

<https://researchcommons.waikato.ac.nz/>

Research Commons at the University of Waikato

Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.

Generics in Go and Lambda Lifting

A thesis
submitted in fulfilment
of the requirements for the degree
of
Master of Science (Research) in Computer Science
at
The University of Waikato
by
Tom Levy



THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2026

Abstract

This thesis examines the history of generics in the Go programming language, and analyses two specific issues that arise: local type declarations inside generic functions, and embedded type parameters.

This work includes a novel algorithm for lambda lifting, a problem closely related to the handling of local types inside generic functions in Go. The correctness of this algorithm is proven with respect to both a specification from the literature and additional specifications. Parts of these proofs are formalised using the Lean 4 theorem prover.

For embedded type parameters, this thesis proposes sound member promotion rules for type parameters embedded in struct types in Go.

The testing methodology used in this thesis centres around *checking*, which strikes a middle ground between formal verification and testing.

To my parents.

Acknowledgements

I would like to thank my supervisor, Steve Reeves, for his guidance and steady support; and Bill Rogers, for supervising my earlier related work with patience and unending curiosity.

I would also like to thank Marco Morazán for validating and reviewing the work on lambda lifting.

Contents

1	Introduction	1
1.1	Lambda lifting	2
1.2	Generics in the Go programming language	3
1.2.1	The need for generics	4
1.2.2	Local types inside generic functions	6
1.2.3	Embedded type parameters	8
2	Testing Methodology	10
3	Background on Generics	14
3.1	Generics in other languages	15
3.1.1	Java	15
3.1.2	C#	16
3.1.3	C++	16
3.1.4	Haskell	17
3.2	Terminology	18
4	History of Generics in Go	20
4.1	Alternatives to generics	21
4.1.1	Code duplication	21
4.1.2	Code generation	22
4.1.3	<code>interface{}</code> and type assertions	23
4.1.4	Reflection	24
4.1.5	Adapters	26
4.2	First-class generic functions	28
4.3	Proposals	30
4.4	Implementation	36
4.5	Literature on generics in Go	37
4.6	Future	40
5	Lambda Lifting	50
5.1	Introduction	51
5.2	Literature	54
5.2.1	The Danvy–Schultz algorithm	56
5.2.2	CertiCoq	59

5.3	Syntax	60
5.4	The algorithm	62
5.4.1	Definitions	62
5.4.2	A simple lifting example	64
5.4.3	A complex lifting example	65
5.4.4	Algorithm 1	66
5.4.5	Algorithm 1b	69
5.4.6	Data structure implementation	71
5.4.7	Optimality and improvements	74
5.5	Specification	77
5.5.1	Fischbach–Hannan lifting specification	77
5.5.2	Johnsson-style lifting	79
5.5.3	Complete lifting	81
5.6	Theorems	82
5.7	Lean implementation	86
5.8	Testing	94
5.8.1	Test case generation	94
5.8.2	Correctness checks	96
5.8.3	Testing Johnsson’s algorithm	98
5.8.4	Testing the Fischbach–Hannan algorithm	99
5.8.5	Results	101
5.8.6	Integration with Plausible	105
5.9	Related problems	107
6	Local Types in Go	109
6.1	Algorithm	111
6.2	Backward compatibility	112
7	Embedded Type Parameters in Go	114
7.1	Introduction	114
7.2	Possible solutions	117
7.3	Motivation	118
7.4	Design	121
7.5	Proposed rules	124
7.5.1	Definitions	125
7.5.2	Algorithm 2	128
7.6	Infinite recursion	132
7.7	Embedding in anonymous structs	133
7.8	Examples	134
7.9	Testing	141
8	Conclusion	142
8.1	Future work	143
	References	144

Appendices	155
A Worst-case lambda lifting output size	155
B Johnsson’s algorithm	158

List of Figures

4 History of Generics in Go

4.1	An example of generics from [74]	47
4.2	Runtime representation for dictionary-passing	47
4.3	Compilation of the example using dictionary-passing	49

