are strictly comparable" (*The Go Programming Language Specification*, 2023). Just like `comparable`, **const** would be an interface that can only be used as a type parameter constraint, and not e.g. as the type of a variable, function parameter or return type. This `comparable` interface is not defined in terms of regular Go code but rather exists on the level of the language itself. **const** would follow the same pattern. The choice of the identifier **const** is to ensure backwards compatibility with existing programs, as this keyword is currently not allowed to be used as a type name. Just like a union type set, **const** can be instantiated with one of the elements of the union, i.e. a non-negative integer literal. This restricts numerical type arguments to strictly compile-time constant integers. Such a type parameter could then be used as the size of an array. With such an extension, we can express the `reversed` function from the previous section as follows:

```
func reversed[T any, N const](arr [N]T) [N]T {
    n := len(arr)
    for i := 0; i < n/2; i++ {
        arr[i], arr[n-i-1] = arr[n-i-1], arr[i]
    }
    return arr
}
```

Note how once again, the body of the function remains unchanged. The only difference is that `N` is now a type parameter bound by the `const` interface. The above function can operate on any array of any size and any element type.

The rest of this work looks at the theory and implementation of the **const** type into the existing Go language. We will examine a language called Featherweight Generic Go With Arrays (FGGA), which is a subset of Go, modulo the addition of numerical type parameters.

Since FGGA only considers "classic" (method set) interfaces, **const** will not be an interface type in FGGA, but rather in its own category. This category can be thought of as non-method set interfaces, since the two have the same restrictions, i.e. they can only be used as type parameter bounds.

4.2 Comparison with existing proposal

Shortly after the Type Parameters proposal was published (Taylor and Griesemer, 2021), a proposal to extend generics to array sizes was published (Werner, 2021). It would allow type set interfaces of the following form:

```
type Array[T any] interface {
    [...]T
}
```

In this design, the parameterisation of the array size is implicit using `...` and does not appear in the list of type parameters, meaning the numerical type parameter cannot be referenced directly. Instead, the proposal author suggests using the feature as follows:

```
type Dim interface {
    [...]struct{}
}

type Matrix[D Dim] [len(D)][len(D)]int
```

The example can be simplified slightly, by inlining the type set directly in into the type parameter constraint (allowed as of Go 1.18):

```
type Matrix[D [...]struct{}] [len(D)][len(D)]int
```

It could then be instantiated as follows:

```
myMatrix := Matrix[[5]struct{}]{}
```

Compare this with the proposal presented in this work. Using the **const** interface, the analogous code would be as follows:

```
type Matrix[D const] [D][D]int

myMatrix := Matrix[5]{}
```

This approach mandates much less boilerplate than the current proposal, as the type consumer is not forced to create "dummy" type arguments, and the type provider is not forced to retrieve an implicit numerical parameter through the `len` function. Explicit numerical type parameters would make generic arrays a first-class feature of Go, consistent with the rest of the language. All the existing compound data types in Go can already be fully type parameterised (slices: `[]T`, maps: `map[K]V` and channels: `chan T`), except for arrays, so this work would bridge that gap (`[N]T`), without making the feature feel like a

```
// Example 1: syntactically same constraints
type ArrayPair[T any, A [...]T, B [...]T] struct {
    left: A,
    right: B,
}

// Example 2: same constraints using shorthand
type ArrayPair[T any, A, B [...]T] struct {
    left: A,
    right: B,
}
```

Figure 8: Examples of ambiguity when using `[...]T` to constrain multiple type parameters

workaround. In addition, explicit numerical type parameters make the code more readable, as the programmer can immediately see when a type is parameterised on integers.

Not only would arrays become first class, but so would numerical type parameters. Currently, arrays are the only types that accept a numerical type parameter, to parameterise the length of an array type. The **const** interface would allow any type or function to accept a constant integer (or another **const** bounded type parameter) as a type argument.

The benefit of Werner's proposal is that it uses existing Go syntax: `[...]T` can already be used to denote an array's type when constructing an array literal:

```
myArray := [...]int{1, 2, 3}
```

where `myArray` has an inferred type of `[3]int`. It's worth noting, however, that this syntax is used for type inference, as opposed to denote the type of a value, similar to how in some cases type arguments can be omitted, where the compiler is able to infer what they are.

Another shortcoming of the implicit `[...]` syntax to parameterise array sizes, is that it becomes unclear whether two type parameters of the same constraint `[...]T` (as shown in figure 8) have the same length. If yes, then how do we express two type parameters of the different lengths? If not, then how do we express that two type parameters must have the same length? How about when we use the shorthand syntax to collapse the bounds of multiple type parameters? Explicit numerical type parameters make this differentiation trivial, enhancing the readability of the code.

The proposal mentions making `len` applicable to array *types* in addition to array values (as seen in Werner's examples presented above), however, as pointed out in the GitHub issue[2], this is unnecessary as the desired behaviour can already be achieved by applying `len` to an instantiated value of an array type parameter.

[2] `https://github.com/golang/go/issues/44253#issuecomment-820999513`

4.3 Allowed `const` type arguments

We've already seen how constant non-negative integers can be used as numerical type arguments. Additionally, since a numerical type parameter stands in for a constant integer, it can itself be used as a type argument. This is consistent with how in Go type parameters satisfy their own bounds, and is the basis for creating nested generic structures. Array type literal length parameters are generalized to accept a **const** type parameter, as seen in the matrix example, to fit into this new definition.

How about an expression like `N + 1` or `2 * N`? Should we allow them as **const** type arguments? The question can be phrased as: is an expression containing a **const** type parameter and constant operations (i.e. ones that can be computed at compile time, if we substitute the type parameter for a concrete integer), a constant expression? Going forwards, when the answer to the above questions is "no", it will be referred to as the conservative approach, whereas a "yes" answer will be referred to as the liberal approach.

Constant expressions evaluate to constant integers, of which non-negative ones can be used as **const** type arguments. This brings us to our first problem, not all constant expressions yield non-negative integers, and we cannot tell what the sign of a "constant" expression involving a type parameter will be until the user has instantiated a generic type or function. Go type checks generic functions/types at the declaration site, rather than the call site, so we need to ensure our approach fits that model.

Consider the signature of a function that returns the head and tail of an array (The const generics project group, 2021):

```
func headAndTail[T any, N const](arr [N]T) (T, [N - 1]T) {
    // implementation code...
}
```

Since an array length cannot be negative, it is only valid to pass arrays of size 1 or more to this function. If the argument array's length was 0, then `N - 1` would evaluate to `-1`, which is an invalid array size. Conceptually, we can think of this constraint as a new interface, a *subtype* of **const**:

```
type constPlus1 interface {
    1 | 2 | 3 | 4 | ...
}
```

This leads us to at least three potential solutions to the above problem. The first would be to fail the type checking of such a function (at declaration site), since the operation is not valid for all instantiations of the **const** interface. Just as underflow can occur when performing operations that can decrease the value of a numerical type parameter, overflow could occur if the instantiated type argument is large enough and a constant expression makes it overflow (i.e. fail to fit into an `int` type, whose size is platform dependant). This is where the first potential solution falls short, since overflow could occur with expressions such as `N + 1` or `2 * N`, those expressions would also not be allowed by same the argument of not being valid for all instantiations, and in effect, we're back to the conservative approach.

The other 2 solutions revolve around constraining the type parameter bound more tightly (as shown in the `constPlus1` interface code). This could be done implicitly: the compiler

could detect that the operation is only permitted for numerical type parameter instantiations greater than 0, and implicitly make the type parameter constraint bounds tighter. I.e. the function still type checks at the declaration site, however, callers would only be able to pass in non-zero sized arrays, which can be checked at compile time. Tools such as language servers could show these tighter constraints to the programmer. The final solution is to require an explicit tighter constraint, through some sort of new refinement syntax, e.g.:

```
func headAndTail[T any, N const[1:]](arr [N]T) (T, [N - 1]T) {
    // implementation code...
}
```

which places a lower bound of 1 on the **const** interface using the well-known slicing notation.

The downside of the liberal model is that it makes the compiler implementation significantly more complex, since it needs to determine what (if any) combination of numerical type parameter values/ranges will type-check, for every possible constant (compile-time) operation (e.g. plus, times, bitwise XOR, type conversion etc.). It can be shown that such checks could be used to perform computation at compile-time, e.g. to solve SAT formulas (A. Wagner, 2021), which can undermine Go's promise of fast compile times, if the programmer (accidentally or deliberately) misuses the type system. As such, in practice, the most reasonable roadmap to implement numerical type parameters in Go would be to start with the conservative approach. Then, if its usefulness outweighs the complexity that would be introduced into a language that strives for simplicity, progress to an explicit liberal model. Finally, if there is demand for it, consider the implicit liberal model, as a type inference feature, and have tools such as language servers show the inferred bounds to the programmer. Due to the complexity of the liberal model, Rust has also opted for the conservative approach for the time being (The const generics project group, 2021).

Appendix E.1 discusses some more complex scenarios under the liberal approach. Investigating in full detail the feasibility and safety of the liberal approach is a topic for future work. The following sections of this work consider the conservative model only.

4.4 Allowed operations on generic arrays

By the conservative approach, any operation on generic arrays must type check for all instantiations of the array. In particular, indexing an array with a constant is checked at compile time, and since the minimum array length that needs to be supported is 0, no constant is safe to index into a generic array. The programmer may wish to move this check to runtime, by first assigning the constant to a non-constant **int** variable, and then performing the index operation. The liberal approach could be used to set a lower bound on an array length, to allow index bound checks on generic arrays to be performed at compile time.

Appendix E.2 discusses the slicing operation, and E.3 discusses how the built-in `len` function can be made to work with generically sized arrays.

```
func first[T any, N const](arr [N]T) T {
    // not allowed: constant index must be valid for all array lengths
    return arr[0]
}

func first[T any, N const](arr [N]T) T {
    i := 0
    // allowed: non-constant index bounds checks are performed at runtime
    return arr[i]
}

func first[T any, N const[1:]](arr [N]T) T {
    // allowed & safe: constraint guarantees array has at least 1 element
    return arr[0]
}
```

Figure 9: Indexing operations on generic arrays

4.5 Summary of proposal

Rust's conservative approach avoids many problems and, as such forms a good starting point for introducing generically sized arrays and numerical type parameters in Go. The liberal approach is a topic for future work, as it has the potential to allow for more complex programs to be checked at compile time.

Constant expressions can be used to instantiate a **const** type parameter and array lengths. Lone **const** type parameters can be used to instantiate **const** type parameters and array lengths. Additionally **const** type parameters can be used as non-const **int** type values. More complex expressions involving a combination of type parameters, constant expressions, and constant operations (such as `+` or `len`), yield a non-constant **int** value, so they can be assigned to a variable of non-constant **int** type, but not used to instantiate **const** type parameters or array lengths.

5 Featherweight Go with Arrays

Featherweight Go is a small, functional, Turing-complete subset of the Go programming language, introduced by Griesemer et al. (2020) for the purpose of showing how generics can be added to the language. This section will extend *Featherweight Go* with arrays (FGA), as found in Go. In a similar fashion, only a subset of array features are included to keep things manageable. In particular, slices are excluded. Since FGA is a subset of Go, many constructs and syntax that are allowed in Go, are disallowed in FGA to keep the rules simpler.

A couple of notes on formal notation: a bar above a term or group of terms denotes a sequence or a rule to be applied to each element in the sequence (Steele, 2017). A sequence may contain 0 or more instances of the terms. In actual programs, various delimiters are required between terms in a sequence. Depending on the construct, this is either a comma or a semicolon (interchangeable with a newline), but these details are omitted from the formal rules. The metavariables e and $\overline{e}$ are considered distinct, i.e. unless stated explicitly, it is not automatically implied that $e \in \overline{e}$. A box around a syntactical term means it cannot appear in a normal user program but can be used internally during reduction. Rules or rule fragments appearing in grey have been taken directly from the original *Featherweight Go* (Griesemer et al., 2020) without any modification. The rules (or rule fragments) in black show the changes introduced when extending the rules to include arrays.

5.1 FGA Syntax

An FGA program is defined in a single file. The program starts with the package name, and since we limit the language to a single file, the package must be named **main**, allowing the program to be compiled into an executable in regular Go. It is then followed by a sequence of 0 or more declarations, and finally the **main** function is required at the end. The **main** function consists of a single expression, and to make it compatible with Go, the expression is assigned to the blank identifier, denoted by an underscore. This makes it so that the Go compiler does not reject the program because of an unused expression or variable.

Two main types of declarations exist: a type declaration and a method declaration. The latter is subdivided further into a regular method declaration and an "array set" method declaration. The reason for having a special syntactical form for methods that set a particular element of an array (more precisely a copy of an array, due to the value semantics of arrays), is that FGA is a functional subset of Go and does not support variable assignment in the general sense. By encapsulating the operation of creating an array copy with a certain element taking on a new value, FGA is able to maintain its functional property. It also makes the rules simpler since there is no need to introduce general assignment, yet the "array set" syntax is fully compatible with Go and will behave as expected.

A type declaration is defined as the keyword **type**, followed by a declared type name, followed by a type literal. A "declared" type name can be any identifier as defined by *The Go Programming Language Specification*, (2023), except for the predeclared **int**. In Go, it is actually legal to redefine predeclared type names, such as the built-in **int**, however, to keep things simple FGA does not permit this. That is to say, the only predeclared type name in FGA is **int**, for values of integer type. No other primitive types (e.g. **bool** or **string**) are defined in FGA.

A type literal can be one of 3 things: a struct literal, an interface literal, or an array literal. A struct type literal consists of the keyword **struct**, followed by a sequence of fields inside curly braces. Each field is a pair of a field identifier and a type name, separated by a space. A type name can be either an aforementioned declared type name or the predeclared type name **int**. The rules specify integer literals as valid type names. However, these are not allowed in user programs and are only used internally by the reduction and typing rules.

An interface type literal consists of the keyword **interface**, followed by a sequence of method specifications inside curly braces. Each method specification consists of a method name (an identifier) followed by the method signature. A method signature consists of a sequence of parameters within parenthesis, followed by a type name denoting the return type of the method. Each parameter is a pair of a variable name (an identifier) and a type name, separated by a space.

An array type literal consists of an integer literal within square brackets, denoting the length of the array followed by a type name, denoting the type of the array elements. This restricts array type literals to a single dimension, i.e. `[2][3]int` is not allowed. However, in practice, this is not a concern, as one can define a nested array in two steps, i.e. declaring the inner array as a type and using that declared type name as the element type of the outer array.

A method declaration consists of the keyword **func**, followed by the receiver in parentheses, followed by a method specification, followed by the method body. The method receiver is just a single parameter, where its type name refers to a declared value type name. A value type name is a type name defined in terms of either a struct type literal or an array type literal. The method body consists of the keyword **return**, followed by an expression, all enclosed within curly braces.

An array set method declaration is a special syntactical term in FGA, which is a valid method in Go. The receiver type refers to a declared array type name, which is a type name defined in terms of an array literal. The method can have any name (identifier), and its parameters can be any identifier as long as all the parameters are named uniquely. However, there are stricter restrictions on the rest of the construct. The return type must match the receiver type. The first parameter must have a type of **int**, while the second parameter may (syntactically) be any type name. The method body is constructed from the parameter names, where x refers to the array, x_1 to the array index, and x_2 to the new value to be assigned to that index. In figure 10, identical metavariables should be understood as identical syntactical terms in the rule defining array set methods, e.g. where a concrete term appears for x in one part of the rule, any other occurrence of the metavariable x must be instantiated with the same concrete term. The same does not apply to other rules in the figure (e.g. array index consists of two potentially distinct subexpressions). This is the only instance where, syntactically, two statements are allowed within the method body, as there is no way in Go to perform an array index assignment and return the array in a single expression.

The expressions that are supported are standard and are described in appendix F.1.

5.2 FGA Reduction

Figure 12 describes the small-step operational semantics of FGA, with auxiliary functions defined in figure 11. A value is a term that cannot be reduced further, i.e. is the final result

Field name	f	Expression	$d, e ::=$
Method name	m	Integer literal	n
Variable name	x	Variable	x
Structure type name	t_S, u_S	Method call	$e.m(\overline{e})$
Interface type name	t_I, u_I	Value literal	$t_V\{\overline{e}\}$
Array type name	t_A, u_A	Select	$e.f$
Value type name	$t_V, u_V ::= t_S \mid t_A$	Array index	$e[e]$
Declared type name	$t_D, u_D ::= t_V \mid t_I$	Addition	$e + e$
Type name	$t, u ::= t_D \mid \textbf{int} \mid \boxed{n}$		
Method signature	$M ::= (\overline{x\ t})\ t$		
Method specification	$S ::= mM$		
Type Literal	$T ::=$		
Structure	$\textbf{struct}\ \{\overline{f\ t}\}$		
Interface	$\textbf{interface}\ \{\overline{S}\}$		
Array	$[n]t$		
Declaration	$D ::=$		
Type declaration	$\textbf{type}\ t_D\ T$		
Method declaration	$\textbf{func}\ (x\ t_V)\ mM\ \{\textbf{return}\ e\}$		
Array set method declaration	$\textbf{func}\ (x\ t_A)\ m(x_1\ \textbf{int},\ x_2\ t)\ t_A\ \{x[x_1] = x_2;\ \textbf{return}\ x\}$		
Program	$P ::= \textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{\_ = e\}$		

Figure 10: FGA syntax

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

$$\frac{(\textbf{type}\ t_A\ [n]t) \in \overline{D}}{\{i \in \mathbb{Z} \mid 0 \leq i < n\} = \mathit{indexBounds}(t_A)}$$

$$\frac{(\textbf{func}\ (x\ t_A)\ m(x_1\ \textbf{int},\ x_2\ t)\ t_A\ \{x[x_1] = x_2;\ \textbf{return}\ x\}) \in \overline{D}}{\mathit{isArraySetMethod}(t_A.m)}$$

Figure 11: FGA auxiliary functions for reduction rules

of computation. In FGA, a value is either an integer literal or a value literal (struct or array), whose elements are all values themselves. An expression d reduces to expression e in a single step (denoted as $d \to e$), if one of the reduction rule templates can be instantiated with the metaexpression $d \to e$ by pattern matching.