5 Lambda Lifting

5.1	The Fischbach–Hannan lifting specification	78
5.2	Johnsson-style lifting	80
5.3	Complete lifting	82
5.4	Additional basic definitions	88
5.5	Type system	90
5.6	Lean translation of the Fischbach–Hannan lifting specification	92
5.7	Declaration graph of a randomly-generated test case	97
5.8	Call graph of a randomly-generated test case	97

7 Embedded Type Parameters in Go

7.1	Embedded type parameters and unsound promotion	116
7.2	Motivational example for embedded type parameters	120

List of Tables

5 Lambda Lifting

5.1	Number of test cases	101
5.2	Results	102
5.3	Running time to test each implementation	103
5.4	Total running time	104

Chapter 1

Introduction

Algorithms are the core of computer science [29]. They are widespread in the field, and many areas of computer science rely on them at a foundational level. They make it possible to efficiently solve problems which otherwise could not be solved. In many cases the same common algorithms are used in multiple areas, and get implemented and re-implemented by many programmers. Improving these algorithms by making them more efficient and simpler is advantageous.

Improving long-standing algorithms carries with it the burden of showing that the improved algorithm is (still) correct. This thesis uses a combination of formal verification (including “pen and paper” proofs and machine-checked proofs) and *checking* [4], which involves verifying that properties hold on specific inputs rather than on all inputs; checking is easier than formal verification but provides weaker guarantees.

This thesis examines two examples of such algorithms in the field of programming language design and implementation, with a specific focus on generics in the Go programming language. The first is a novel algorithm for lambda lifting. The second is a sound promotion algorithm for embedded type parameters in the Go programming language. These examples have been identified because there is a practical need for them, and improvement is desired.

The research questions and hypothesis of this thesis are:

1. In what ways can we improve algorithms used in the field of programming language design and implementation?
2. Can these improvements be applied to real-world software?
3. How can we verify that our improved algorithms are correct?

4. Hypothesis: Many areas of computer science depend on algorithms at a foundational level. Improving these algorithms, which are often widespread and common, can benefit those areas.

The thesis supports this hypothesis with two examples.

The thesis is organised into eight chapters. [Chapter 2](#) describes the checker approach. [Chapter 3](#) gives a broad introduction to generics, while [Chapter 4](#) discusses the history of generics in Go and a possible future direction. [Chapter 5](#) is an extended version of a paper on lambda lifting [49] presented in the IEEE/ACM 14th International Conference on Formal Methods in Software Engineering (FormaliSE '26), co-authored with Steve Reeves. [Chapter 6](#) discusses the related problem of local types in Go, and [Chapter 7](#) describes a promotion algorithm for embedded type parameters in Go. [Chapter 8](#) concludes.

We now give a detailed introduction to lambda lifting and to generics in Go.

1.1 Lambda lifting

Many programming languages allow functions to be declared inside other functions, a feature known as “nested functions” or “local functions”. It is useful because it allows programmers to avoid namespace pollution, group code logically, and most importantly, because it allows the local functions to refer to identifiers from the enclosing scope. Access to the enclosing scope can make the code more concise and improve readability, since values from the enclosing scope do not need to be passed explicitly to the local functions. Further, in languages that allow functions to be used as values, nested functions offer a convenient way to create functions that have associated data, such as a callback with associated state.

However, it can be challenging to handle nested functions in compilers due to the non-local references to variables from enclosing scopes. In particular, the use of nested functions as values is incompatible with stack-based function calling; this is known as the “funarg” problem [91, 58].

Lambda lifting [35] is a technique that converts a program containing nested functions into an equivalent program that does not make use of nested functions. This is useful because it means later compilation passes do not need to handle

nested functions. The idea is to “lift” each local function to the top-level scope, and introduce additional parameters as necessary to satisfy references to free variables. Determining the required set of additional parameters is non-trivial due to the existence of mutually-recursive functions; this task is the core of the lambda lifting algorithm.

The original lambda lifting algorithm [35] has time complexity $O(n^3)$. A series of papers [15, 55] has led to an $O(n^2)$ algorithm [56] for performing lambda lifting, however this algorithm is complex.

In Chapter 5 we present a simple algorithm whose worst-case time complexity is $O(n^2)$ like the fastest existing algorithm. We go further and show that the time complexity of our algorithm is linear in the size of the output program.

We prove that our algorithm is correct with respect to an existing formal specification from the literature [19] and with respect to our notion of a “complete” and “minimal” lifting, making it the first algorithm proven to compute the *minimal* required set of additional parameters.

We also formalise the lambda lifting specification from [19] in Lean 4, and use that to prove some of the properties and test our algorithm on a large suite of generated test cases.

Chapter 5 concludes with a brief description of the connection between lambda lifting and local types inside generic functions in the Go programming language.

1.2 Generics in the Go programming language

Go is a statically typed, compiled programming language developed at Google [66]. It was conceived in 2007 and became an open source project in 2009.¹ Go was designed as a systems language with garbage collection and well-integrated concurrency [20]. Two of the core goals of Go are fast compilation [66] and backward compatibility [14].

One of the major omissions in Go’s original type system was generics—the ability for types and functions to have type parameters. Generics was the most-requested feature in the annual Go user surveys of 2016–2017 and 2019–2021.²

¹<https://go.dev/doc/faq#history> “Frequently Asked Questions (FAQ)”. The Go Documentation. Retrieved 7 July 2026.

²<https://go.dev/blog/survey2016-results>, <https://go.dev/blog/survey2017-results>,

The need for generics was the original motivation for this research.

For a long time the Go developers were resistant to adding generics due to the complexity and the challenge of finding a design that is expressive and performant.³ After several rejected proposals, generics were added in 2022. The implementation in the compiler is challenging and there is room for further work.

1.2.1 The need for generics

To illustrate the need for generics, consider the following simple Go function that reverses a slice (resizable array) of integers:

```
func ReverseInts(original []int) []int {
    reversed := make([]int, len(original))
    for i := 0; i < len(original); i++ {
        reversed[len(original)-i-1] = original[i]
    }
    return reversed
}
```

Variations of this example are commonly used to illustrate generics [82], with the natural question being how this function could be generalised to support slices with other element types.

Some notes on the syntax of Go:

- The type `[]T` denotes a slice (resizable array) with elements of type `T`.
- The types of function parameters are written after their names, and the return type of a function is written after the parameter list; this is the opposite from the C family.
- The `:=` operator is a shorthand variable declaration. The type of the variable is inferred from the type of the right-hand side.
- The built-in function `make` creates slices (and values of other built-in types). It is special in that its first parameter is a type; user-defined functions cannot take types as arguments.

<https://go.dev/blog/survey2019-results>, <https://go.dev/blog/survey2020-results>, <https://go.dev/blog/survey2021-results>

³https://go.dev/doc/faq#beginning_generics “Frequently Asked Questions (FAQ)”. The Go Documentation. Retrieved 7 July 2026.

A function that does the same but for a slice of strings would look as follows:

```
func ReverseStrings(original []string) []string {
    reversed := make([]string, len(original))
    for i := 0; i < len(original); i++ {
        reversed[len(original)-i-1] = original[i]
    }
    return reversed
}
```

This function is identical, except that the type `int` has been replaced with `string`.

The Go community is not averse to copying code,⁴ but that becomes unmaintainable when the amount of code and the number of copies grow; in particular, fixing bugs becomes a significant burden.

Prior to the addition of generics, there was no satisfactory general-purpose approach for writing code (such as for reversing a slice) that could be used with arbitrary types; we discuss the approaches that were available and their shortcomings in [Section 4.1](#).

With generics, a function `Reverse` can be written as follows:

```
func Reverse[T any](original []T) []T {
    reversed := make([]T, len(original))
    for i := 0; i < len(original); i++ {
        reversed[len(original)-i-1] = original[i]
    }
    return reversed
}
```

This function has a *type parameter* named `T`. The keyword `any` is an alias for the empty interface, `interface{}`, and indicates that there are no constraints on `T`. To use this function, one would write:

```
var numbers []int = []int{1, 2, 3}
numbers = Reverse[int](numbers)
var words []string = []string{"hello", "world"}
words = Reverse[string](words)
```

Type inference allows the type argument to be omitted in the above cases, so one could equivalently write `Reverse(numbers)`.