The auxiliary predicate *isArraySetMethod* returns true if and only if the method given by the type name and method name exists in the sequence of declarations and is syntactically an array set method, as defined in figure 10. The rule R-Array-Set says that given a method call expression, where the receiver is an array literal value, the first argument is an integer literal, and the second argument is a value, evaluates to an array value literal, with the same elements but for the n^{th} index which is replaced with the value in the second argument of the method call. This rule can be applied if and only if the method in question is an array set method and the integer literal n is within the bounds of the array.

The remaining reduction rules are standard and are described in appendix F.2. An example of the reduction rules being applied to a very simple FGA program can be found in appendix H.1.

5.3 FGA Typing

Figures 14 and 15 describe the typing rules in FGA, with auxiliary functions defined in figure 13. A term is considered well-typed (or well-formed) when there exists a typing rule that the term can pattern match against, followed by the *ok* symbol. The <: symbol denotes a subtyping or "implements" relation between two types. The metaexpression $\Gamma \vdash e : t$ denotes a 3-tuple relation where an expression e is of type t in the environment Γ, which maps variables x to types t.

The first set of rules defines the subtyping relation between types. A type is said to be a subtype of another type in FGA when it *implements* the latter type. Any type implements itself (rules $<:_V$ $<:_{int}$, $<:_n$, and implicitly $<:_I$). The *methods* helper function returns the set of method specifications (later referred to as a "method set") defined on a given type (including array set methods). A type implements an interface if the type's method set is a superset of the interface's method set. By this definition, interfaces can also implement other interfaces (and themselves).

Value $v ::= t_V\{\overline{v}\} \mid n$

Evaluation context	$E ::=$
Hole	$\square$
Method call receiver	$E.m(\overline{e})$
Method call arguments	$v.m(\overline{v}, E, \overline{e})$
Addition LHS	$E + e$
Addition RHS	$n + E$
Value literal	$t_V\{\overline{v}, E, \overline{e}\}$
Select	$E.f$
Index receiver	$E[e]$
Index argument	$t_A\{\overline{v}\}[E]$

Reduction $\boxed{d \longrightarrow e}$

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

R-INDEX
$$\frac{n \in \mathit{indexBounds}(t_A)}{t_A\{\overline{v}\}[n] \longrightarrow v_n}$$

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

R-ARRAY-SET
$$\frac{n \in \mathit{indexBounds}(t_A) \qquad \mathit{isArraySetMethod}(t_A.m)}{t_A\{\overline{v}\}.m(n, v) \longrightarrow t_A\{\overline{v}\}[n := v]}$$

R-ADDITION
$$\frac{n_1 + n_2 = n}{n_1 + n_2 \longrightarrow n}$$

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

Figure 12: FGA reduction rules

A type, identified by its name, is considered well-formed when it's either the predeclared **int** type (rule T-Int-Type), or it is the type name of one of the type declarations in the program (rule T-Named). An array set method is well-formed if the 2nd parameter's type is a subtype of the receiver type's element type and the receiver type is itself a well-formed array type (rule T-Func-Arrayset).

An array literal expression is well-formed and of the array type that was instantiated, if the array type is well-formed, the number of elements matches the size of the array (note that unlike in Go, zero values are not defined in FGA, as such, all elements of an array must be explicitly instantiated upfront), and all elements' types are subtypes of the array's element type (rule T-Array-Literal).

For reasons that will be discussed when type parameters are introduced, integer literals are of their own types, e.g. the expression 1 is of type 1 (rule T-Int-Literal). Integer literal types are subtypes of the **int** type, e.g. $1 <:$ **int** (rule $<:_{int-n}$). Go does not make such a distinction (i.e. each integer literal being its own type), however, integer literals (constants) in Go may be "untyped", which can be used as subtypes of other numerical types (e.g. **int**, **byte**, **int16** etc.). In FGA, integer literals are treated similarly to untyped integer constants in Go (*The Go Programming Language Specification*, 2023).

Because of the distinction between constant (literal) and non-constant integer types, there are two typing rules for an array index expression. In both cases, the array expression must be well-typed and of an array type, and the array index expression's type is of the element type of the array. When the index expression is of the non-constant **int** type, no other checks are performed (statically) (rule T-Array-Index). However, when the index expression is of an integer literal type, an index bounds check is also statically performed via the type system (rule T-Array-Index-Literal).

$$\frac{(\textbf{type}\ t_A\ [n]t) \in \overline{D}}{t = elementType(t_A)} \qquad \frac{(\textbf{type}\ t_A\ [n]t) \in \overline{D}}{n = lenType(t_A)} \qquad \frac{}{methods(\textbf{int}) = \{\}} \qquad \frac{}{methods(n) = \{\}}$$

$$methods(t_V) = \{mM \mid (\textbf{func}\ (x\ t_V)\ mM\ \{\textbf{return}\ e\}) \in \overline{D}\} \cup$$
$$\{m(x_1\ \textbf{int},\ x_2\ t)\ t_V \mid (\textbf{func}\ (x\ t_V)\ m(x_1\ \textbf{int},\ x_2\ t)\ t_V\ \{x[x_1] = x_2;\ \textbf{return}\ x\}) \in \overline{D}\}$$

$$\frac{\textbf{type}\ t_I\ \textbf{interface}\{\overline{S}\} \in \overline{D}}{methods(t_I) = \overline{S}} \qquad \frac{distinct(\overline{m})}{unique(\overline{mM})} \qquad \frac{a_i \neq a_j\ \forall a_i, a_j \in \overline{a},\ i \neq j}{distinct(\overline{a})}$$

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

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

$$\frac{}{notReferenced(\overline{t_r}, \textbf{int})} \qquad \frac{}{notReferenced(\overline{t_r},\ \textbf{interface}\ \{\overline{S}\})}$$

$$\frac{notReferenced(\overline{t_r}, t)}{notReferenced(\overline{t_r},\ [n]t)} \qquad \frac{notReferenced(\overline{t_r}, t)\ \forall t \in \overline{f\ t}}{notReferenced(\overline{t_r},\ \textbf{struct}\{\overline{f\ t}\})}$$

$$\frac{(\textbf{type}\ t_D\ T) \in \overline{D} \qquad \overline{t_r \neq t_D} \qquad notReferenced(\overline{t_r}, t_D, T)}{notReferenced(\overline{t_r}, t_D)}$$

Figure 13: FGA auxiliary functions for typing rules

The remainder of the rules are described in appendix F.3. An example of the typing rules being applied to a very simple FGA program can be found in appendix H.2.

5.4 FGA Properties

An array index expression or an array-set method call expression panics if they contain an array type t_A, and an array index n, where $n \notin indexBounds(t_A)$. An expression e panics if $e = E[d]$, where E is any evaluation context, and d is an expression that panics.

The progress and preservation properties covered in *Featherweight Go* apply to FGA and are defined as follows (Griesemer et al., 2020):

Property 5.1 (Preservation) *If $\emptyset \vdash d : u$ and $d \longrightarrow e$ then $\emptyset \vdash e : t$ for some t with $t <: u$.*

Property 5.2 (Progress) *If $\emptyset \vdash d : u$ then either d is a value, $d \longrightarrow e$ for some e, or d panics.*

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

$$\frac{}{t_V <: t_V}\ <:_V \qquad \frac{}{\textbf{int} <: \textbf{int}}\ <:_{int} \qquad \frac{}{n <: n}\ <:_n \qquad \frac{}{n <: \textbf{int}}\ <:_{int-n} \qquad \frac{methods(t) \supseteq methods(t_I)}{t <: t_I}\ <:_I$$

$$\frac{}{\textbf{int}\ ok}\ \text{T-INT-TYPE} \qquad \frac{(\textbf{type}\ t\ T) \in \overline{D}}{t\ ok}\ \text{T-NAMED}$$

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

$$\frac{n \geq 0 \qquad t\ ok}{[n]t\ ok}\ \text{T-ARRAY}$$

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

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

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

$$\frac{u = elementType(t_A) \qquad t <: u \qquad t_A\ ok}{\textbf{func}\ (x\ t_A)\ m(x_1\ \textbf{int},\ x_2\ t)\ t_A\ \{x[x_1] = x_2;\ \textbf{return}\ x\}}\ \text{T-FUNC-ARRAYSET}$$

Figure 14: FGA typing rules (1 of 2)

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

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

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

$$\textsc{T-ARRAY-LITERAL}\ \frac{t_A\ \mathit{ok} \qquad |\overline{e}| = \mathit{lenType}(t_A) \qquad \Gamma \vdash \overline{e : t} \qquad u = \mathit{elementType}(t_A) \qquad \overline{t <: u}}{\Gamma \vdash t_A\{\overline{e}\} : t_A}$$

$$\textsc{T-INT-LITERAL}\ \frac{}{\Gamma \vdash n : n} \qquad \textsc{T-ARRAY-INDEX}\ \frac{\Gamma \vdash e_1 : t_A \qquad \Gamma \vdash e_2 : \mathbf{int} \qquad t = \mathit{elementType}(t_A)}{\Gamma \vdash e_1[e_2] : t}$$

$$\textsc{T-ARRAY-INDEX-LITERAL}\ \frac{\Gamma \vdash e_1 : t_A \qquad \Gamma \vdash e_2 : n \qquad 0 \le n < \mathit{lenType}(t_A) \qquad t = \mathit{elementType}(t_A)}{\Gamma \vdash e_1[e_2] : t}$$

$$\textsc{T-INT-LITERAL-ADDITION}\ \frac{\Gamma \vdash e_1 : n_1 \qquad \Gamma \vdash e_2 : n_2 \qquad n_1 + n_2 = n}{\Gamma \vdash e_1 + e_2 : n}$$

$$\textsc{T-INT-ADDITION}\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2 \qquad t_1 <: \mathbf{int} \qquad t_2 <: \mathbf{int} \qquad \mathbf{int} \in \{t_1, t_2\}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}}$$

Programs $\boxed{P\ \mathit{ok}}$

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

Figure 15: FGA typing rules (2 of 2)

6 Featherweight Generic Go with Arrays

Featherweight Generic Go (FGG) is an extension to *Featherweight Go* that introduces generics via type parameters, formalised in (Griesemer et al., 2020). This section adapts FGG with the syntax and restrictions introduced in the Type Parameters Proposal and ultimately implemented in Go v1.18 (Taylor and Griesemer, 2021), and extends it with arrays and numerical type parameters that can be used to create generic array types, referred to as FGGA going forwards.

Rules or rule fragments that remain unchanged from FGA are shown in grey, whereas new rules or rule modifications are shown in black.

6.1 FGGA Syntax

Syntactically, a type is now one of 4 things — it is either a type parameter, which is just an identifier, an integer literal, the keyword **const** (which cannot be used as a regular type in user programs) or a named type. Named types are defined in terms of type names from FGA, with the addition of a sequence of zero or more type arguments enclosed in square brackets, following the type name. The type arguments are themselves types. While it's not explicitly stated in figure 16, when there are no type arguments, the square brackets *must* be omitted, to ensure compatibility with Go. The consequence of this is that syntactically, there is no way to differentiate between a type parameter and a named type with no type arguments. A value type is a named type whose base type name (i.e. outermost type name) is a value type name. The situation is analogous for interface types. An integer-like type is either an integer literal, or a type parameter, and can be used as the size of an array type literal. A bound is used to restrict the set of type arguments that can be used in place of a type parameter. A type parameter constraint Φ is used to define a type parameter along with its constraint (bound).

Method signatures, struct type literals, array type literals and value literals are updated to used types in the place of type names. Type declarations now have a sequence of type parameter constraints following the type name, that are required when instantiating the type. Similarly to type arguments, a type with no type parameters *must* omit the square brackets.

Unlike in FGG, method receivers are invariant, as such, there is no need to repeat the type parameter constraints on the receiver type parameters. Instead, a simple type parameter sequence is used. In Go, these type parameters can be named differently from type parameters in the type declaration, since the order of the parameters is sufficient to identify them, however, to keep the rules simpler, in FGGA the type parameter names must match exactly with the ones specified in the type declaration. Another feature omitted from the current implementation of generics in Go, but present in FGG, are method-specific type parameters. In order to be compatible with the current implementation, these constructs are also omitted from FGGA.

Field name	f	Type	$\tau, \sigma ::=$
Method name	m	Type parameter	α
Variable name	x	Named type	$t[\overline{\tau}]$
Structure type name	t_S, u_S	Integer literal	n
Interface type name	t_I, u_I	Constant	$\boxed{\textbf{const}}$
Array type name	t_A, u_A	Value type	$\tau_V, \sigma_V ::= t_V[\overline{\tau}]$
Value type name	$t_V, u_V ::= t_S \mid t_A$	Interface type	$\tau_I, \sigma_I ::= t_I[\overline{\tau}]$
Declared type name	$t_D, u_D ::= t_V \mid t_I$	Integer-like type	$\tau_n, \sigma_n ::= \alpha \mid n$
Type name	$t, u ::= t_D \mid \textbf{int}$	Bound	$\gamma ::= \tau_I \mid \textbf{const}$
Type parameter	α	Type parameter	
Method signature	$M ::= (\overline{x\ \tau})\ \tau$	constraint	$\Phi ::= \alpha\ \gamma$
Method specification	$S ::= mM$	Expression	$e ::=$
Type Literal	$T ::=$	Integer literal	n
Structure	$\textbf{struct}\ \{\overline{f\ \tau}\}$	Variable	x
Interface	$\textbf{interface}\ \{\overline{S}\}$	Method call	$e.m(\overline{e})$
Array	$[\tau_n]\tau$	Value literal	$\tau_V\{\overline{e}\}$
		Select	$e.f$
		Array index	$e[e]$
Declaration	$D ::=$	Addition	$e + e$
Type declaration	$\textbf{type}\ t[\overline{\Phi}]\ T$		
Method declaration	$\textbf{func}\ (x\ t_V[\overline{\alpha}])\ mM\ \{\textbf{return}\ e\}$		
Array set method declaration	$\textbf{func}\ (x\ t_A[\overline{\alpha}])\ m(x_1\ \textbf{int},\ x_2\ \tau)\ t_A[\overline{\alpha}]\ \{x[x_1] = x_2;\ \textbf{return}\ x\}$		
Program	$P ::= \textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{\_ = e\}$		

Figure 16: FGGA syntax

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

$$\frac{(\textbf{func}\ (x\ t_V[\overline{\alpha}])\ m(\overline{x\ \tau})\ \tau\ \{\textbf{return}\ e\}) \in \overline{D} \qquad \theta = (\overline{\alpha := \sigma})}{\mathit{body}(t_V[\overline{\sigma}].m) = (x : t_V[\overline{\sigma}], \overline{x : \tau}).e[\![\theta]\!]}$$

$$\frac{(\textbf{type}\ t_A[\overline{\Phi}]\ [n]\tau) \in \overline{D} \qquad \tau_A = t_A[\overline{\tau}]}{\{i \in \mathbb{Z} \mid 0 \leq i < n\} = \mathit{indexBounds}(\tau_A)}$$

$$\frac{(\textbf{type}\ t_A[\overline{\Phi}]\ [\alpha_i]\tau) \in \overline{D} \qquad (\overline{\alpha\ \gamma}) = \overline{\Phi} \qquad \tau_A = t_A[\overline{\tau}] \qquad n = \tau_i}{\{i \in \mathbb{Z} \mid 0 \leq i < n\} = \mathit{indexBounds}(\tau_A)}$$

$$\frac{(\textbf{func}\ (x\ t_A[\overline{\alpha}])\ m(x_1\ \textbf{int},\ x_2\ \tau)\ t_A[\overline{\alpha}]\ \{x[x_1] = x_2;\ \textbf{return}\ x\}) \in \overline{D} \qquad \tau_A = t_A[\overline{\tau}]}{\mathit{isArraySetMethod}(\tau_A.m)}$$

Figure 17: FGGA auxiliary functions for reduction rules

6.2 FGGA Reduction

The reduction rules for FGGA are nigh identical to the ones found in FGA, with the only notable difference being that type names t are replaced with types τ.

More interesting differences occur in the auxiliary functions. In FGA, *indexBounds* took a type name and performed a simple lookup in the type declaration to extract the size of the array. In FGGA, there are two cases of the *indexBounds* function. When the size of the array type is an integer literal, then the bounds are calculated in the same way as in FGA, based on a lookup of the named type declaration of the array type. If however, the array type declaration has a type parameter in place of the array size, then the bounds are calculated from the integer literal type argument in the array type that corresponds to the type parameter used as the array size in the declaration. The correspondence is determined by matching on the same position in the sequence of type parameter constraints and the sequence of type arguments.

The *body* auxiliary function has also been updated to perform substitution of method type parameters with the receiver's type arguments within the method expression. A map within a pair of double square brackets applied to a term (e.g. $e[\![\theta]\!]$ in the *body* function) denotes a substitution application in this and following rules. The type arguments are mapped to the type parameters based on their respective positions.

Similarly, the *fields* auxiliary function has been updated to perform a type parameter substitution with type arguments on the resulting struct fields. This is performed analogously to how it's done in the *body* function, except that type parameters are extracted from the type parameter constraints in the struct type declaration.

$$\text{Value} \qquad v ::= \tau_V\{\overline{v}\} \mid n$$

Evaluation context	$E ::=$
Hole	$\square$
Method call receiver	$E.m(\overline{e})$
Method call arguments	$v.m(\overline{v}, E, \overline{e})$
Addition LHS	$E + e$
Addition RHS	$n + E$

Value literal	$\tau_V\{\overline{v}, E, \overline{e}\}$
Select	$E.f$
Index receiver	$E[e]$
Index argument	$\tau_A\{\overline{v}\}[E]$

Reduction $\boxed{d \longrightarrow e}$

$$\text{R-FIELD}\ \frac{(\overline{f\ \tau}) = \mathit{fields}(\tau_S)}{\tau_S\{\overline{v}\}.f_i \longrightarrow v_i} \qquad \text{R-INDEX}\ \frac{n \in \mathit{indexBounds}(\tau_A)}{\tau_A\{\overline{v}\}[n] \longrightarrow v_n} \qquad \text{R-CALL}\ \frac{(x : \tau_V, \overline{x : \tau}).e = \mathit{body}(\mathit{type}(v).m)}{v.m(\overline{v}) \longrightarrow e[x := v, \overline{x := v}]}$$

$$\text{R-ARRAY-SET}\ \frac{n \in \mathit{indexBounds}(\tau_A) \qquad \mathit{isArraySetMethod}(\tau_A.m)}{\tau_A\{\overline{v}\}.m(n, v) \longrightarrow \tau_A\{\overline{v}\}[n := v]} \qquad \text{R-ADDITION}\ \frac{n_1 + n_2 = n}{n_1 + n_2 \longrightarrow n} \qquad \text{R-CONTEXT}\ \frac{d \longrightarrow e}{E[d] \longrightarrow E[e]}$$

Figure 18: FGGA reduction rules

6.3 FGGA Typing

Δ is defined as a typing environment mapping type parameters α to their bounds γ. Type parameter constraint sequences $\overline{\Phi}$ may implicitly coerce to type environments. As before in FGA, Γ is defined as an environment mapping variables x to types τ.