⁴“A little copying is better than a little dependency”. Rob Pike. Go Proverbs. Gopherfest 2015. <https://www.youtube.com/watch?v=PAAkCSZUG1c&t=9m28s>

1.2.2 Local types inside generic functions

Go has supported local type declarations inside functions since the very first version.⁵ Before the addition of generics, their sole purpose was to limit the scope of the declared type—local type declarations could be trivially converted to global type declarations by choosing unique identifiers.

With the addition of generics, local type declarations have become much more interesting and useful, because the declaration may refer to the type parameters of the enclosing function (as demonstrated in the example below).

Handling such local type declarations in the compiler directly is challenging; the most sensible approach is to automatically convert them to global types during an early compilation pass. This requires introducing type parameters to the type declarations, analogously to lambda lifting; these type parameters are called *implicit type parameters* in Go.

The existing implementation in the Go compiler inserts *all* the type parameters from the enclosing scope as implicit type parameters, even if they are not used in the type declaration.⁶ This choice has an observable effect on type identity.

We argue that type parameters should only be added as implicit type parameters if they are referenced by the local type (directly or indirectly) to match the expectations of programmers. This corresponds to the notion of a “minimal lifting”. We support our position with an analysis of the corpus of publicly available Go source code.

The following example demonstrates the usefulness of local types inside generic functions. It is a partial code fragment that shows how any unstable sorting algorithm can be used to implement stable sorting (where the order of equal elements is preserved), by recording the original index of each element and using that to break ties in comparisons. The function has a type parameter called `E`, and contains a local type declaration of a struct type `ElemWithIndex` that has a field of type `E`. (The type `ElemWithIndex` is necessary because Go does not provide a built-in “Pair” type.)

⁵https://go.dev/doc/go_spec, lines 953 and 848.

⁶<https://github.com/golang/go/issues/58573> “spec: are local types in different instantiations of a generic function identical?”

```

func StableSort[E any](elements []E, cmp func(a, b E) int) []E {
    type ElemWithIndex struct {
        element    E
        origIndex int
    }
    indexed := make([]ElemWithIndex, len(elements))
    for i, e := range elements {
        indexed[i] = ElemWithIndex{e, i}
    }
    ... Sort(indexed, func(a, b ElemWithIndex) int { ... }) ...
    return ...
}

```

If we wish to convert `ElemWithIndex` to a global type (either manually or during compilation), we need to add a type parameter and insert corresponding type arguments at each reference to `ElemWithIndex`, to allow it to continue to refer to `E`:

```

type ElemWithIndex[E any] struct {
    element    E
    origIndex int
}
func StableSort[E any](elements []E, cmp func(a, b E) int) []E {
    indexed := make([]ElemWithIndex[E], len(elements))
    for i, e := range elements {
        indexed[i] = ElemWithIndex[E]{e, i}
    }
    ... Sort(indexed, func(a, b ElemWithIndex[E]) int { ... }) ...
    return ...
}

```

Performing this transformation manually is tedious and adds verbosity, so allowing local type declarations to refer to the type parameters of the enclosing function is convenient for programmers.

In that example, `E` must clearly be added as an implicit type parameter of `ElemWithIndex` for the converted program to be valid. However, if the function declared a type such as `type Point struct { x, y int}` that does not reference the type parameter `E`, it is not clear whether this type should have `E` as an implicit type parameter. We argue that adding unreferenced type parameters as implicit type parameters is surprising and unlikely to match the programmer's intent.

1.2.3 Embedded type parameters

Embedded fields [20, §Struct types] are a feature of Go that allows a form of composition which resembles subclassing.⁷ A struct type may contain embedded field declarations, written as a type without a field name. Methods of the embedded field are *promoted* to methods of the enclosing type; the effect is as if the programmer wrote wrapper methods that delegate to the corresponding methods of the embedded field. If multiple embedded fields have methods with the same name, disambiguation rules are used to determine which of these methods will be promoted [20, §Selectors].⁸

As an aside, the key difference between Go’s embedding and subclassing is that embedding does not allow *open recursion* [65, §18.1], as demonstrated by the following ill-advised program:

```
package main

import "math"

type Vec2D struct {
    X, Y float64
}

func (v Vec2D) MagSquared() float64 { return v.X*v.X + v.Y*v.Y }
func (v Vec2D) Mag() float64 { return math.Sqrt(v.MagSquared()) }

type Vec3D struct {
    Vec2D // Embedded field
    Z      float64
}

func (v Vec3D) MagSquared() float64 {
    return v.X*v.X + v.Y*v.Y + v.Z*v.Z
}

func main() {
    v := Vec3D{Vec2D{1, 1}, 1}
    println(v.Mag())
    // Output: 1.4142135623730951
}
```

⁷https://go.dev/doc/faq#Is_Go_an_object-oriented_language “Frequently Asked Questions (FAQ)”. The Go Documentation. Retrieved 7 July 2026.