The "implements" relation now includes a typing environment Δ. As before, all types implement themselves (rules $<:_{\text{Param}}$, $<:_V$, $<:_{int}$, $<:_{const}$ and $<:_n$), and integer literal types implement the **int** type (rule $<:_{int-n}$). Non-negative integer literals now also implement the **const** type (rule $<:_{const-n}$), so that they can be used as type arguments, where the bound of the type parameter is **const**. Since the use case of numerical type parameters is to generically size arrays, and arrays cannot be of a negative size, negative integers do not implement **const**.

The $methods_\Delta$ auxiliary now also accepts a typing environment as input (denoted by a subscript Δ), as a type parameter bound lookup may be necessary to retrieve the methods of a type. For a named value type $t_V[\overline{\tau}]$ $methods_\Delta$ returns the set of method specifications of the base value type t_V, with type substitutions performed on them. E.g. if the declared value type `Foo` with a type parameter `T` has a method with the specification `f(x T) T`, then $methods_\Delta$(`Foo[int]`), where Δ is any typing environment (as it is not relevant in the case of looking up methods of a named value type), would return {(`f(x int) int`)}. The implication of this is that one instantiation of a generic type could implement a certain interface, while another could not, depending on the type arguments. A $methods_\Delta$ lookup on an interface has similarly been updated to return a type-substituted method set. Additionally, the method set of a type parameter α is equal to the method set of its bound γ, as specified in the typing environment Δ.

The integer literal type can now appear in user programs, although with restrictions as to where they can be used. They can only be used as type arguments in a named type and as the size parameter of an array type literal but not e.g. as a standalone type of a variable or a return type. Const type parameters (i.e. those that are bound by **const**) have the exact same restrictions as integer literal types. To distinguish between these two kinds of user-program types, the $isConst_\Delta$ predicate was created, which simply checks if the type is a subtype of **const**. Because both **const** type arguments and array sizes must be non-negative, we can also restrict all integer literal types to be non-negative (rule T-N-Type).

A type parameter α is considered well-typed if it appears in the typing environment Δ (rule T-Param). The rule T-Named has been updated to type-check each type argument in the named type, recursively applying one of "well-formed type" rules ($\Delta \vdash \tau\ ok$) on each argument, and checking whether each argument satisfies the type parameter bound via the $(\overline{\Phi :=_\Delta \tau})$ lookup. The lookup differs from the regular type substitution map lookup (denoted without the subscript Δ), that it additionally checks the bounds of each type argument via the subtyping relation, which itself is done via a type substitution on the type parameters as well as the bounds, because bounds may refer to other type parameters in the type parameter constraints sequence. Not only must the type arguments implement the type-substituted bounds of their corresponding type parameters, but they must also have an equal "**const**-ness", i.e. either both the argument and bound return true for *isConst*, or they both return false. The need for this check can be illustrated with the following example: `2 <: any` holds, but 2 cannot be used as a variable type, and so it cannot be used to instantiate a type parameter bound by `any`. This is enforced in the rule because $isConst_\Delta(2) \neq isConst_\Delta(\texttt{any})$. Both the type-checking and non type-checking lookups yield a map from type parameters to types, denoted by η.

The rule T-Formal specifies what it means for type parameter constraint sequences (i.e. the formal type parameters found in a type declaration) to be well typed. All type parameters must be distinct, and their bounds must be well-typed, in the context of the typing environment that is created from the type parameter constraints that are being checked. This environment is necessary for recursively defined type parameter bounds, e.g. if `Eq` is an interface with one type parameter, then we can define another type `Foo` with a type parameter α bound by `Eq[`α`]`, i.e. the type parameter is referenced within its own bound.

There is a restriction in the current implementation of Go, that FGGA also adheres to, and that is that no type in any type parameter bound can refer to the type being declared (directly or indirectly). E.g. the interface type declaration `Eq` cannot have a type parameter α bound by `Eq[`α`]` itself (*The Go Programming Language Specification*, 2023).

The check for this restriction is performed via the *notReferenced* auxiliary in the T-Type rule for each type parameter bound γ. *notReferenced* is defined recursively, and described in appendix G.1.1.

An array size can now be any valid **const** type, and the element type must be a valid non-**const** type, evaluated in the typing environment $\overline{\Phi}$. The *lenType* auxiliary now has two cases. One as before: when the array size is an integer literal (i.e. non-generic). The other is the generic case, where the array size is a type parameter, in which case *lentype* returns the type argument that is used to instantiate the array size type parameter (by matching on the type parameter position). Array literals in FGGA can only be constructed when the *lenType* is an integer literal type — either when the array length type is non-generic, or

has been instantiated with an integer literal (rule T-Array-Literal). When the array size is instantiated with a type parameter, the size is unknown, hence it is not safe to assume the array can hold any elements (e.g. it could be instantiated later with a size of 0). In practice, Go's zero values could be used to instantiate generically sized arrays, but since FGGA does not support them, even empty generic array initialisation is not allowed.

It is worth noting that generic, non-instantiated arrays (i.e. where *lenType* does not return an integer literal) can only be indexed using a non-constant integer of type **int**. This is to stay consistent with current expectations: indexing into an array using a constant integer literal is only allowed to fail at compile-time. In line with the Go spirit of type-checking generic code at the declaration site, as opposed to the call site, indexing into generic arrays using integer literals is not allowed in FGGA, even if the index would remain within bounds for all array instantiations in the scope of the current program. The programmer may still achieve such behaviour by explicitly assigning the integer literal to a non-const **int** variable and then performing an index operation.

6.4 FGGA Properties

As before, an array index expression or an array-set method call expression panics if they contain an array type τ_A, and an array index n, where $n \notin \mathit{indexBounds}(\tau_A)$. An expression e panics if $e = E[d]$, where E is any evaluation context, and d is an expression that panics.

The progress and preservation properties covered in *Featherweight Go* apply to FGGA and are defined as follows (Griesemer et al., 2020):

Property 6.1 (Preservation) *If* $\emptyset;\emptyset \vdash d : \sigma$ *and* $d \longrightarrow e$ *then* $\emptyset;\emptyset \vdash e : \tau$ *for some* τ *with* $\tau <: \sigma$.

Property 6.2 (Progress) *If* $\emptyset;\emptyset \vdash d : \sigma$ *then either* d *is a value,* $d \longrightarrow e$ *for some* e*, or* d *panics.*

$$\frac{(\textbf{type}\ t_A[\overline{\Phi}]\ [\tau_n]\tau) \in \overline{D}}{\tau = \mathit{elementType}(t_A)} \qquad \frac{\tau_A = t_A[\overline{\tau}] \qquad (\overline{\alpha\ \gamma}) = \overline{\Phi} \qquad (\textbf{type}\ t_A[\overline{\Phi}]\ [\alpha_i]\tau) \in \overline{D}}{\tau_i = \mathit{lenType}(\tau_A)}$$

$$\frac{\tau_A = t_A[\overline{\tau}] \qquad (\textbf{type}\ t_A[\overline{\Phi}]\ [n]\tau) \in \overline{D}}{n = \mathit{lenType}(\tau_A)} \qquad \frac{(\textbf{type}\ t[\overline{\Phi}]\ T) \in \overline{D}}{\overline{\Phi} = \mathit{typeParams}(t)} \qquad \frac{\Delta \vdash \tau <: \textbf{const}}{\mathit{isConst}_\Delta(\tau)}$$

$$\frac{(\overline{\alpha\ \gamma}) = \overline{\Phi} \qquad \eta = (\overline{\Phi := \tau}) \qquad \Delta \vdash (\overline{\alpha <: \gamma})[\![\eta]\!] \qquad \overline{\mathit{isConst}_\Delta(\alpha) = \mathit{isConst}_\Delta(\gamma)[\![\eta]\!]}}{(\overline{\Phi :=_\Delta \tau}) = \eta}$$

$$\overline{\mathit{notReferenced}_\alpha(\overline{t_r},\ n)} \qquad \overline{\mathit{notReferenced}_\alpha(\overline{t_r},\ \textbf{const})} \qquad \overline{\mathit{notReferenced}_\alpha(\overline{t_r},\ \alpha)}$$

$$\frac{\overline{\mathit{notReferenced}_\alpha(\overline{t_r},\ \tau)} \qquad \overline{t_r \neq t} \qquad (\overline{\alpha\ \gamma}) = \mathit{typeParams}(t) \qquad \overline{\mathit{notReferenced}_\alpha(\overline{t_r}, t, \gamma)}}{\mathit{notReferenced}_\alpha(\overline{t_r},\ t[\overline{\tau}])}$$

$$\overline{\mathit{notReferenced}(\overline{t_r}, \alpha)} \qquad \overline{\mathit{notReferenced}(\overline{t_r}, \textbf{int})} \qquad \overline{\mathit{notReferenced}(\overline{t_r},\ \textbf{interface}\ \{\overline{S}\})}$$

$$\frac{\mathit{notReferenced}(\overline{t_r}, \tau)}{\mathit{notReferenced}(\overline{t_r},\ [\tau_n]\tau)} \qquad \frac{\mathit{notReferenced}(\overline{t_r}, \tau)\ \forall \tau \in \overline{f\ \tau}}{\mathit{notReferenced}(\overline{t_r},\ \textbf{struct}\{\overline{f\ \tau}\})}$$

$$\frac{(\textbf{type}\ t_D[\overline{\Phi}]\ T) \in \overline{D} \qquad \overline{t_r \neq t_D} \qquad \eta = (\overline{\Phi := \tau}) \qquad \mathit{notReferenced}(\overline{t_r}, t_D, T[\![\eta]\!])}{\mathit{notReferenced}(\overline{t_r}, t_D[\overline{\tau}])}$$

$$\overline{\mathit{methods}_\Delta(\textbf{int}) = \{\}} \qquad \overline{\mathit{methods}_\Delta(n) = \{\}}$$

$$\frac{\eta = (\overline{\Phi := \tau}) \qquad \overline{\Phi} = \mathit{typeParams}(t_V)}{\begin{array}{c}\mathit{methods}_\Delta(t_V[\overline{\tau}]) = \{(mM)[\![\eta]\!] \mid (\textbf{func}\ (x\ t_V[\overline{\alpha}])\ mM\ \{\textbf{return}\ e\}) \in \overline{D}\} \cup \\ \{m(x_1\ \textbf{int},\ x_2\ \tau[\![\eta]\!])\ t_V[\overline{\tau}] \mid (\textbf{func}\ (x\ t_V[\overline{\alpha}])\ m(x_1\ \textbf{int},\ x_2\ \tau)\ t_V[\overline{\alpha}]\ \{x[x_1] = x_2;\ \textbf{return}\ x\}) \in \overline{D}\}\end{array}}$$

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

$$\frac{\mathit{distinct}(\overline{m})}{\mathit{unique}(\overline{mM})} \qquad \frac{a_i \neq a_j\ \forall a_i, a_j \in \overline{a},\ i \neq j}{\mathit{distinct}(\overline{a})} \qquad \mathit{tdecls}(\overline{D}) = [t_D \mid (\textbf{type}\ t_D[\overline{\Phi}]\ T) \in \overline{D}]$$

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

Figure 19: FGG auxiliary functions for typing with arrays

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

$$\frac{}{\Delta \vdash \alpha \mathbin{<:} \alpha}\ \mathbin{<:}\textsc{Param} \qquad \frac{}{\Delta \vdash \tau_V \mathbin{<:} \tau_V}\ \mathbin{<:}_V \qquad \frac{}{\Delta \vdash \mathbf{int} \mathbin{<:} \mathbf{int}}\ \mathbin{<:}_{int} \qquad \frac{}{\Delta \vdash \mathbf{const} \mathbin{<:} \mathbf{const}}\ \mathbin{<:}_{const} \qquad \frac{}{\Delta \vdash n \mathbin{<:} n}\ \mathbin{<:}_n$$

$$\frac{}{\Delta \vdash n \mathbin{<:} \mathbf{int}}\ \mathbin{<:}_{int-n} \qquad \frac{methods_\Delta(\tau) \supseteq methods_\Delta(\tau_I)}{\Delta \vdash \tau \mathbin{<:} \tau_I}\ \mathbin{<:}_I \qquad \frac{n \geq 0}{\Delta \vdash n \mathbin{<:} \mathbf{const}}\ \mathbin{<:}_{const-n} \qquad \frac{(\alpha : \mathbf{const}) \in \Delta}{\Delta \vdash \alpha \mathbin{<:} \mathbf{const}}\ \mathbin{<:}_{const-\textsc{Param}}$$

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

$$\frac{n \geq 0}{\Delta \vdash n\ ok}\ \textsc{T-N-TYPE} \qquad \frac{}{\Delta \vdash \mathbf{int}\ ok}\ \textsc{T-INT-TYPE} \qquad \frac{(\alpha : \gamma) \in \Delta}{\Delta \vdash \alpha\ ok}\ \textsc{T-PARAM}$$

$$\frac{\Delta \vdash \overline{\tau\ ok} \qquad (\mathbf{type}\ t[\overline{\Phi}]\ T) \in \overline{D} \qquad \eta = (\overline{\Phi :=_\Delta \tau})}{\Delta \vdash t[\overline{\tau}]\ ok}\ \textsc{T-NAMED}$$

Well-formed type formals $\boxed{\Delta \vdash \mathbf{const}\ ok}$ $\boxed{\overline{\Phi}\ ok}$

$$\frac{}{\Delta \vdash \mathbf{const}\ ok}\ \textsc{T-CONST} \qquad \frac{(\overline{\alpha\ \gamma}) = \overline{\Phi} \qquad distinct(\overline{\alpha}) \qquad \overline{\Phi} \vdash \overline{\gamma\ ok}}{\overline{\Phi}\ ok}\ \textsc{T-FORMAL}$$

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

$$\frac{distinct(\overline{x}) \qquad \overline{\Phi} \vdash \overline{\tau\ ok} \qquad \overline{\Phi} \vdash \tau\ ok \qquad \overline{\neg isConst_{\overline{\Phi}}(\tau)} \qquad \neg isConst_{\overline{\Phi}}(\tau)}{\overline{\Phi} \vdash m(\overline{x\ \tau})\ \tau\ ok}\ \textsc{T-SPECIFICATION}$$

$$\frac{distinct(\overline{f}) \qquad \overline{\Phi} \vdash \overline{\tau\ ok} \qquad \overline{\neg isConst_{\overline{\Phi}}(\tau)}}{\overline{\Phi} \vdash \mathbf{struct}\ \{\overline{f\ \tau}\}\ ok}\ \textsc{T-STRUCT} \qquad \frac{unique(\overline{S}) \qquad \overline{\Phi} \vdash \overline{S\ ok}}{\overline{\Phi} \vdash \mathbf{interface}\ \{\overline{S}\}}\ \textsc{T-INTERFACE}$$

$$\frac{\overline{\Phi} \vdash \tau_n\ ok \qquad isConst_{\overline{\Phi}}(\tau_n) \qquad \overline{\Phi} \vdash \tau\ ok \qquad \neg isConst_{\overline{\Phi}}(\tau)}{\overline{\Phi} \vdash [\tau_n]\tau\ ok}\ \textsc{T-ARRAY}$$

Figure 20: FGGA typing rules (1 of 3)

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

T-TYPE

$$\frac{\overline{\Phi\ ok} \qquad \overline{\Phi} = (\overline{\alpha\ \gamma}) \qquad \overline{notReferenced_{\alpha}(t, \gamma)} \qquad \overline{\Phi} \vdash T\ ok \qquad notReferenced(t,\ T)}{\textbf{type}\ t[\overline{\Phi}]\ T\ ok}$$

T-FUNC

$$\frac{\begin{array}{lll} & distinct(x, \overline{x}) & \overline{\Phi} = typeParams(t_V) \\ (\overline{\alpha\ \gamma}) = \overline{\Phi} & \overline{\Phi} \vdash m(\overline{x\ \tau})\ \sigma\ ok & \overline{\Phi}; x : t_V[\overline{\alpha}],\ \overline{x : \tau} \vdash e : \tau \qquad \overline{\Phi} \vdash \tau <: \sigma \end{array}}{\textbf{func}\ (x\ t_V[\overline{\alpha}])\ m(\overline{x\ \tau})\ \sigma\ \{\textbf{return}\ e\}\ ok}$$

T-FUNC-ARRAYSET

$$\frac{\sigma = elementType(t_A) \qquad \overline{\Phi} = typeParams(t_A) \qquad (\overline{\alpha\ \gamma}) = \overline{\Phi} \qquad \overline{\Phi} \vdash \tau <: \sigma}{\textbf{func}\ (x\ t_A[\overline{\alpha}])\ m(x_1\ \textbf{int},\ x_2\ \tau)\ t_A[\overline{\alpha}]\ \{x[x_1] = x_2;\ \textbf{return}\ x\}}$$

Figure 21: FGGA typing rules (2 of 3)

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

T-INT-LITERAL
$$\frac{}{\Delta;\ \Gamma \vdash n : n}$$

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

T-CALL
$$\frac{\Delta;\ \Gamma \vdash e : \tau \qquad \Delta;\ \Gamma \vdash \overline{e : \tau} \qquad (m(\overline{x\ \sigma})\ \sigma) \in \mathit{methods}_\Delta(\tau) \qquad \Delta \vdash \overline{\tau <: \sigma}}{\Delta;\ \Gamma \vdash e.m(\overline{e}) : \sigma}$$

T-ARRAY-LITERAL
$$\frac{\begin{array}{c}\Delta \vdash \tau_A\ \mathit{ok} \\ \overline{\Phi} = \mathit{typeParams}(t_A) \qquad \tau_A = t_A[\overline{\sigma}] \qquad \eta = (\overline{\Phi := \sigma}) \qquad \sigma = \mathit{elementType}(t_A)[\![\eta]\!] \\ \mathit{lenType}(\tau_A) <: n \qquad |\overline{e}| = \mathit{lenType}(\tau_A) \qquad \Delta;\ \Gamma \vdash \overline{e : \tau} \qquad \Delta \vdash \overline{\tau <: \sigma}\end{array}}{\Delta;\ \Gamma \vdash \tau_A\{\overline{e}\} : \tau_A}$$