⁸If the enclosing type does not declare a method with that name, and there is a unique shallowest method (based on the number of levels of embedded fields), then that method is promoted; otherwise no method with that name is promoted. Go does not allow method overloading (multiple methods with the same name but different type signatures).

The syntax `func (v Vec2D) Mag()` declares a method with receiver type `Vec2D`; by convention, the receiver’s name is a short identifier rather than `this` or `self`. In this example the method `Vec2D.Mag` is promoted to `Vec3D.Mag`. Calling `Vec3D.Mag` is equivalent to calling the method `Vec2D.Mag` of the embedded field, which in turn calls `Vec2D.MagSquared` (not `Vec3D.MagSquared`); hence the output of the program is $\sqrt{2}$, not $\sqrt{3}$.

The interaction between embedding and generics is complex. The accepted proposal for generics allows the type parameters of generic struct types to be used as embedded field types [85, §Embedded type parameter], but the early compiler implementation was unsound (as we demonstrate in [Section 7.1](#)). Embedded type parameters were disallowed prior to the release of generics due to other reasons.

One simple approach for allowing embedded type parameters is to only promote methods specified in the type parameter’s constraint, and not promote the other methods of the corresponding type argument. This approach is sound, but we argue that it does not increase the expressive power of the language and therefore would not be a useful language feature.

We address this by proposing sound promotion rules for embedded type parameters that allow promotion of the methods of corresponding type arguments. These rules are complex, but they allow interesting and useful programming idioms, such as the “mixin” pattern [5].

Chapter 2

Testing Methodology

Software verification approaches can be loosely classified into two broad categories: testing and formal verification; here we also introduce the category of checking. Formal verification proves the desired properties hold on all inputs, but requires significant effort. Testing generally requires much less effort, but gives a lower level of assurance—it only shows that the properties hold on the examples tested. Checking [4] verifies that properties hold on arbitrary given inputs.

For the lambda lifting algorithms, the strongest assurance of correctness would clearly be fully machine-checked proofs, but constructing such proofs is very difficult due to the level of detail required (and each algorithm would require a separate analysis). Instead, we take a pragmatic hybrid approach: we give a “pen and paper” proof of the correctness of our simple algorithm (with some parts formalised in Lean), and we use testing to obtain some assurance regarding the correctness of a selection of other algorithms (and some additional assurance for our algorithm). This is feasible to implement—in particular, the testing framework can be reused across all algorithms, so the only additional work required for each algorithm is to provide an implementation.

The inherent limitations of this approach are that paper proofs are susceptible to human error, and exhaustive enumeration of the test cases is not possible. Nevertheless, we feel that the level of assurance given by this approach is more than satisfactory for our purposes.

The other usual limitations also apply—the specifications that we test against might be incorrect or incomplete; there could be bugs in the Lean kernel, compiler, or interpreter; there could be bugs in the operating system or

hardware on which the tests are performed; and so on.

Besides formal verification, another approach is to check the results of the algorithm during each execution rather than certify the algorithm itself. This idea has been explored in the literature.

Blum and Kanna [4] give an excellent introduction to checkers, contrasting them with verification. They describe checking as “the simpler task of verifying that a given program returns a correct answer on a given input rather than on all inputs”, and state that “checking is not as good as verification, but it is easier to do”. They focus on probabilistic checkers (based on the idea of probabilistic interactive proofs from cryptography); simplifying, for a mathematical function $\pi(x)$ and for any program $P(x)$ that supposedly computes $\pi(x)$, they define a program checker as a probabilistic algorithm $C_\pi^P(I; k)$ that reports¹ whether $P(I) = \pi(I)$ and is incorrect with probability less than $1/2^k$. Note that the checker is independent of P , despite what the notation appears to suggest; [90, p. 830] states explicitly that P is an input of the checker. Interactive checkers may invoke P on arbitrary inputs, while non-interactive checkers only invoke P on I (this is mentioned in [4, §6] and described more explicitly in [90]). Non-probabilistic checkers can be defined by replacing $1/2^k$ with 0. The definition only supports checking for equality, but it could be generalised to accept an arbitrary proposition $\phi(I, O)$ by replacing $P(I) = \pi(I)$ with $\phi(I, P(I))$. Blum and Kanna briefly consider the idea of proving checkers correct, but instead they choose to only require checkers to be in some way independent of the program being checked; this is enforced by requiring the running time of the checker to be $o(T(n))$ where $T(n)$ is the smallest known computation time for π [90].

Goerigk et al. [21] propose an approach which integrates the checker’s result into a formal proof of a postcondition. Briefly, for a partial function $f(x)$ with precondition $P(x)$ and postcondition $Q(x, y)$, the idea is to divide Q into two parts, a proposition $Q_1(x, y)$ and a checker $q_2(x, y)$, then prove $P(x) \wedge \text{defined}(f(x)) \Rightarrow Q_1(x, f(x))$ and $Q_1(x, y) \wedge q_2(x, y) \Rightarrow Q(x, y)$. This proves that the partial function

$$x \mapsto \text{if } \text{defined}(f(x)) \wedge q_2(x, f(x)) \text{ then } f(x) \text{ else } \text{undefined}$$

¹The actual definition is slightly looser: if $P(I) = \pi(I)$ but there exists some x such that $P(x) \neq \pi(x)$, then the checker is permitted to report anything; this rule is refined in [90].

is partially correct with respect to P and Q . Note that the proofs are for a particular implementation of f —the proof of Q_1 may use the fact that $y = f(x)$ and rely on the implementation details of f . This is the most suitable approach for our use cases, mainly because of the integration with formal verification.

Mehlhorn et al. [52] define the notion of a “certifying algorithm”, building on earlier work [73, 53, 41]. A certifying algorithm is an algorithm that produces with each output a *witness* which can be used by a checker to confirm that the output is correct. For example, a certifying algorithm for determining whether a graph is bipartite could return a two-colouring of the graph as the witness when it reports the graph is bipartite, or an odd-length cycle as the witness when it reports the graph is non-bipartite [52, §2.1]. Mehlhorn et al. insist that checkers must be simple programs, and that the witness may only be trivial in exceptional cases. They mention that one option for ensuring the correctness of a checker is to use formal verification, while the other option (which they appear to favour) is to pragmatically assume that the checker is correct thanks to its simplicity. This definition is a poor fit for our use cases, because our checkers generally do not require witnesses (and because we favour formal verification over simplicity).

A related body of work is the design by contract approach [54] for object-oriented programming, which could be loosely summarised as specifying preconditions, postconditions, and class invariants using assertions². The earlier works do not have a direct link to formal verification; the Java Modeling Language [44] is notable in that it uses the same syntax for both runtime assertion checking and formal verification. One of the major advantages of Lean is that it was designed from the outset as both a theorem prover and a general-purpose functional programming language, so it offers seamless integration between the two. Another related field is Runtime Verification [1], which deals with analysing the execution trace of a stateful system; it is not applicable to our use cases.

A notable example of the checker approach is [59], a certifying compiler for a subset of the C programming language that produces a formal proof of type safety for the generated assembly code on success.

²These assertions equate to runtime checks. Interestingly, some of the earliest uses of the term “assertion” in computer science [23, 87, 32] were in the context of proving correctness rather than runtime checking.

A recent example from a different field is [38], which deals with verifying collision avoidance for a polygonal object moving in the plane along a path. A paper proof of the method was given, and the implementation generates a machine-checkable proof for each output (limited to rectangular objects). The proofs are checked using PVS (Prototype Verification System [61, 62]), a specification language integrated with a theorem prover; the algorithm is implemented in Python.