T-ARRAY-INDEX
$$\frac{\begin{array}{c}\overline{\Phi} = \mathit{typeParams}(t_A) \qquad \tau_A = t_A[\overline{\tau}] \\ \eta = (\overline{\Phi := \tau}) \qquad \tau = \mathit{elementType}(t_A)[\![\eta]\!] \qquad \Delta;\ \Gamma \vdash e_1 : \tau_A \qquad \Delta;\ \Gamma \vdash e_2 : \mathbf{int}\end{array}}{\Delta;\ \Gamma \vdash e_1[e_2] : \tau}$$

T-ARRAY-INDEX-LITERAL
$$\frac{\begin{array}{c}\overline{\Phi} = \mathit{typeParams}(t_A) \qquad \tau_A = t_A[\overline{\tau}] \qquad \eta = (\overline{\Phi := \tau}) \qquad \tau = \mathit{elementType}(t_A)[\![\eta]\!] \\ \Delta;\ \Gamma \vdash e_1 : \tau_A \qquad \Delta;\ \Gamma \vdash e_2 : n \qquad \mathit{lenType}(\tau_A) <: n_\tau \qquad 0 \le n < \mathit{lenType}(\tau_A)\end{array}}{\Delta;\ \Gamma \vdash e_1[e_2] : \tau}$$

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

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

T-INT-LITERAL-ADDITION
$$\frac{\Delta;\ \Gamma \vdash e_2 : n_2 \qquad \Delta;\ \Gamma \vdash e_1 : n_1 \qquad n_1 + n_2 = n}{\Delta;\ \Gamma \vdash e_1 + e_2 : n}$$

T-INT-ADDITION
$$\frac{\Delta;\ \Gamma \vdash e_1 : \tau_1 \qquad \Delta;\ \Gamma \vdash e_2 : \tau_2 \qquad \tau_1 <: \mathbf{int} \qquad \tau_2 <: \mathbf{int} \qquad \mathbf{int} \in \{\tau_1, \tau_2\}}{\Delta;\ \Gamma \vdash e_1 + e_2 : \mathbf{int}}$$

Programs $\boxed{P\ \mathit{ok}}$

T-PROG
$$\frac{\mathit{distinct}(\mathit{tdecls}(\overline{D}), \mathbf{int}) \qquad \mathit{distinct}(\mathit{mdecls}(\overline{D})) \qquad \overline{D\ \mathit{ok}} \qquad \emptyset;\ \emptyset \vdash e : \tau}{\mathbf{package\ main};\ \overline{D}\ \mathbf{func\ main}()\ \{\_ = e\}\ \mathit{ok}}$$

Figure 22: FGGA typing rules (3 of 3)

7 Monormorphisation from FGGA to Go

Monormorphisation is an implementation technique for generics (parametric polymorphism) which involves translating generic code into non-generic code. For every instantiation of a generic type, a non-generic type is produced in the output monomorphised program. This approach allows for zero-cost abstractions, as there is no runtime penalty for executing monomorphised code as opposed to non-generic code. Monormorphisation is used by languages such as Rust to implement generics (*Rust Compiler Development Guide*, 2023). Monormorphisation was first formalised in *Featherweight Go* (Griesemer et al., 2020).

There are, however, certain limitations with this approach. Monormorphisation can lead to an explosion in the output code size, and not all programs can be monomorphised (Griesemer et al., 2020). Other approaches have been adopted by many languages that support parametric polymorphism. Java uses type erasure (Gosling et al., 2023), C# uses a hybrid of monomorphisation at runtime (as opposed to the traditional compile-time) and code sharing (Kennedy and Syme, 2001), and Go uses a hybrid of monormorphisation and dictionary passing (Scales and Randall, 2022).

Monormorphisation is appropriate for the restricted use of numerical type parameters in FGGA, since output code size explosion is not an issue — for every concrete type argument n, at most 1 output type will be produced for every type depending on the aforementioned parameter n (directly or indirectly).

This section presents a formalisation of monomorphisation from FGGA to Go, based on *Featherweight Go* (Griesemer et al., 2020). Because regular type parameters are now part of Go, they are output as is, and only numerical type parameters are "eliminated" in the monomorphisation process.

7.1 Formalisation

Δ is an environment mapping numerical type parameters α_n (i.e. those bound by **const**) to integer literals n.

The monomorphisation process consists of two stages — type collection and type translation. The type collection phase begins by collecting all the referenced named types in the main expression. The types are collected into the set ω, with one entry for every (type name, sequence of integers) pair. The sequence of integers in the pair is the named type's type argument list, with all non integer-value arguments removed.

After the initial collection, we apply the function G on ω, which collects further types from type declarations (*T-closure*) and methods (*M-closure* and *A-closure*) of the types in ω.

This process is repeated, applying G to the output of the previous G application until the fixed point of G is reached, i.e. when applying G produces an output that is the same as its input. This notion is captured in the I-Prog rule, where Ω is the fixed point.

Because only referenced types are collected, a side effect of monomorphisation is that any unused types are eliminated. This makes sense, because in general, if a generic type is never instantiated, we don't have any candidates it could monomorphise down to.

The second phase translates the main expression, and the collected types in Ω into valid Go code. The main task of this phase is to move all numerical type arguments from the

collected types into the names of the named types (rule M-Named), and remove any numerical type parameters or arguments from type parameter/argument lists while preserving any non-numerical type parameters/arguments (rules M-Named, M-α and M-Constraints). The remainder of the rules either recursively apply type translation to their components or are base cases that require no translation at all.

The formal rules describe the process in two distinct stages, whereas the implementation does both in a single stage (translating as it collects types). All the information needed to translate a term can be derived from the result of type collection on that term. E.g. given $\Delta = \emptyset$ the expression $e = Arr[2, \mathbf{int}]\{1, 2\}$, we know it produces $\omega = \{Arr[2]\}$ and translates to $e^\dagger = \langle Arr, 2\rangle[\mathbf{int}]\{1, 2\}$, where $\langle t, \overline{n}\rangle$ signifies the output type name. In an implementation, the name should be generated such that ideally it doesn't conflict with any other type name that the programmer might declare. Additionally, we assume the program has already been type-checked before being fed into the monomorphiser (the implemenation should type-check before monomorphising).

7.2 Monormorphisation properties

Any well-typed FGGA program P can be monomorphised into $P^\dagger$.

Property 7.1 (Totality) $P\ ok \implies P \mapsto P^\dagger$

Any well-typed FGGA program P is well-typed after monomorphisation, both in FGGA and in Go.

Property 7.2 (Soundess) $P\ ok \implies P^\dagger\ ok$

Any monomorphised program $P^\dagger$ has a one-to-one correspondence in behaviour (reduction equivalence) to the original program P (Griesemer et al., 2020). I.e. executing program P one step and then monomorphising the resulting program, is equivalent to monomorphising program P to $P^\dagger$ and executing $P^\dagger$ one step, modulo dead-type elimination. $P_0^\dagger$ reduces to $P_1^{☠}$ — denoting the program contains dead types, which we can eliminate by applying the monomorphisation procedure $P_1^{☠} \mapsto P_1^\dagger$. Figure 25 visualises this correspondence.

Property 7.3 (Bisimulation)

$$\frac{P_0 \mapsto P_0^\dagger \qquad P_0 \longrightarrow P_1 \qquad P_0^\dagger \longrightarrow P_1^{☠} \qquad P_1^{☠} \mapsto P_1^\dagger}{P_1 \mapsto P_1^\dagger}$$

$$\boxed{P \longrightarrow P'}$$

$$\frac{d \longrightarrow e}{\mathbf{package\ main};\ \overline{D}\ \mathbf{func\ main}()\ \{\_ = d\} \longrightarrow \mathbf{package\ main};\ \overline{D}\ \mathbf{func\ main}()\ \{\_ = e\}}$$

Type-instance sets $\boxed{\omega, \Omega}$

ω, Ω range over sets containing elements of the form $t[\overline{n}]$

Expressions $\boxed{\Delta \vdash e \blacktriangleright \omega}$

$$\text{I-INT-LITERAL}\ \frac{}{\Delta \vdash n \blacktriangleright \emptyset} \qquad \text{I-VAR}\ \frac{}{\Delta \vdash x \blacktriangleright \emptyset} \qquad \text{I-LITERAL}\ \frac{\Delta \vdash t_V[\overline{\tau}] \blacktriangleright \omega_\tau \qquad \Delta \vdash \overline{e \blacktriangleright \omega}}{\Delta \vdash t_V[\overline{\tau}]\{\overline{e}\} \blacktriangleright \overline{\omega_\tau} \cup \overline{\omega}} \qquad \text{I-FIELD}\ \frac{\Delta \vdash e \blacktriangleright \omega}{\Delta \vdash e.f \blacktriangleright \omega}$$

$$\text{I-INDEX}\ \frac{\Delta \vdash e \blacktriangleright \omega \qquad \Delta \vdash e' \blacktriangleright \omega'}{\Delta \vdash e[e'] \blacktriangleright \omega \cup \omega'} \qquad \text{I-CALL}\ \frac{\Delta \vdash e \blacktriangleright \omega \qquad \Delta \vdash \overline{e \blacktriangleright \omega}}{\Delta \vdash e.m(\overline{e}) \blacktriangleright \omega \cup \overline{\omega}} \qquad \text{I-ADD}\ \frac{\Delta \vdash e \blacktriangleright \omega \qquad \Delta \vdash e' \blacktriangleright \omega'}{\Delta \vdash e + e' \blacktriangleright \omega \cup \omega'}$$

Method specifications and type literals $\boxed{\Delta \vdash S \blacktriangleright \omega}$ $\boxed{\Delta \vdash T \blacktriangleright \omega}$

$$\text{I-SPECIFICATION}\ \frac{\Delta \vdash \overline{\tau \blacktriangleright \omega} \qquad \Delta \vdash \tau \blacktriangleright \omega}{\Delta \vdash m(\overline{x\ \tau})\ \tau \blacktriangleright \omega \cup \overline{\omega}} \qquad \text{I-STRUCT}\ \frac{\Delta \vdash \overline{\tau \blacktriangleright \omega}}{\Delta \vdash \textbf{struct}\ \{\overline{f\ \tau}\} \blacktriangleright \overline{\omega}} \qquad \text{I-ARRAY}\ \frac{\Delta \vdash \tau \blacktriangleright \omega}{\Delta \vdash [\tau_n]\tau \blacktriangleright \omega}$$

$$\text{I-INTERFACE}\ \frac{\Delta \vdash \overline{S \blacktriangleright \omega}}{\Delta \vdash \textbf{interface}\ \{\overline{S}\} \blacktriangleright \overline{\omega}}$$

Types $\boxed{\Delta \vdash \tau \blacktriangleright \omega}$

$$\text{I-INT}\ \frac{}{\Delta \vdash \textbf{int} \blacktriangleright \emptyset} \qquad \text{I-N}\ \frac{}{\Delta \vdash n \blacktriangleright \emptyset} \qquad \text{I-}\alpha\ \frac{}{\Delta \vdash \alpha \blacktriangleright \emptyset} \qquad \text{I-CONST}\ \frac{}{\Delta \vdash \textbf{const} \blacktriangleright \emptyset}$$

$$\text{I-NAMED}\ \frac{\overline{\tau \blacktriangleright \omega_\tau}}{\Delta \vdash t[\overline{\tau}] \blacktriangleright \{instance(t[\overline{\tau}][\![\Delta]\!])\} \cup \overline{\omega_\tau}}$$

Programs $\boxed{P \blacktriangleright \Omega}$

$$\text{I-PROG}\ \frac{\emptyset \vdash e \blacktriangleright \omega \qquad \Omega = \lim_{n \to \infty} G^n(\omega)}{\textbf{package main};\ \overline{D}\ \textbf{func main}()\ \{\_ = e\} \blacktriangleright \Omega}$$

Auxiliary functions

$$\frac{\overline{n} = \overline{n : n \in \overline{\tau}}}{\mathit{instance}(t[\overline{\tau}]) = t[\overline{n}]} \qquad \frac{\overline{\alpha_n} = \overline{\alpha : \alpha \ \mathbf{const} \in \overline{\Phi}}}{(\overline{\Phi} := \overline{n}) = (\overline{\alpha_n := n})}$$

$$G(\omega) = \mathit{T\text{-}closure}(\omega) \cup \mathit{M\text{-}closure}(\omega) \cup \mathit{A\text{-}closure}(\omega)$$

$$\mathit{T\text{-}closure}(\omega) = \bigcup \left\{ \overline{\omega'} \cup \omega'' \;\middle|\; \begin{array}{c} t[\overline{n}] \in \omega,\ (\mathbf{type}\ t[\overline{\Phi}]\ T) \in \overline{D},\ \overline{\Phi} = (\overline{\alpha\ \gamma}), \\ \overline{(\overline{\Phi} := \overline{n}) \vdash \gamma \blacktriangleright \omega'},\ (\overline{\Phi} := \overline{n}) \vdash T \blacktriangleright \omega'' \end{array} \right\}$$

$$\mathit{M\text{-}closure}(\omega) =$$

$$\bigcup \left\{ \omega' \cup \omega'' \;\middle|\; \begin{array}{c} t_V[\overline{n}] \in \omega,\ (\mathbf{func}\ (x\ t_V[\overline{\alpha}])\ mM\ \{\mathbf{return}\ e\}) \in \overline{D}, \\ \overline{\Phi} = \mathit{typeParams}(t_V),\ (\overline{\Phi} := \overline{n}) \vdash mM \blacktriangleright \omega',\ (\overline{\Phi} := \overline{n}) \vdash e \blacktriangleright \omega'' \end{array} \right\}$$

$$\mathit{A\text{-}closure}(\omega) =$$

$$\bigcup \left\{ \omega' \;\middle|\; \begin{array}{c} t_V[\overline{n}] \in \omega,\ \overline{\Phi} = \mathit{typeParams}(t_V),\ (\overline{\Phi} := \overline{n}) \vdash \tau \blacktriangleright \omega', \\ (\mathbf{func}\ (x\ t_V[\overline{\alpha}])\ m(x_1\ \mathbf{int},\ x_2\ \tau)\ t_V[\overline{\alpha}]\ \{x[x_1] = x_2;\ \mathbf{return}\ x\}) \in \overline{D} \end{array} \right\}$$

Figure 23: Type collection phase of FGGA to Go monormorphisation

Expressions $\boxed{\Delta \vdash e \mapsto e^\dagger}$

M-INT-LITERAL
$$\frac{}{\Delta \vdash n \mapsto n}$$

M-VAR
$$\frac{}{\Delta \vdash x \mapsto x}$$

M-LITERAL
$$\frac{\Delta \vdash \tau_V \mapsto \tau_V^\dagger \qquad \Delta \vdash \overline{e \mapsto e^\dagger}}{\Delta \vdash \tau_V\{\overline{e}\} \mapsto \tau_V^\dagger\{\overline{e^\dagger}\}}$$

M-FIELD
$$\frac{\Delta \vdash e \mapsto e^\dagger}{\Delta \vdash e.f \mapsto e^\dagger.f}$$

M-INDEX
$$\frac{\Delta \vdash e \mapsto e^\dagger \qquad \Delta \vdash e' \mapsto e'^\dagger}{\Delta \vdash e[e'] \mapsto e^\dagger[e'^\dagger]}$$

M-CALL
$$\frac{\Delta \vdash e \mapsto e^\dagger \qquad \Delta \vdash \overline{e \mapsto e^\dagger}}{\Delta \vdash e.m(\overline{e}) \mapsto e^\dagger.m(\overline{e^\dagger})}$$

M-ADD
$$\frac{\Delta \vdash e \mapsto e^\dagger \qquad \Delta \vdash e' \mapsto e'^\dagger}{\Delta \vdash e + e' \mapsto e^\dagger + e'^\dagger}$$

Method signatures and type literals $\boxed{\Delta \vdash M \mapsto M^\dagger}$ $\boxed{\Delta \vdash T \mapsto T^\dagger}$

M-SIGNATURE
$$\frac{\Delta \vdash \overline{\tau \mapsto \tau^\dagger} \qquad \Delta \vdash \tau \mapsto \tau^\dagger}{\Delta \vdash (\overline{x\ \tau})\ \tau \mapsto (\overline{x\ \tau^\dagger})\ \tau^\dagger}$$

M-STRUCT
$$\frac{\Delta \vdash \overline{\tau \mapsto \tau^\dagger}}{\Delta \vdash \textbf{struct}\ \{\overline{f\ \tau}\} \mapsto \textbf{struct}\ \{\overline{f\ \tau^\dagger}\}}$$

M-ARRAY
$$\frac{\Delta \vdash \tau_n \mapsto \tau_n^\dagger \qquad \Delta \vdash \tau \mapsto \tau^\dagger}{\Delta \vdash [\tau_n]\tau \mapsto [\tau_n^\dagger]\tau^\dagger}$$

M-INTERFACE
$$\frac{\Delta \vdash \overline{M \mapsto M^\dagger}}{\Delta \vdash \textbf{interface}\ \{\overline{mM}\} \mapsto \textbf{interface}\ \{\overline{mM^\dagger}\}}$$

Types $\boxed{\Delta \vdash \tau \mapsto \tau^\dagger}$

M-INT
$$\frac{}{\Delta \vdash \textbf{int} \mapsto \textbf{int}}$$

M-N
$$\frac{}{\Delta \vdash n \mapsto n}$$

M-α
$$\frac{}{\Delta \vdash \alpha \mapsto \alpha[\![\Delta]\!]}$$

M-NAMED
$$\frac{\Delta \vdash \overline{\tau \mapsto \tau^\dagger} \qquad \overline{n} = \overline{n : n \in \overline{\tau^\dagger}} \qquad \overline{\tau_c^\dagger} = \overline{\tau : (\tau \in \overline{\tau^\dagger}) \wedge (\tau \notin \overline{n})}}{\Delta \vdash t[\overline{\tau}] \mapsto \langle t, \overline{n} \rangle [\overline{\tau_c^\dagger}]}$$

Type parameter constraints $\boxed{\Delta \vdash \overline{\Phi} \mapsto \overline{\Phi}^\dagger}$

M-CONSTRAINTS
$$\frac{\Delta \vdash \overline{\gamma \mapsto \gamma^\dagger} \qquad \overline{\Phi}^\dagger = \overline{(\alpha\ \gamma^\dagger) : ((\alpha\ \gamma) \in \overline{\Phi}) \wedge (\gamma \neq \textbf{const})}}{\Delta \vdash \overline{\Phi} \mapsto \overline{\Phi}^\dagger}$$

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

M-PROGRAM

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

Auxiliary functions

$$\frac{\overline{\Phi} = (\overline{\alpha\ \gamma}) \qquad \overline{\alpha_c} = \overline{\alpha : (\alpha \in \overline{\alpha}) \wedge ((\alpha\ \textbf{const}) \notin \overline{\Phi})}}{\overline{\Phi} := \overline{\alpha_c}}$$

$$\mathcal{D}(\Omega) = \mathcal{D}_T(\Omega) \cup \mathcal{D}_M(\Omega) \cup \mathcal{D}_A(\Omega)$$

$$\mathcal{D}_T(\Omega) =$$

$$\left\{ \frac{t[\overline{n}] \in \Omega \qquad (\textbf{type}\ t[\overline{\Phi}]\ T) \in \overline{D} \qquad (\overline{\Phi} := \overline{n}) \vdash \overline{\Phi} \mapsto \overline{\Phi}^\dagger \qquad (\overline{\Phi} := \overline{n}) \vdash T \mapsto T^\dagger}{\textbf{type}\ \langle t, \overline{n} \rangle [\overline{\Phi}^\dagger]\ T^\dagger} \right\}$$

$$\mathcal{D}_M(\Omega) =$$

$$\left\{ \frac{\begin{array}{c} t_V[\overline{n}] \in \Omega \qquad (\textbf{func}\ (x\ t_V[\overline{\alpha}])\ mM\ \{\textbf{return}\ e\}) \in \overline{D} \\ \overline{\Phi} = \mathit{typeParams}(t_V) \qquad \overline{\Phi} := \overline{\alpha_c} \qquad (\overline{\Phi} := \overline{n}) \vdash M \mapsto M^\dagger \qquad (\overline{\Phi} := \overline{n}) \vdash e \mapsto e^\dagger \end{array}}{\textbf{func}(x\ \langle t_V, \overline{n} \rangle [\overline{\alpha_c}])\ mM^\dagger\ \{\textbf{return}\ e^\dagger\}} \right\}$$

$$\mathcal{D}_A(\Omega) =$$

$$\left\{ \frac{\begin{array}{c} t_V[\overline{n}] \in \Omega \qquad (\textbf{func}\ (x\ t_V[\overline{\alpha}])\ m(x_1\ \textbf{int},\ x_2\ \tau)\ t_V[\overline{\alpha}]\ \{x[x_1] = x_2;\ \textbf{return}\ x\}) \in \overline{D} \\ \overline{\Phi} = \mathit{typeParams}(t_V) \qquad \overline{\Phi} := \overline{\alpha_c} \qquad (\overline{\Phi} := \overline{n}) \vdash \tau \mapsto \tau^\dagger \end{array}}{\textbf{func}\ (x\ \langle t_V, \overline{n} \rangle [\overline{\alpha_c}])\ m(x_1\ \textbf{int}, x_2\ \tau^\dagger)\ \langle t_V, \overline{n} \rangle [\overline{\alpha_c}]\ \{x[x_1] = x_2;\ \textbf{return}\ x\}} \right\}$$

Figure 24: Translation phase of FGGA to Go monormorphisation

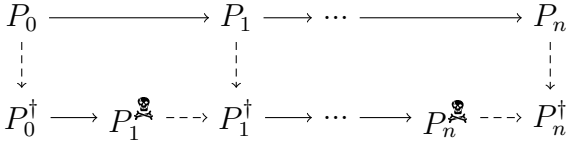

Figure 25: Bisimulation. A solid arrow indicates a program reduction step. A dashed arrow indicates monormorphisation of a program.

8 Implementation

As part of this work, two interpreters were implemented, one for FGA and one for FGGA. In addition, a monomorphiser was implemented translating FGGA to Go. The output of the monomorphiser is a valid subset of FGGA, so it can be interpreted by the FGGA interpreter as well as compiled by the official Go compiler.

One of the aims of building the interpreters was to test the formal syntax, reduction and typing rules defined in sections 5 and 6, and amend them when inconsistencies or issues arose. The aim of the monomorphiser was to prototype a translation from const-generic Go code to regular Go code, which is a useful first step to introducing this feature into the mainstream compiler. One could also extend the monomorphiser to support a proper superset of Go, which would allow developers to use generically sized arrays in their source code, and use the monomorphiser as the first step of the compilation process, the result of which could be fed directly into the Go compiler. The interpreters also support dynamic (run-time) checking of the progress and preservation properties as the input program is executed, and terminate upon reaching a state that was previously seen (i.e. some forms of infinite loops are detected at run-time).

A similar set of programs was found in the set of artifacts in Featherweight Go (Griesemer et al., 2020) — interpreters for the two languages formalised, and a monomorphiser translating generics (the feature introduced by Featherweight Go) into code that the official Go compiler at the time could handle.

8.1 Libraries and patterns

ANLTR[3] was used to generate the parser, and the interpreters and monomorphiser were implemented in Go itself.

The visitor pattern was heavily employed throughout the code for operations including building the abstract syntax tree (AST), preprocessing to remove ambiguity resulting from the grammar (as discussed in section 6.1), type checking, reduction, monomorphisation, and various other auxiliary operations. The pattern makes it easy to recursively traverse the AST and apply the desired operation.

8.2 Testing

The programs were written using test-driven development (TDD), where the test inputs for most components were small example programs, aimed at testing a single feature or case.

Since under TDD no feature may be implemented without an accompanying test, all developed features are covered by tests. TDD allows for fearless refactoring, since all features are fully covered by tests. Using example programs as test inputs allows for arbitrary refactoring of the implementation. During development, such a major refactoring was undertaken — namely moving the type parameter substitution (the need for which arising from the grammar ambiguity) from an ad hoc basis to a separate preprocessing step (i.e. a separate pass of the AST) before any type checking started.

[3]ANLTR: https://www.antlr.org/

Package	No. of tests	Coverage (%)
ast	—	99.6
parsetree	1	97.4
reduction	86	—
typecheck	109	98.6

Figure 26: Test statistics per package for FGA interpreter[5]

Package	No. of tests	Coverage (%)
ast	—	97.9
auxiliary	—	100.0
codegen	20	100.0
monomo	28	99.1
parsetree	2	97.6
preprocessor	1	87.8
reduction	97	96.0
typecheck	230	94.5

Figure 27: Test statistics per package for FGGA interpreter and FGGA to Go monomorphiser

A core part of the implementation was coming up with edge cases, writing tests to check how the interpreter behaved, and if necessary updating both the formal rules and implementation to cover the edge case. E.g in FGGA the *notReferenced* auxiliary had to be updated to instantiate the type literals it recursively checks, as a result of the described process.

In fact, following this process, two bugs in the official Go compiler (as of Go 1.22) were caught, causing the compiler to either crash due to stack overflow, or incorrectly type check a program (behaviour dependent on input). The two errors were subsequently reported as issues on Go's GitHub repository, where exact details can be found[4].

[4]issue 65711 and 65714

[5]Package test coverage was calculated using a modified cover command from the Go standard library. The modified code can be found at https://github.com/dawidl022/go/tree/package-coverage. The coverage is calculated based on all tests, not just ones found in the same package.

9 Conclusion

The main contributions this project achieved:

- Formalisation of arrays in a subset of Go (Featherweight Go with Arrays).
- Formalising an extension of FGA with regular and numerical type parameters that can be used to create arrays of generic sizes.
- Implementing interpreters for the above two languages to dynamically test their safety properties. The implementation underwent rigorous testing with hundreds of small example programs.
- Formalising a translation (monomorphisation) of FGGA to regular Go.
- Implementing the formalised monomorphiser.
- Submitting a formal language feature proposal and discussing the addition with the Go community, including the core Go team.
- Reporting confirmed bugs in actual Go compiler as a result of the rigorous testing.

9.1 Further work

- Formal proofs of progress and preservation properties, similar to the original *Featherweight Go* work (Griesemer et al., 2020).
- Address Go team's feedback on proposal (Lachowicz, 2024) and come to a consensus with the Go community about the future of the new feature. This may include investigating the original vision for generics in Go e.g. from the list of omissions in the accepted proposal (Taylor and Griesemer, 2021) and the older contracts proposal (Taylor and Griesemer, 2019).
- Go compiler implementation, which includes integrating the design proposed in this work with Go's "GCShape stenciling with Dictionaries" approach for generics (Scales and Randall, 2022).
- Formalise a more expressive technique for genericising over arrays, e.g. by using refinements types to specify the range of numerical type parameters accepted by a type.

References

Aronson, S., 2017. *Const generics* [Online]. Available from: `https://rust-lang.github.io/rfcs/2000-const-generics.html` [Accessed November 25, 2023].

Barendregt, H.P., 1981. *The lambda calculus: its syntax and semantics*. New York, N.Y.: Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co.

Bove, A. and Dybjer, P., 2009. Dependent types at work. In: *Language engineering and rigorous software development: international lernet alfa summer school 2008, piriapolis, uruguay, february 24 - march 1, 2008, revised tutorial lectures* [Online]. Ed. by A. Bove, L.S. Barbosa, A. Pardo, and J.S. Pinto. Berlin, Heidelberg: Springer Berlin Heidelberg, pp.57–99. Available from: `https://doi.org/10.1007/978-3-642-03153-3_2`.

Brady, E.C., 2011. Idris —: systems programming meets full dependent types. *Proceedings of the 5th acm workshop on programming languages meets program verification* [Online], PLPV '11. Austin, Texas, USA: Association for Computing Machinery, pp.43–54. Available from: `https://doi.org/10.1145/1929529.1929536`.

Cheney, D., 2013. *How to write benchmarks in Go* [Online]. Available from: `https://eli.thegreenplace.net/2023/common-pitfalls-in-go-benchmarking/` [Accessed January 19, 2024].

Gamboa, C., Canelas, P., Timperley, C., and Fonseca, A., 2023. Usability-oriented design of liquid types for java. *Proceedings of the 45th international conference on software engineering* [Online], ICSE '23. Melbourne, Victoria, Australia: IEEE Press, pp.1520–1532. Available from: `https://doi.org/10.1109/ICSE48619.2023.00132`.

Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., Smith, D., and Bierman, G., 2023. *The Java® language specification* [Online]. Oracle. Available from: `https://docs.oracle.com/javase/specs/jls/se21/html/index.html` [Accessed November 21, 2023].

Griesemer, R., Hu, R., Kokke, W., Lange, J., Taylor, I.L., Toninho, B., Wadler, P., and Yoshida, N., 2020. Featherweight Go. *Proc. ACM Program. Lang.* [Online], 4(OOPSLA). Available from: `https://doi.org/10.1145/3428217`.

Igarashi, A., Pierce, B., and Wadler, P., 1999. Featherweight Java: a minimal core calculus for Java and GJ. *SIGPLAN Not.* [Online], 34(10), pp.132–146. Available from: `https://doi.org/10.1145/320385.320395`.

ISO/IEC, 2018. *Programming languages — C*. (ISO/IEC 9899:2018). Geneva, Switzerland: ISO/IEC.

Jhala, R. and Vazou, N., 2021. Refinement types: a tutorial. *Foundations and trends® in programming languages* [Online], 6(3-4), pp.159–317. Available from: `https://doi.org/10.1561/2500000032`.

Johnston, P., 2017. *Creating a circular buffer in c and c++* [Online]. Available from: https://embeddedartistry.com/blog/2017/05/17/creating-a-circular-buffer-in-c-and-c/ [Accessed January 26, 2024].

Kennedy, A. and Syme, D., 2001. Design and implementation of generics for the .net common language runtime. *Sigplan not.* [Online], 36(5), pp.1–12. Available from: https://doi.org/10.1145/381694.378797.

Kulesza, T., 2020. *Go developer survey 2019 results* [Online]. The Go Blog. Available from: https://go.dev/blog/survey2019-results [Accessed November 24, 2023].

Lachowicz, D., 2024. *Proposal: go 2: const generics* [Online]. Available from: https://github.com/golang/go/issues/65555 [Accessed April 24, 2024].

Merrick, A., 2021. *Go developer survey 2020 results* [Online]. The Go Blog. Available from: https://go.dev/blog/survey2020-results [Accessed November 24, 2023].

Merrick, A., 2022. *Go developer survey 2021 results* [Online]. The Go Blog. Available from: https://go.dev/blog/survey2021-results [Accessed November 24, 2023].

Myers, A., 2009. *CS 6110 lecture 8: evaluation contexts, semantics by translation* [Online]. Cornell University. Available from: https://courses.cs.cornell.edu/cs6110/2009sp/lectures/lec08-sp09.pdf [Accessed November 27, 2023].

Norell, U., 2007. *Towards a practical programming language based on dependent type theory* [Online]. PhD thesis. SE-412 96 Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology. Available from: https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf.

Pierce, B.C., 2002. *Types and programming languages*. 1st ed. The MIT Press.

Pike, R., 2015. *Simplicity is complicated* [Online]. Available from: https://go.dev/talks/2015/simplicity-is-complicated.slide [Accessed November 25, 2023].

Rust compiler development guide, 2023 [Online]. Available from: https://rustc-dev-guide.rust-lang.org/ [Accessed November 25, 2023].

Scales, D. and Randall, K., 2022. *Go 1.18 implementation of generics via dictionaries and gcshape stenciling* [Online]. Google Open Source. Available from: https://go.googlesource.com/proposal/+/refs/heads/master/design/generics-implementation-dictionaries-go1.18.md [Accessed November 25, 2023].

Steele, G.L., 2017. It's time for a new old language. *Sigplan not.* [Online], 52(8), p.1. Available from: https://doi.org/10.1145/3155284.3018773.

Taylor, I.L. and Griesemer, R., 2019. *Contracts — draft design* [Online]. Google Open Source. Available from: https://go.googlesource.com/proposal/+/master/design/go2draft-contracts.md [Accessed April 24, 2024].

Taylor, I.L. and Griesemer, R., 2021. *Type parameters proposal* [Online]. Google Open Source. Available from: https://go.googlesource.com/proposal/+/HEAD/design/43651-type-parameters.md [Accessed November 16, 2023].

The const generics project group, 2021. *Const generics MVP hits beta!* [Online]. Rust Blog. Available from: https://blog.rust-lang.org/2021/02/26/const-generics-mvp-beta.html [Accessed November 12, 2023].

The Go programming language specification, 2021 [Online]. Google Open Source. Available from: https://go.dev/doc/go1.17_spec [Accessed January 20, 2024].

The Go programming language specification, 2023 [Online]. Google Open Source. Available from: https://go.dev/ref/spec [Accessed November 16, 2023].

The Rust reference, 2020 [Online]. Available from: https://doc.rust-lang.org/stable/reference/ [Accessed November 21, 2023].

Tsai, J., 2020. *reflect.DeepEqual on two empty slices returns false* [Online]. Available from: https://github.com/golang/go/issues/42265#issuecomment-718304456 [Accessed January 26, 2024].

Vazou, N., 2015. *A gentle introduction to liquid types* [Online]. University of California San Diego. Available from: https://goto.ucsd.edu/~ucsdpl-blog/liquidtypes/2015/09/19/liquid-types/ [Accessed April 20, 2024].

Wagner, A., 2021. *Proposal: go 2: spec: generic parameterization of array sizes* [Online]. Google Open Source. Available from: https://github.com/golang/go/issues/44253#issuecomment-821047754 [Accessed January 23, 2024].

Wagner, B., 2023. *C# reference: arrays* [Online]. Microsoft. Available from: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/arrays [Accessed November 23, 2023].

Werner, A., 2021. *Proposal: generic parameterization of array sizes* [Online]. Google Open Source. Available from: https://go.googlesource.com/proposal/+/HEAD/design/44253-generic-array-sizes.md [Accessed November 26, 2023].

Xi, H. and Pfenning, F., 1999. Dependent types in practical programming. *Proceedings of the 26th acm sigplan-sigact symposium on principles of programming languages* [Online], POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, pp.214–227. Available from: https://doi.org/10.1145/292540.292560.

A Risk Assessment

Risk	Impact	Likelihood Rating	Impact Rating	Preventive Actions
Being occupied with other activities during semester	Not spending enough time on project, leading to unfinished/poor quality project	High	High	Start project ahead of the start of semester
Flaw in design	Design and all implementations need to be fixed	Low-Medium	Medium	Regularly discuss design choices with supervisor and submit proposal to Go community

Table 1: Risk Register

B Project Plan

Minor changes were made since the initial project plan. Most notably, select arithmetic operators are to become part of the scope of the formal rules and interpreter, since they have the potential to play an important role in compelling code examples, showcasing generic array sizes. It is in the nature of arrays for there to be a way to loop over them, and operators are primitives that can allow for that.

The less significant change was a slight offset in the schedule of the tasks. This is not a concern since the project timeline had a large margin to begin with, with optional "extension" tasks (not required for the submission of the project) making up the final third of the timeline.

Each horizontal section in the chart denotes a milestone, comprised of one or more tasks. Grey rectangles indicate tasks already completed at the time of updating the project plan.

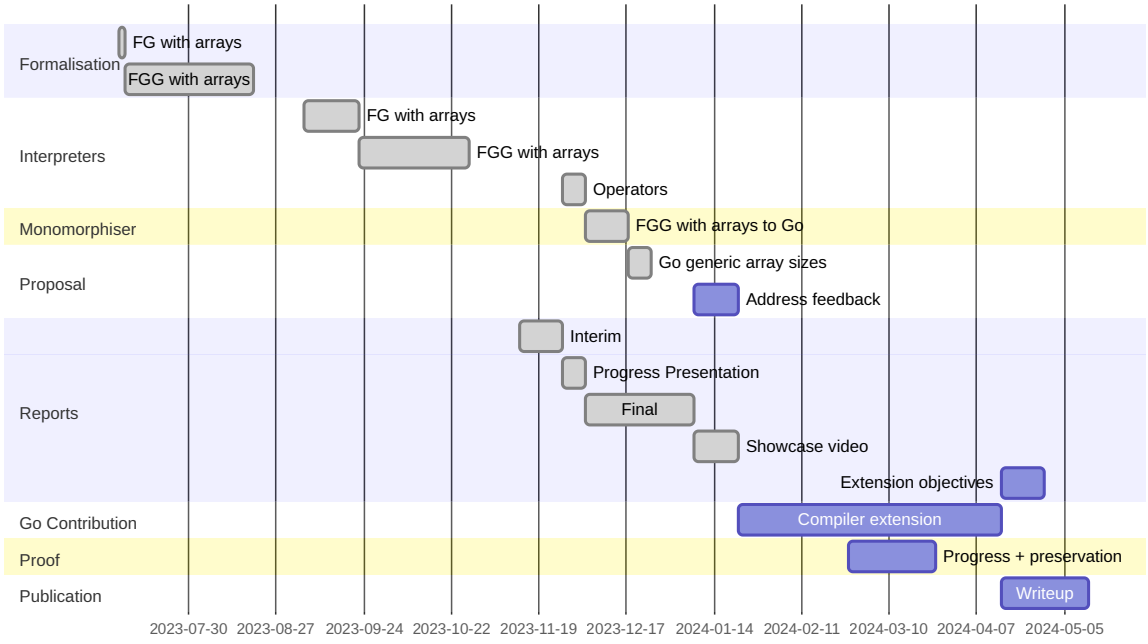

Figure 28: Timeplan of tasks, milestones and deliverables

C Description of array semantics

C.1 Value type

Arrays offer semantic differences, that may make it more appealing to build data structures from for certain use cases. We've seen that arrays are value types, and as such, copies can be easily made through simple assignment to a variable or passing into a function. While cloning a single slice is not much more difficult, if we have a large, nested data structure, that is slice-based, performing a deep-copy is a verbose and manual process (there is no standard library deep-copy function as of Go 1.21). If instead, the data structure was array-based, with no internal pointers (only potentially a top-level pointer to the data structure), performing a deep copy becomes trivial.

C.2 Comparison

Arrays are comparable, while slices are not. Again, for a simple example, one could define a function that compares the size and elements of slices to determine if two slices are equal in value (not reference). It is slightly more difficult to do the same with a nested slice-based data structure (although this time around, the standard library does provide `reflect.DeepEqual` with some caveats[6]), a nested array-based data structure (with no internal pointers) can be simply compared with the `==` operator.

This comparable attribute of arrays becomes extremely important when we wish to use a collection of elements as a key in a hash map. Only comparable data structures can be used as map keys, which means that arrays can be used as map keys, while slices cannot. This also applies recursively, so slice-based data structures cannot be used as map keys, while array-based ones can.

[6] `reflect.DeepEqual` differentiates between a nil and an empty slice, despite them semantically being equivalent. `cmp.Equal` with the `cmpopts.EquateEmpty` option can be used instead to equate semantically empty slices (Tsai, 2020).

D Code examples

D.1 Generic programming in C using macros

```
#include <stdio.h>

#define ARRAY(TYPE, SIZE, NAME) typedef struct { \
    TYPE x[SIZE]; \
} NAME; \
\
TYPE NAME##_first(NAME s) { \
    return s.x[0]; \
}


ARRAY(int, 5, Foo)

ARRAY(char*, 2, Bar)

int main(void) {
    Foo f = { 1, 2, 3, 4, 5 };
    printf("%d\n", Foo_first(f));

    Bar b = { "hello", "world" };
    printf("%s\n", Bar_first(b));
}
```

D.2 Full FGA implementation of resizeable arrays

In this particular example, the capacity of the resizeable array is hard-coded to 5. This can of course be any value, but must be hard-coded without generically sized arrays. Because FGA (unlike FGGA) does not support generics at all, some structs are monomorphised "by hand" to work with all the types required for the example, e.g. `Func` and `FuncA`.

One may notice that this implementation is much more verbose than the idiomatic Go example. This is due to a design decision to keep FGA simple and not introduce "unnecessary" constructs such as boolean and subtraction (and therefore negative integers). Church-like encoding is used for booleans and conditional logic (Barendregt, 1981), whereas natural numbers (that can be incremented and decremented for the length of the `Array`) use `succ`, `pred` and `isZero` similarly to how they're found in B.C. Pierce (2002)'s calculus of booleans and numbers. Functions have been encoded in a similar way as in *Featherweight Go* (Griesemer et al., 2020) — structs are created to hold arguments, which implement the `Func` interface with a single `call` method that takes no arguments. This way we can create arbitrary closures (callbacks) for conditional selection.

```
type Array struct {
    arr Arr
    len Nat
}

func (a Array) Push(el int) Array {
    return a.Cap().ifLessEqA(a.len,
        ArrayFunc{a},
        PushFunc{a, el})
}
func (f PushFunc) call() Array {
    return Array{
        f.a.arr.set(
            f.a.len.val(), f.el),
        Succ{f.a.len}}
}

func (a Array) Pop() Array {
    return Array{a.arr, a.len.pred()}
}

func (a Array) Get(i Nat) int {
    return a.len.ifLessEq(i,
        IntFunc{0},
        ArrGetFunc{a.arr, i.val()})
}
func (f ArrGetFunc) call() int {
    return f.arr[f.i]
}

func (a Array) Len() Nat {
    return a.len
}

func (a Array) Cap() Nat {
    return Succ{Succ{Succ{Succ{Succ{Zero{}}}}}}
}

type EmptyArrayFunc struct {
}

func (e EmptyArrayFunc) call() Array {
    return Array{Arr{0, 0, 0, 0, 0}, Zero{}}
}
```

```
type Arr [5]int

func (a Arr) set(i int, val int) Arr {
    a[i] = val;
    return a
}

type Nat interface {
    val() int
    pred() Nat
    ifLessEq(other Nat, ifTrue Func, ifFalse Func) int
    ifLessEqA(other Nat, ifTrue FuncA, ifFalse FuncA) Array
    isZero() Bool
    isZeroA() BoolA
}

type Bool interface {
    eval(ifTrue Func, ifFalse Func) int
}

type BoolA interface {
    eval(ifTrue FuncA, ifFalse FuncA) Array
}

type Func interface {
    call() int
}

type FuncA interface {
    call() Array
}

type True struct {
}

type TrueA struct {
}

func (t True) eval(ifTrue Func, ifFalse Func) int {
    return ifTrue.call()
}

func (t TrueA) eval(ifTrue FuncA, ifFalse FuncA) Array {
    return ifTrue.call()
}
```

```
type False struct {
}

type FalseA struct {
}

func (f False) eval(ifTrue Func, ifFalse Func) int {
    return ifFalse.call()
}

func (f FalseA) eval(ifTrue FuncA, ifFalse FuncA) Array {
    return ifFalse.call()
}

type Zero struct{}

func (z Zero) val() int {
    return 0
}

func (z Zero) pred() Nat {
    return z
}

func (z Zero) isZero() Bool {
    return True{}
}

func (z Zero) isZeroA() BoolA {
    return TrueA{}
}

func (z Zero) ifLessEq(other Nat, ifTrue Func, ifFalse Func) int {
    return ifTrue.call()
}

func (z Zero) ifLessEqA(other Nat, ifTrue FuncA, ifFalse FuncA) Array {
    return ifTrue.call()
}

type Succ struct {
    predF Nat
}
```

```
func (s Succ) val() int {
    return s.predF.val() + 1
}

func (s Succ) pred() Nat {
    return s.predF
}

func (s Succ) isZero() Bool {
    return False{}
}

func (s Succ) isZeroA() BoolA {
    return FalseA{}
}

func (s Succ) ifLessEq(other Nat, ifTrue Func, ifFalse Func) int {
    return other.isZero().eval(
        ifFalse, IfLessEq{s.pred(), other.pred(), ifTrue, ifFalse})
}

func (s Succ) ifLessEqA(other Nat, ifTrue FuncA, ifFalse FuncA) Array {
    return other.isZeroA().eval(
        ifFalse, IfLessEqR{s.pred(), other.pred(), ifTrue, ifFalse})
}

type IfLessEq struct {
    a       Nat
    b       Nat
    ifTrue  Func
    ifFalse Func
}

type IfLessEqR struct {
    a       Nat
    b       Nat
    ifTrue  FuncA
    ifFalse FuncA
}

func (i IfLessEq) call() int {
    return i.a.ifLessEq(i.b, i.ifTrue, i.ifFalse)
}

func (i IfLessEqR) call() Array {
```

```
    return i.a.ifLessEqA(i.b, i.ifTrue, i.ifFalse)
}

type ArrayFunc struct {
    a Array
}

func (r ArrayFunc) call() Array {
    return r.a
}

type PushFunc struct {
    a  Array
    el int
}

type ArrGetFunc struct {
    arr Arr
    i   int
}

type IntFunc struct {
    i int
}

func (i IntFunc) call() int {
    return i.i
}
```

D.3 Full FGA implementation of dequeues

```
type Deque struct {
    arr   Arr
    front Nat
    back  Nat
}

func (d Deque) PushFront(el int) Deque {
    return d.succ(d.front).ifEqD(d.back, DequeFunc{d}, PushFrontFunc{d, el})
}
func (f PushFrontFunc) call() Deque {
    return Deque{
        f.d.arr.set(f.d.front.val(), f.el),
        f.d.succ(f.d.front), f.d.back}
```

```
}

func (d Deque) PopFront() Deque {
    return d.front.ifEqD(d.back, DequeFunc{d}, PopFrontFunc{d})
}
func (f PopFrontFunc) call() Deque {
    return Deque{f.d.arr, f.d.pred(f.d.front), f.d.back}
}

func (d Deque) GetFront() int {
    return d.front.ifEq(d.back,
        IntFunc{0}, ArrGetFunc{d.arr, d.pred(d.front).val()})
}

func (d Deque) PushBack(el int) Deque {
    return d.front.ifEqD(d.pred(d.back), DequeFunc{d}, PushBackFunc{d, el})
}
func (f PushBackFunc) call() Deque {
    return Deque{
        f.d.arr.set(f.d.pred(f.d.back).val(), f.el),
        f.d.front, f.d.pred(f.d.back)}
}

func (d Deque) PopBack() Deque {
    return d.front.ifEqD(d.back, DequeFunc{d}, PopBackFunc{d})
}
func (f PopBackFunc) call() Deque {
    return Deque{f.d.arr, f.d.front, f.d.succ(f.d.back)}
}

func (d Deque) GetBack() int {
    return d.front.ifEq(d.back, IntFunc{0}, ArrGetFunc{d.arr, d.back.val()})
}

func (d Deque) pred(n Nat) Nat {
    return n.ifEqN(Zero{}, CapFunc{d}, PredFunc{n})
}

func (d Deque) succ(n Nat) Nat {
    return n.ifEqN(d.Cap(), ZeroFunc{}, SuccFunc{n})
}

func (d Deque) Cap() Nat {
    return Succ{Succ{Succ{Succ{Succ{Zero{}}}}}}
}
```

```
type EmptyDequeFunc struct {
}

func (e EmptyDequeFunc) call() Deque {
    return Deque{Arr{0, 0, 0, 0, 0, 0}, Zero{}, Zero{}}
}

type CapFunc struct {
    d Deque
}

func (f CapFunc) call() Nat {
    return f.d.Cap()
}

type PredFunc struct {
    n Nat
}

func (f PredFunc) call() Nat {
    return f.n.pred()
}

type ZeroFunc struct {
}

func (z ZeroFunc) call() Nat {
    return Zero{}
}

type SuccFunc struct {
    n Nat
}

func (f SuccFunc) call() Nat {
    return Succ{f.n}
}

type Arr [6]int

func (a Arr) set(i int, val int) Arr {
    a[i] = val;
    return a
}
```

```
type Nat interface {
    val() int
    pred() Nat
    ifEq(other Nat, ifTrue Func, ifFalse Func) int
    ifEqN(other Nat, ifTrue FuncN, ifFalse FuncN) Nat
    ifEqD(other Nat, ifTrue FuncD, ifFalse FuncD) Deque
    isZero() Bool
    isZeroN() BoolN
    isZeroD() BoolD
}

type Bool interface {
    eval(ifTrue Func, ifFalse Func) int
}

type BoolN interface {
    eval(ifTrue FuncN, ifFalse FuncN) Nat
}

type BoolD interface {
    eval(ifTrue FuncD, ifFalse FuncD) Deque
}

type Func interface {
    call() int
}

type FuncN interface {
    call() Nat
}

type FuncD interface {
    call() Deque
}

type True struct {
}

type TrueN struct {
}

type TrueD struct {
}
```

```
func (t True) eval(ifTrue Func, ifFalse Func) int {
    return ifTrue.call()
}

func (t TrueN) eval(ifTrue FuncN, ifFalse FuncN) Nat {
    return ifTrue.call()
}

func (t TrueD) eval(ifTrue FuncD, ifFalse FuncD) Deque {
    return ifTrue.call()
}

type False struct {
}

type FalseN struct{}

type FalseD struct {
}

func (f False) eval(ifTrue Func, ifFalse Func) int {
    return ifFalse.call()
}

func (f FalseN) eval(ifTrue FuncN, ifFalse FuncN) Nat {
    return ifFalse.call()
}

func (f FalseD) eval(ifTrue FuncD, ifFalse FuncD) Deque {
    return ifFalse.call()
}

type Zero struct{}

func (z Zero) val() int {
    return 0
}

func (z Zero) pred() Nat {
    return Zero{}
}

func (z Zero) ifEq(other Nat, ifTrue Func, ifFalse Func) int {
    return other.isZero().eval(ifTrue, ifFalse)
}
```

```
func (z Zero) ifEqN(other Nat, ifTrue FuncN, ifFalse FuncN) Nat {
    return other.isZeroN().eval(ifTrue, ifFalse)
}

func (z Zero) ifEqD(other Nat, ifTrue FuncD, ifFalse FuncD) Deque {
    return other.isZeroD().eval(ifTrue, ifFalse)
}

func (z Zero) isZero() Bool {
    return True{}
}

func (z Zero) isZeroN() BoolN {
    return TrueN{}
}

func (z Zero) isZeroD() BoolD {
    return TrueD{}
}

type Succ struct {
    predF Nat
}

func (s Succ) val() int {
    return s.predF.val() + 1
}

func (s Succ) pred() Nat {
    return s.predF
}

func (s Succ) ifEq(other Nat, ifTrue Func, ifFalse Func) int {
    return other.isZero().eval(ifFalse, IfEq{s.pred(), other.pred(), ifTrue, ifFalse})
}

func (s Succ) ifEqN(other Nat, ifTrue FuncN, ifFalse FuncN) Nat {
    return other.isZeroN().eval(ifFalse, IfEqN{s.pred(), other.pred(), ifTrue, ifFalse})
}

func (s Succ) ifEqD(other Nat, ifTrue FuncD, ifFalse FuncD) Deque {
    return other.isZeroD().eval(ifFalse, IfEqD{s.pred(), other.pred(), ifTrue, ifFalse})
}
```

```
type IfEq struct {
    a       Nat
    b       Nat
    ifTrue  Func
    ifFalse Func
}

type IfEqN struct {
    a       Nat
    b       Nat
    ifTrue  FuncN
    ifFalse FuncN
}

type IfEqD struct {
    a       Nat
    b       Nat
    ifTrue  FuncD
    ifFalse FuncD
}

func (i IfEq) call() int {
    return i.a.ifEq(i.b, i.ifTrue, i.ifFalse)
}

func (i IfEqN) call() Nat {
    return i.a.ifEqN(i.b, i.ifTrue, i.ifFalse)
}

func (i IfEqD) call() Deque {
    return i.a.ifEqD(i.b, i.ifTrue, i.ifFalse)
}

func (s Succ) isZero() Bool {
    return False{}
}

func (s Succ) isZeroN() BoolN {
    return FalseN{}
}

func (s Succ) isZeroD() BoolD {
    return FalseD{}
}
```

```
type DequeFunc struct {
    d Deque
}

func (f DequeFunc) call() Deque {
    return f.d
}

type PushFrontFunc struct {
    d  Deque
    el int
}

type PopFrontFunc struct {
    d Deque
}

type IntFunc struct {
    i int
}

func (f IntFunc) call() int {
    return f.i
}

type ArrGetFunc struct {
    arr Arr
    i   int
}

func (f ArrGetFunc) call() int {
    return f.arr[f.i]
}

type PushBackFunc struct {
    d  Deque
    el int
}

type PopBackFunc struct {
    d Deque
}
```

E Proposal addendum

E.1 More complex const bounds

What about when two distinct type parameters appear in a "constant" expression, such as in the (somewhat contrived) example below:

```
func difference[T any, N const, M const](a [N]T, b [M]T) [N - M]T {
    // some presumably useful code...
}
```

This code is only safe to execute when $N \geq M$. This adds additional complexity, as we need to extend our approach from before if we wish to pursue the liberal approach. In the implicit model, the compiler would have to reject all instantiations of the function where $N < M$. In the explicit model, the slicing notation would be allowed to accept constant expressions, and since type parameters are constant expressions, they could be used to explicitly constrain the type parameters. Go already permits referencing other type parameters, including the one that is being constrained, in the constraint of a type parameter. The compiler would still need to verify that the bounds specified by the programmer make the operation legal for all instantiations.

```
func difference[T any, N const[M:], M const](a [N]T, b [M]T) [N - M]T {
    // some presumably useful code...
}
```

Another area of concern that may arise with the liberal approach is recursively defined functions or types, where the numerical type parameter differs in each recursive instantiation (as shown in figure 29). Attempting to monomorphise such code would lead to extreme code bloat, and should not be allowed. In both the implicit and explicit models, this is a non-issue, as without knowing how the function or type will behave at runtime, there is no range of values `N` that would be guaranteed to not cause an underflow/overflow, so such a function cannot be defined. E.g. if in `newArrStrange` we give `N` an explicit upper bound of x, than the compiler would complain, since within the body of the function `newArrStrange` could be recursively called with a type argument of $x + 1$, which exceeds the type parameter's upper bound.

```
func newArrStrange[T any, N const, M const](n int) [N + M]T {
    if n == 0 {
        return [N + M]T{}
    }
    return newArrStrange[T, N + 1, M - 1]()
}
```

Figure 29: Contrived recursively defined construction of array

```
func expressions[T any, N const](arr [N]T) {
    const _ = len(arr) // does not compile
          _ = len(arr) // compiles - non-constant int type

    const _ = N        // does not compile
          _ = N        // compiles - non-constant int type
}
```

Figure 30: Expressions derived from a numerical type parameter are non-constant

E.2 Slicing generic arrays

Since in the conservative model, we have no guarantees about the array bounds, the only permitted slicing operation is `[:]`, which creates a slice of the entire array. Just as with the index operation, if the programmer wishes to move the bounds check to runtime, they can simply reslice the slice obtained from `[:]`. It is worth noting, that as of Go 1.21, slicing a generic variable that is constrained by a union of array types is forbidden (even `[:]` is not allowed). The liberal approach could be used to set lower bounds on array lengths to allow compile-time safe generic array slicing, other than just `[:]`.

E.3 The `len` function

Go's built-in `len` function is special in the sense that depending on the context, it may or may not be computed at compile time (*The Go Programming Language Specification*, 2023). If it is computed at compile time, we can assign the result to a **const** variable, or use it in any other place that requires a compile-time constant non-negative integer value, such as for the length of an array.

An example of a compile-time evaluation of `len` is when it is applied to an array value literal, and an example of a run-time evaluation of `len` is when it is applied to a slice value. The question arises, how should `len` treat generic arrays (specifically, ones that are parameterised on size)?

If we take Rust's conservative approach of prohibiting the use of expressions that include constant type parameters (except when the expression is a lone type parameter) as type arguments (The const generics project group, 2021), which include applying `len` to a generically sized array, then it becomes clear why `len` of a generically sized array should yield a non-constant integer (as indeed is the case in Rust). Without this restriction, the compiler would have to keep track of which constant was derived from an expression containing a type-parameter, to prevent the later usage of such a constant in another **const** expression used as a type argument. The same logic applies to the numerical type parameters themselves: if they could be assigned to constant variables, then the constant variable could be used as part of a constant expression used to instantiate a **const** type parameter, leading to the problems discussed in the previous section. And so, when **const** type parameters are used as values, the resulting expression type should be a non-constant **int** under the conservative model.

```
type array interface {
    [2]int | [3]int
}

func foo[A array](a A) {
    // compile error: len(a) (value of type int) is not constant
    const n = len(a)
}
```

Figure 31: As of Go 1.18, `len` of an array union type set interface value is non-constant

If we view the generally sized arrays as the union of array types of all sizes (similar to the explicitly enumerated array type union we can already represent in code), then we can keep things as they are currently in Go. I.e. a generic type bound by a union of arrays has a non-constant `len`.

This is likely to avoid constraining the compiler implementation. With a full monomorphisation approach, the **const** of the example above is not an issue, as each instantiation of the function has its own local `const n` with a distinct value. However, in the current GC Shape Stenciling approach used by the Go compiler as of Go 1.18 (Scales and Randall, 2022), it is possible to construct a union of array types that have the same GC shape, yet the length of the array type differs. In such cases, **const**s within a generic function would be difficult or even impossible to handle correctly. A simpler explanation is that generic types simply always have non-constant lengths (given `len` is defined on all types of the type set), regardless of whether or not values of their constraint type could yield a constant length (*The Go Programming Language Specification*, 2023).

F Featherweight Go with Arrays addendum

F.1 FGA Syntax: Expressions

Expressions may take on a number of forms. The most basic expressions are integer literals (e.g. 0, 1, 2, 10, -1 etc.). Variables, which in the context of FGA happen to be parameter names, are also basic expressions.

Complex expressions may involve 0 or more subexpressions. The first kind are method calls, which are recursively defined as an expression, followed by a dot, followed by a method name (an identifier), followed by a sequence of expressions enclosed in parentheses. Another recursively defined kind of expression is a value literal, which consists of a value type name, followed by a sequence of expressions enclosed in curly brackets. Value literals are used to instantiate structs or arrays. The select expression consists of an expression, followed by a dot and the field name (an identifier), which is used to select a field from a struct. An array index consists of an expression, followed by another expression enclosed in square brackets. The array index expression is used to retrieve an element of an array (the first subexpression) at a specified index (the second subexpression).

F.2 FGA Reduction

The auxiliary function *fields* looks up the struct type name given as an argument in the sequence of declarations, and returns the sequence of fields in the definition of the struct type. The rule R-Field says that a select expression on a struct literal *value* evaluates to the field value corresponding to the same position in the struct literal as the field name is in the struct type declaration.

The auxiliary function *indexBounds* returns a set of valid index indices of an array type, i.e. the set of indices that are within the bounds of the array. The rule R-Index says an array index expression on an array literal *value* with an integer literal n as the index reduces to the element of the array at index n, if and only if n is within bounds of the array.

The auxiliary function *body* looks up the method declaration given by the type name and method name, and returns the expression in the body of that method with a template for the receiver and parameters. The rule R-Call says that a method call expression where the receiver is a value and the arguments are also values, reduces to the *body* of the method defined on the *type* of the receiver, with the actual parameters (receiver and arguments) substituted for the formal parameters in the *body* template.

Apart from the computation rules described above, there is also a congruence rule defined in terms of evaluation contexts, which says that if d evaluates to e, then d in the context of E evaluates to e in the same context E. The evaluation context defines the order of evaluation (Myers, 2009), when there are multiple subexpressions in a single expression, and where at least one of the subexpressions is not a value.

By the evaluation context rules, a method call must first reduce its receiver to a value, and subsequently reduce its arguments to values, one by one. A value literal has its elements reduced to values, one by one. A select expression must first reduce its receiver to a value. An array index must first reduce its receiver to an array literal value, and then its index expression to a value.

F.3 FGA Typing

A method specification is considered well-typed if the parameter names are distinct (i.e. each parameter name is different) and the parameter and return types are themselves well-typed (rule T-Specification).

A struct type literal is well-formed if all of its fields are distinct and the field types are themselves well-formed (rule T-Struct). An interface type literal is well-formed if all of its method specifications are unique (i.e. each method specified in the interface has a different name) and well-formed (rule T-Interface). An array type literal is well-formed if the integer literal defining the size of the array is greater than or equal to zero, and the array element type is well-formed (rule T-Array).

A type declaration is well-formed if its type literal is well-formed (rule T-Type). A method declaration is well-formed if the parameters (including the receiver) are distinct, the receiver, parameter and return types are well-formed and the method expression's type is a subtype of the return type under the environment formed from the method parameters (and receiver) (rule T-Func).

The rule T-Var formally defines what it means for a variable x to be of type t under the environment Γ, i.e. when the pair $x : t$ is in Γ. A method call expression is well-formed and of the type u being the method specification's return type, when the receiver expression is well-typed and has a method named m in the method set of its type. In addition, all the argument expressions must be well-typed, and be subtypes of the formal parameter types (rule T-Call). A struct literal expression is well-formed and of the struct type that was instantiated, if the struct type is well-formed, and each element's type of the struct literal is a subtype of the corresponding field types in the struct type declaration (rule T-Struct-Literal). A field select expression is well-typed if the receiver is well-typed and of struct type. The expression is of the same type as the field corresponding to the selected field name in the struct type declaration (rule T-Field).

Finally, a program is well-typed when all the type declaration names are distinct (and do not coincide with the predeclared **int** type name). The auxiliary function *tdecls* returns all type names declared in the program. The method declarations must also be distinct, in the sense that no two methods declared on the same type may have the same name. The auxiliary function *mdecls* returns all pairs of $t_V.m$ (receiver type + method name) declared in the program. All declarations along with the main expression must also be well-formed (rule T-Prog).

G Featherweight Generic Go with Arrays addendum

G.1 FGGA Typing

The $methods_\Delta$ auxiliary is now a partial function, not defined on the **const** type. This is because there is currently never a context in which the $methods_\Delta$ would need to be applied to **const**. In a future extension, type parameters bounded by **const** could be used as expressions (since they will be instantiated with integer values in user programs), in which case we could define $methods_\Delta(\textbf{const}) = \{\}$ so that they implement the empty interface. We could then additionally have a rule stating that type parameters bounded by **const** implement **int**.

Because $methods_\Delta$ is not defined on **const**, there is an additional subtyping rule that specifies that type parameters which are bound by **const**, are also subtypes of **const** (rule $<:_{const-\text{Param}}$). All type parameters are subtypes of their bounds, but where the bounds are a regular interface (i.e. not **const**), this can already be derived via the $<:_I$ rule.

The remainder of the T-Type rule checks that the type declaration's type parameters constraints $\overline{\Phi}$ are well-formed (via T-Formal), and that the type literal is well-typed in the typing environment formed from $\overline{\Phi}$.

The rule T-Const asserts that **const** type passes the bounds type check in the T-Formal rule in any environment Δ. The T-Specification rule has been updated to type check the parameter and return types under the typing environment $\overline{\Phi}$. Additionally, neither the parameter nor return types may be **const** types. T-Struct has similarly been updated to check the field types in the environment $\overline{\Phi}$, and to only permit non-**const** field types.

The T-Func and T-Func-Arrayset rules have been updated to look up the type parameter constraints based on the receiver type parameters to construct typing environments for the updated type-checking rules. Rules T-Int-Literal, T-Var, T-Call, T-Struct-Literal, T-Field and T-Program are all updated to use types τ and typing environments Δ. The supertype of an array literal's elements is determined by type parameter substitution on the array's element type.

As before, there are two rules for performing an array index operation, one where the type of the index is **int**, and one where the type is an integer literal (rules T-Array-Index and T-Array-Index-Literal).

G.1.1 Not Referenced predicate

The recursive case of *notReferenced* exists for named types $t[\overline{\tau}]$, and the base cases are the remaining type kinds; integer literals cannot be used as a type name, and neither can the keyword **const**, so there is no possibility of a self-reference for those types. If a type is named the same as a type parameter, the type parameter will shadow the type name, and no self-reference occurs. When checking for self-reference in a named type, first we check whether the named-type's type name t is equal to any of the types we have already seen $\overline{t_r}$. Initially, the seen types sequence only contains the type we are type checking in T-Type. *notReferenced* is performed recursively on all type arguments, as those could also be referencing one of the types from $\overline{t_r}$. To detect indirect self-references (i.e. circular references), *notReferenced* is applied on all type parameter bounds of the named type's declaration, appending the type name t to the sequence of seen type names $\overline{t_r}$ to check for self-references. While theoretically,

we could have just a single type t_r to check for at a time, rather than a sequence, this would lead to problems when checking types that are self-referential, but do not contain the initial type t_r in the cycle, as the rule would recurse indefinitely.

H Formal derivation examples

H.1 Featherweight Go with Arrays reduction

Below are derivation trees reducing an example FGA expression down to a value. Each reduction step in the example program has its own derivation tree.

$D_0 = \textbf{type}\ \text{any}\ \textbf{interface}\{\}$
$D_1 = \textbf{type}\ \text{AnyArray2}\ [2]\,\text{any}$
$D_2 = \textbf{func}(\text{this}\ \ \text{AnyArray2})\ \text{First}()\ \ \text{any}\ \ \{\textbf{return}\ \ \text{this}[0]\}$
$D_3 = \textbf{func}(\text{this}\ \ \text{AnyArray2})\ \ \text{Set}(\text{i}\ \ \textbf{int}, \text{v}\ \ \ \text{any})\ \ \text{AnyArray2}\ \ \{\,\text{this}[\text{i}] = \text{v};\ \ \textbf{return}\ \ \text{this}\,\}$
$\overline{D} = (D_0, D_1, D_2, D_3)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{D_1 \in \overline{D}}}{\{0,1\} = \mathit{indexBounds}(\text{AnyArray2})}}{0 \in \mathit{indexBounds}(\text{AnyArray2})} \qquad \dfrac{\overline{D_3 \in \overline{D}}}{\mathit{isArraySetMethod}(\text{AnyArray2}\,.\,\text{Set})}}{\text{AnyArray2}\,\{1,2\}.\,\text{Set}(0,3) \to \text{AnyArray2}\,\{3,2\}}\ \text{R-Array-Set}}{\text{AnyArray2}\,\{1,2\}.\,\text{Set}(0,3).\,\text{First}() \to \text{AnyArray2}\,\{3,2\}.\,\text{First}()}\ \text{R-Context}$$

$$\dfrac{\dfrac{\overline{D_2 \in \overline{D}}}{(\text{this} : \text{AnyArray2}).\,\text{this}[0] = \mathit{body}(\text{AnyArray2}\,.\,\text{First})}}{\text{AnyArray2}\,\{3,2\}.\,\text{First}() \to \text{AnyArray2}\,\{3,2\}[0]}\ \text{R-Call}$$

$$\dfrac{\dfrac{\dfrac{\overline{D_1 \in \overline{D}}}{\{0,1\} = \mathit{indexBounds}(\text{AnyArray2})}}{0 \in \mathit{indexBounds}(\text{AnyArray2})}}{\text{AnyArray2}\,\{3,2\}[0] \to 3}\ \text{R-Index}$$

H.2 Featherweight Go with Arrays type checking

Below are derivation trees type checking two simple FGA programs, using a common set of declarations. Since each type checking derivation tree would be too large to fit on a single page, the trees have been split into multiple smaller subtrees.

$$
\begin{aligned}
D_0 &= \textbf{type}\ \text{any}\ \textbf{interface}\{\} \\
D_1 &= \textbf{type}\ \text{AnyArray2}\ [2]\,\text{any} \\
D_2 &= \textbf{func}(\text{this}\ \ \text{AnyArray2})\ \text{First}()\ \text{any}\ \{\textbf{return}\ \text{this}[0]\} \\
D_3 &= \textbf{func}(\text{this}\ \ \text{AnyArray2})\ \text{Length}()\ \textbf{int}\ \{\textbf{return}\ 2\} \\
\overline{D} &= (D_0, D_1, D_2, D_3) \\
e_1 &= \text{AnyArray2}\,\{1,2\}.\,\text{First}() \\
e_2 &= \text{AnyArray2}\,\{1,2\}.\,\text{Length}() \\
AA2 &= \text{AnyArray2}
\end{aligned}
$$

$$
\frac{\dfrac{}{\textbf{interface}\{\}\ \mathit{ok}}\ \textsc{T-Interface} \qquad \dfrac{}{\mathit{notReferenced}(\text{any}, \textbf{interface}\{\})}}{D_0\ \mathit{ok}}\ \textsc{T-Type}
$$

$$
\frac{\dfrac{}{2 \geq 0} \qquad \dfrac{\dfrac{}{D_0 \in \overline{\overline{D}}}\ \textsc{T-Named}}{\text{any}\ \ \mathit{ok}}}{[2]\,\text{any}\ \ \mathit{ok}}\ \textsc{T-Array}
$$

$$
\frac{\dfrac{}{D_0 \in \overline{\overline{D}}} \qquad \dfrac{}{\text{AnyArray2} \neq \text{any}} \qquad \dfrac{}{\mathit{notReferenced}(\text{AnyArray2}, \text{any}, \textbf{interface}\{\})}}{\mathit{notReferenced}(\text{AnyArray2}, \text{any})}
$$

$$
\frac{[2]\,\text{any}\ \ \mathit{ok} \qquad \dfrac{\mathit{notReferenced}(\text{AnyArray2}, \text{any})}{\mathit{notReferenced}(\text{AnyArray2}, [2]\,\text{any})}}{D_1\ \mathit{ok}}\ \textsc{T-Type} \qquad \frac{}{\mathit{distinct}(\text{this})}
$$

$$
\frac{\dfrac{}{D_1 \in \overline{\overline{D}}}}{\text{AnyArray2}\ \ \mathit{ok}}\ \textsc{T-Named} \qquad \frac{\dfrac{}{D_0 \in \overline{\overline{D}}}\ \textsc{T-Named}}{\text{any}\ \ \mathit{ok}}\ \textsc{T-Named} \qquad \frac{(\text{this} : AA2) \in (\text{this} : AA2)}{\text{this} : AA2 \vdash \text{this} : AA2}\ \textsc{T-Var}
$$

$$
\frac{}{\text{this} : AA2 \vdash 0 : 0}\ \textsc{T-Int-Literal} \qquad \frac{\dfrac{}{0 \leq 0 < 2}}{0 \leq 0 < \mathit{lenType}(AA2)}
$$

$$\frac{\overline{\overline{D_1 \in \overline{D}}}}{\text{any} = \mathit{elementType}(AA2)}$$

$$\frac{\begin{array}{c}\text{this} : AA2 \vdash \text{this} : AA2 \qquad \text{this} : AA2 \vdash 0 : 0 \\ 0 \leq 0 < \mathit{lenType}(AA2) \qquad \text{any} = \mathit{elementType}(AA2)\end{array}}{\text{this} : AA2 \vdash \text{this}[0] : \text{any}}\ \textsc{T-Array-Index}$$

$$\frac{\overline{\mathit{methods}(\text{any}) \supseteq \mathit{methods}(\text{any})}}{\text{any} <: \text{any}}\ \text{I}$$

$$\frac{\text{AnyArray2}\ \ \mathit{ok} \qquad \text{any}\ \ \mathit{ok} \qquad \begin{array}{c}\mathit{distinct}(\text{this}) \\ \text{this} : \text{AnyArray2} \vdash \text{this}[0] : \text{any}\end{array} \qquad \text{any} <: \text{any}}{D_2\ \ \mathit{ok}}\ \textsc{T-Func}$$

$$\frac{}{\textbf{int}\ \ \mathit{ok}}\ \textsc{T-Int-Type} \qquad \frac{}{\text{this} : \text{AnyArray2} \vdash 2 : 2}\ \textsc{T-Int-Literal} \qquad \frac{}{2 <: \textbf{int}}\ \textsc{Int-N}$$

$$\frac{\text{AnyArray2}\ \ \mathit{ok} \qquad \textbf{int}\ \ \mathit{ok} \qquad \begin{array}{c}\mathit{distinct}(\text{this}) \\ \text{this} : \text{AnyArray2} \vdash 2 : 2\end{array} \qquad 2 <: \textbf{int}}{D_3\ \ \mathit{ok}}\ \textsc{T-Func}$$

$$\frac{\mathit{methods}(1) \supseteq \mathit{methods}(\text{any})}{1 <: \text{any}}\ <:_I \qquad \frac{\mathit{methods}(2) \supseteq \mathit{methods}(\text{any})}{2 <: \text{any}}\ <:_I$$

$$\frac{}{\emptyset \vdash 1 : 1}\ \textsc{T-Int-Literal} \qquad \frac{}{\emptyset \vdash 2 : 2}\ \textsc{T-Int-Literal}$$

$$\frac{\begin{array}{c}\text{AnyArray2}\ \ \mathit{ok} \qquad \emptyset \vdash 1 : 1 \\ \emptyset \vdash 2 : 2 \qquad \text{any} = \mathit{elementType}(\text{AnyArray2}) \qquad 1 <: \text{any} \qquad 2 <: \text{any}\end{array}}{\emptyset \vdash \text{AnyArray2}\,\{1, 2\} : \text{AnyArray2}}\ \textsc{T-Array-Literal}$$

$$\frac{\overline{(\text{First}()\ \ \text{any}) \in \{(\text{First}()\ \ \text{any}), (\text{Length}()\ \textbf{int})\}}}{(\text{First}()\ \ \text{any}) \in \mathit{methods}(\text{AnyArray2})}$$

$$\frac{\emptyset \vdash \text{AnyArray2}\,\{1, 2\} : \text{AnyArray2} \qquad (\text{First}()\ \ \text{any}) \in \mathit{methods}(\text{AnyArray2})}{\emptyset \vdash e_1 : \text{any}}\ \textsc{T-Call}$$

$$\frac{D_0\ ok \qquad D_1\ ok \qquad D_2\ ok \qquad D_3\ ok}{\overline{D\ ok}}$$

$$\frac{\overline{distinct(tdecls(\overline{D}), \mathbf{int})} \qquad \overline{distinct(mdecls(\overline{D}))} \qquad \overline{D\ ok} \qquad \emptyset \vdash e_1 : \text{any}}{\mathbf{main};\ \overline{D}\ \mathbf{func\ main}()\ \{\_ = e_1\}\ ok}\ \text{T-Prog}$$

$$\frac{\overline{(\text{Length}()\ \mathbf{int}) \in \{(\text{First}()\ \text{any}), (\text{Length}()\ \mathbf{int})\}}}{(\text{Length}()\ \mathbf{int}) \in methods(\text{AnyArray2})}$$

$$\frac{\emptyset \vdash \text{AnyArray2}\,\{1,2\} : \text{AnyArray2} \qquad (\text{Length}()\ \mathbf{int}) \in methods(\text{AnyArray2})}{\emptyset \vdash e_2 : \mathbf{int}}\ \text{T-Call}$$

$$\frac{\overline{distinct(tdecls(\overline{D}))} \qquad \overline{distinct(mdecls(\overline{D}))} \qquad \overline{D\ ok} \qquad \emptyset \vdash e_2 : \mathbf{int}}{\mathbf{main};\ \overline{D}\ \mathbf{func\ main}()\ \{\_ = e_2\}\ ok}\ \text{T-Prog}$$

H.3 Featherweight Generic Go with Arrays reduction

Below are derivation trees reducing an example FGGA expression down to a value.

$D_0 = \mathbf{type}\ \text{any}\ \mathbf{interface}\{\}$
$D_1 = \mathbf{type}\ \text{Array[N}\ \mathbf{const}, \text{T}\ \ \text{any]}\ \text{[N]}\,\text{T}$
$D_2 = \mathbf{func}(\text{this}\ \ \text{Array[N, T]})\ \text{Get(i}\ \mathbf{int})\ \text{any}\ \{\mathbf{return}\ \text{this[i]}\}$
$D_3 = \mathbf{func}(\text{this}\ \ \text{Array[N, T]})\ \text{Set(i}\ \mathbf{int}, \text{v}\ \ \text{T})\ \text{AnyArray2}\ \{\,\text{this[i]} = \text{v};\ \mathbf{return}\ \text{this}\,\}$
$\overline{D} = (D_0, D_1, D_2, D_3)$

$$\frac{\dfrac{\dfrac{\overline{D_1 \in \overline{D}}}{\{0,1\} = indexBounds(\text{Array}[2, \mathbf{int}])} \qquad \dfrac{\overline{D_3 \in \overline{D}}}{isArraySetMethod(\text{Array}\,.\,\text{Set})}}{0 \in indexBounds(\text{Array}[2, \mathbf{int}])}\ \text{R-Array-Set}}{\text{Array}[2, \mathbf{int}]\{1,2\}.\,\text{Set}(0,3) \to \text{Array}[2, \mathbf{int}]\{3,2\}}\ \text{R-Context}}{\text{Array}[2, \mathbf{int}]\{1,2\}.\,\text{Set}(0,3).\,\text{Get}(0) \to \text{Array}[2, \mathbf{int}]\{3,2\}.\,\text{Get}(0)}\ \text{R-Context}$$

$$\dfrac{\dfrac{\overline{D_2 \in \overline{D}}}{(\text{this} : \text{Array}[2, \textbf{int}], i : \textbf{int}).\, \text{this}[i] = \mathit{body}(\text{Array}[2, \textbf{int}].\, \text{First})}}{\text{Array}[2, \textbf{int}]\{3, 2\}.\, \text{Get}(0) \to \text{Array}[2, \textbf{int}]\{3, 2\}[0]}\ \text{R-Call}$$

$$\dfrac{\dfrac{\dfrac{\overline{D_1 \in \overline{D}}}{\{0, 1\} = \mathit{indexBounds}(\text{Array}[2, \textbf{int}])}}{0 \in \mathit{indexBounds}(\text{Array}[2, \textbf{int}])}}{\text{Array}[2, \textbf{int}]\{3, 2\}[0] \to 3}\ \text{R-Index}$$

H.4 Featherweight Generic Go with Arrays type checking

Below is a derivation tree type checking a simple FGGA program.

$$\begin{aligned}
D_0 &= \textbf{type}\ \ \text{any}\ \ \textbf{interface}\{\} \\
D_1 &= \textbf{type}\ \ \text{Array[N}\ \ \textbf{const}, \text{T}\ \ \text{any]}\ \ \text{[N]}\ \text{T} \\
D_2 &= \textbf{func}(\text{this}\ \ \text{Array[N, T]})\ \ \text{Get(i}\ \ \textbf{int})\ \ \text{any}\ \ \{\textbf{return}\ \ \text{this[i]}\} \\
\overline{D} &= (D_0, D_1, D_2) \\
e_1 &= \text{Array}[2, \textbf{int}]\{1, 2\}.\, \text{Get}(0) \\
e_2 &= \text{Array}[2, \textbf{int}]\{1, 2\}.\, \text{Length}() \\
\overline{\Phi} &= (\text{N}\ \ \textbf{const}, \text{T}\ \ \ \text{any})
\end{aligned}$$

$$\frac{\dfrac{}{\emptyset \vdash \textbf{interface}\{\}\ ok}\ \textsc{T-Interface} \qquad \dfrac{}{\mathit{notReferenced}(\text{any}, \textbf{interface}\{\})}}{D_0\ ok}\ \textsc{T-Type}$$

$$\frac{\dfrac{}{\mathit{distinct}(\text{N}, \text{T})} \qquad \dfrac{}{\overline{\Phi} \vdash \textbf{const}\ ok}\ \textsc{T-Const} \qquad \dfrac{\dfrac{}{D_0 \in \overline{D}}}{\overline{\Phi} \vdash \text{any}\ ok}\ \textsc{T-Named}}{\overline{\Phi}\ ok}\ \textsc{T-Formal}$$

$$\frac{\dfrac{\dfrac{}{(\text{N} : \textbf{const}) \in \overline{\Phi}}}{\overline{\Phi} \vdash \text{N} <: \textbf{const}}\ \textsc{Const-Param}}{\mathit{isConst}_{\overline{\Phi}}(\text{N})} \qquad \frac{\dfrac{}{(\text{N} : \textbf{const}) \in \overline{\Phi}}}{\overline{\Phi} \vdash \text{N}\ ok}\ \textsc{T-Param}$$

$$\frac{\overline{\Phi} \vdash \text{N}\ ok \qquad \mathit{isConst}_{\overline{\Phi}}(\text{N}) \qquad \dfrac{\dfrac{}{(\text{T} : \text{any}) \in \overline{\Phi}}}{\overline{\Phi} \vdash \text{T}\ ok}\ \textsc{T-Param} \qquad \dfrac{}{\neg \mathit{isConst}_{\overline{\Phi}}(\text{T})}}{\overline{\Phi} \vdash [\text{N}]\,\text{T}\ ok}\ \textsc{T-Array}$$

$$\frac{}{\mathit{notReferenced}_{\alpha}(\text{Array}, \textbf{const})} \qquad \frac{\dfrac{}{\text{Array} \neq \text{any}}}{\mathit{notReferenced}_{\alpha}(\text{Array}, \text{any})}$$

$$\frac{\dfrac{}{\mathit{notReferenced}(\text{Array}, \text{T})}}{\mathit{notReferenced}(\text{Array}, [\text{N}]\,\text{T})}$$

$$\frac{\begin{array}{ccc}\overline{\Phi}\ ok & \mathit{notReferenced}_{\alpha}(\text{Array}, \textbf{const}) & \\ \mathit{notReferenced}_{\alpha}(\text{Array}, \text{any}) & \overline{\Phi} \vdash [\text{N}]\,\text{T}\ ok & \mathit{notReferenced}(\text{Array}, [\text{N}]\,\text{T})\end{array}}{D_1\ ok}\ \textsc{T-Type}$$

$$\frac{}{\mathit{distinct}(\text{this})} \qquad \frac{}{D_1 \in \overline{D}} \qquad \frac{}{\neg \mathit{isConst}_{\overline{\Phi}}(\textbf{int})} \qquad \frac{}{\overline{\Phi} \vdash \textbf{int}\ ok}\ \textsc{T-Int-Type}$$

$$\frac{\dfrac{}{(\text{T} : \text{any}) \in \overline{\Phi}}}{\overline{\Phi} \vdash \text{T}\ ok}\ \textsc{T-Param} \qquad \frac{D_1 \in \overline{D}}{\overline{\Phi} = \mathit{typeParams}(\text{Array})}$$

$$\frac{\eta_0 = (\mathrm{N} := \mathrm{N}, \mathrm{T} := \mathrm{T})}{\eta_0 = (\mathrm{N}\ \ \mathbf{const} := \mathrm{N}, \mathrm{T}\ \ \mathrm{any} := \mathrm{T})} \qquad \frac{D_1 \in \overline{D}}{\mathrm{T} = \mathit{elementType}(\mathrm{Array})[\eta_0]}$$

$$\frac{(\mathrm{this} : \mathrm{Array}[\mathrm{N}, \mathrm{T}]) \in (\mathrm{this} : \mathrm{Array}[\mathrm{N}, \mathrm{T}], \mathrm{i} : \mathbf{int})}{\overline{\Phi}; \mathrm{this} : \mathrm{Array}[\mathrm{N}, \mathrm{T}], \mathrm{i} : \mathbf{int} \vdash \mathrm{this} : \mathrm{Array}[\mathrm{N}, \mathrm{T}]}\ \textsc{T-Var}$$

$$\frac{(\mathrm{i} : \mathbf{int}) \in (\mathrm{this} : \mathrm{Array}[\mathrm{N}, \mathrm{T}], \mathrm{i} : \mathbf{int})}{\overline{\Phi}; \mathrm{this} : \mathrm{Array}[\mathrm{N}, \mathrm{T}], \mathrm{i} : \mathbf{int} \vdash \mathrm{i} : \mathbf{int}}\ \textsc{T-Var}$$

$$\frac{\begin{array}{c}\overline{\Phi} = \mathit{typeParams}(\mathrm{Array}) \\ \eta_0 = (\mathrm{N}\ \ \mathbf{const} := \mathrm{N}, \mathrm{T}\ \ \mathrm{any} := \mathrm{T}) \qquad \mathrm{T} = \mathit{elementType}(\mathrm{Array})[\eta_0] \\ \overline{\Phi}; \mathrm{this} : \mathrm{Array}[\mathrm{N}, \mathrm{T}], \mathrm{i} : \mathbf{int} \vdash \mathrm{this} : \mathrm{Array}[\mathrm{N}, \mathrm{T}] \\ \overline{\Phi}; \mathrm{this} : \mathrm{Array}[\mathrm{N}, \mathrm{T}], \mathrm{i} : \mathbf{int} \vdash \mathrm{i} : \mathbf{int}\end{array}}{\overline{\Phi}; \mathrm{this} : \mathrm{Array}[\mathrm{N}, \mathrm{T}] \vdash \mathrm{this}[\mathrm{i}] : \mathrm{T}}\ \textsc{T-Array-Index}$$

$$\frac{}{\overline{\overline{\Phi} \vdash \mathrm{T} <: \mathrm{T}}}\ <:_{\textsc{Param}} \qquad \frac{}{\overline{\neg \mathit{isConst}_{\overline{\Phi}}(T)}}$$

$$\frac{\begin{array}{c}\mathit{distinct}(\mathrm{this}) \qquad D_1 \in \overline{D} \qquad \neg \mathit{isConst}_{\overline{\Phi}}(\mathbf{int}) \qquad \overline{\Phi} \vdash \mathbf{int}\ \ \mathit{ok} \\ \overline{\Phi} \vdash \mathrm{T}\ \ \mathit{ok} \qquad \overline{\Phi}; \mathrm{this} : \mathrm{Array}[\mathrm{N}, \mathrm{T}] \vdash \mathrm{this}[\mathrm{i}] : \mathrm{T} \qquad \overline{\Phi} \vdash \mathrm{T} <: \mathrm{T} \qquad \neg \mathit{isConst}_{\overline{\Phi}}(T)\end{array}}{D_2\ \ \mathit{ok}}\ \textsc{T-Func}$$

$$\frac{\overline{2 \geq 0}}{\emptyset \vdash 2 <: \mathbf{const}}\ <:_{const-n} \qquad \frac{\overline{\mathit{methods}_{\emptyset}(\mathbf{int}) \supseteq \mathit{methods}_{\emptyset}(\mathrm{any})}}{\emptyset \vdash \mathbf{int} <: \mathrm{any}}\ <:_{I}$$

$$\frac{\dfrac{\eta_1 = (\mathrm{N} := 2, \mathrm{T} := \mathbf{int})}{\eta_1 = (\mathrm{N}\ \ \mathbf{const} := 2, \mathrm{T}\ \ \mathrm{any} := \mathbf{int})} \qquad \emptyset \vdash 2 <: \mathbf{const} \qquad \emptyset \vdash \mathbf{int} <: \mathrm{any}}{\eta_1 = (\mathrm{N}\ \ \mathbf{const} :=_{\emptyset} 2,\ \ \mathrm{T}\ \ \mathrm{any} :=_{\emptyset} \mathbf{int})}$$

$$\frac{(\mathrm{Get}(\mathrm{i}\ \ \mathbf{int})\ \ \mathbf{int}) \in \{(\mathrm{Get}(\mathrm{i}\ \ \mathbf{int})\ \ \mathrm{T})[\eta_1], (\mathrm{Length}()\ \ \mathbf{int})[\eta_1]\}}{(\mathrm{Get}(\mathrm{i}\ \ \mathbf{int})\ \ \mathbf{int}) \in \mathit{methods}_{\emptyset}(\mathrm{Array}[2, \mathbf{int}])}$$

$$\frac{\dfrac{\overline{2 \geq 0}}{\emptyset \vdash 2\ \ \mathit{ok}}\ \textsc{T-N-Type} \qquad \dfrac{}{\emptyset \vdash \mathbf{int}\ \ \mathit{ok}}\ \textsc{T-Int-Type} \qquad \overline{\overline{D_1 \in \overline{D}}} \qquad \eta_1}{\emptyset \vdash \mathrm{Array}[2, \mathbf{int}]\ \ \mathit{ok}}\ \textsc{T-Named}$$

$$\frac{\overline{D_1 \in \overline{D}}}{\overline{\overline{\Phi}} = \mathit{typeParams}(\text{Array})} \qquad \frac{\eta_2 = (\text{N} :=2,\ \ \text{T} := \textbf{int})}{\eta_2 = (\text{N}\ \ \textbf{const} := 2,\ \ \text{T}\ \ \text{any} := \textbf{int})}$$

$$\frac{\textbf{int} = \text{T}[\eta_2]}{\textbf{int} = \mathit{elementType}(\text{Array})[\eta_2]} \qquad \frac{}{\emptyset; \emptyset \vdash 1 : 1, 2 : 2}\ \text{T-Int-Literal}$$

$$\frac{}{\emptyset \vdash 1 <: \textbf{int}}\ <:_{int-n} \qquad \frac{}{\emptyset \vdash 2 <: \textbf{int}}\ <:_{int-n}$$

$$\frac{\begin{array}{c}\emptyset \vdash \text{Array}[2, \textbf{int}]\ \mathit{ok} \qquad \overline{\Phi} = \mathit{typeParams}(\text{Array}) \\ \eta_2 = (\text{N} :=2, \text{T} :=\textbf{int}) \qquad \textbf{int} = \mathit{elementType}(\text{Array})[\eta_2] \\ \emptyset; \emptyset \vdash 1 : 1, 2 : 2 \qquad \emptyset \vdash 1 <: \textbf{int} \qquad \emptyset \vdash 2 <: \textbf{int}\end{array}}{\emptyset; \emptyset \vdash \text{Array}[2, \textbf{int}]\{1, 2\} : \text{Array}[2, \textbf{int}]}\ \text{T-Array-Literal}$$

$$\frac{}{\emptyset; \emptyset \vdash 0 : 0}\ \text{T-Int-Literal} \qquad \frac{}{\emptyset \vdash 0 <: \textbf{int}}\ \text{Int-N}$$

$$\frac{\begin{array}{c}(\text{Get(i}\ \ \textbf{int})\ \ \textbf{int}) \in \mathit{methods}_{\emptyset}(\text{Array}[2, \textbf{int}]) \\ \emptyset; \emptyset \vdash \text{Array}[2, \textbf{int}]\{1, 2\} : \text{Array}[2, \textbf{int}] \qquad \emptyset; \emptyset \vdash 0 : 0 \qquad \emptyset \vdash 0 <: \textbf{int}\end{array}}{\emptyset; \emptyset \vdash e_1 : \textbf{int}}\ \text{T-Call}$$

$$\frac{D_0\ \mathit{ok} \qquad D_1\ \mathit{ok} \qquad D_2\ \mathit{ok} \qquad D_3\ \mathit{ok}}{\overline{D\ \mathit{ok}}}$$

$$\frac{\overline{\mathit{distinct}(\mathit{tdecls}(\overline{D}), \textbf{int})} \qquad \overline{\mathit{distinct}(\mathit{mdecls}(\overline{D}))} \qquad \overline{D\ \mathit{ok}} \qquad \emptyset; \emptyset \vdash e_1 : \textbf{int}}{\textbf{main};\ \overline{D}\ \textbf{func main}()\ \{\_ = e_1\}\ \mathit{ok}}\ \text{T-Prog}$$