The fundamental limitation of the checker approach is that there is no guarantee of availability—it is possible that the check will fail when the algorithm is executed on certain inputs. In the case of lambda lifting (and compilers in general), availability does not seem to be a major concern [45], so checking is a good alternative to verification.

Having considered the variations of the checker approach found in the literature, we characterise it as follows for this thesis:

1. A checker is a program that determines whether a proposition holds on an output of another program.
2. Checkers should be verified. One option is to formally prove that the checker is correct (preferably using a machine-checked proof). Another option is to write a certifying checker that constructs a proof of the proposition (for example a proof that can be verified by a theorem prover³).
3. There should be good reason to believe that the proposition will always hold—the primary purpose of the checker is verification, not control flow for error handling.
4. The checker should support all valid inputs (in contrast with tests, which are often limited to a fixed set of known inputs and outputs).
5. The program being checked may include in its output a *witness* [52] to assist the checker, but this is optional.

We used this approach to test our lambda lifting algorithm and our promotion algorithm for Go.

³In Lean, one can directly build a Lean proof and return the proof alongside a value using `Subtype`. This technique does not seem to be widely used; it is discussed at [69].

Chapter 3

Background on Generics

Generics, also known as parametric polymorphism, is a language facility that allows types and functions to have type parameters. The term “parametric polymorphism” was coined by Strachey in 1967 [71, 50]; the term “generics” was popularised by Java, appearing in 1997 [6]. Generics is important in statically-typed programming languages for allowing programmers to write type-safe code that can operate on different types, as shown in [Section 1.2.1](#).

Even statically-typed languages without generics typically have some highly-limited form of “generics”, restricted to built-in types, operators, and functions. For example, most languages allow programmers to declare arrays with a particular element type, such as an array of integers. As another example, the built-in function `append` in the Go programming language adds an element to a *slice* (Go’s resizable array type); its type signature cannot be represented without type parameters, so it is handled in the type checker as a special case. Generics allow programmers to declare their own types and functions with a similar level of expressiveness.

In this chapter we mention some of the implementations of generics used in existing programming languages, and give a glossary that summarises the key concepts.

3.1 Generics in other languages

3.1.1 Java

Java is a widely used object-oriented programming language. Generics were added to the language in 2004 with the J2SE 1.5 release (after earlier research projects [6]).

Generics in Java are a compile-time construct implemented using *type erasure* [22]. After type checking code, the compiler effectively translates it into non-generic code by replacing type parameters with `Object` and inserting casts as required. This implementation relies on the fact that all types in Java (with the exception of the *primitive types* `int`, `char`, etc.) use a uniform representation—they are represented as a fixed-size reference to an object. This means that `String[]` (an array of `String`) and `Object[]` have the same layout in memory, can be operated on by the same code.

The decision to use type erasure was influenced by Java’s binary compatibility guarantees. With type erasure, the standard library (in particular, the collections library) could be retrofitted with generics to improve type safety and lift the burden of casts from the programmer, without changing the binary interface—after type erasure, the compiled library code is the same as in pre-generics Java.

Type erasure also simplified implementation, since only the compiler needs to be modified. Java code is compiled to bytecode, an intermediate form which is executed by the Java Virtual Machine (JVM). Neither the bytecode instruction set nor the virtual machine was affected by the addition of generics, since the compiler converts generic code to non-generic code using erasure.

A consequence of type erasure is that casts between generic types cannot be checked; when casting an object to `ArrayList<String>`, the runtime can check that the object is an instance of `ArrayList`, but it cannot check that the type argument is `String` because that information is not recorded at runtime.

Java’s implementation relies on uniform representation; the primitive types cannot be used directly as type parameters, since their representation is not compatible with the representation of reference types. The approach taken in Java to “box” the primitive types using wrapper classes roughly equivalent to the following:

```
class Integer { int value; }
```

The wrapper classes are reference types, so they can be used as type parameters. To pass a primitive value to a method that expects an `Object`, an instance of the wrapper class is constructed (“boxing”); to use the primitive value stored in an object, one must first cast it to the correct wrapper class, and then the value field can be accessed (“unboxing”). For convenience, the compiler automatically performs boxing and unboxing in most cases.

This approach is simple, but incurs a performance and memory penalty since a heap-allocated wrapper object is needed for each boxed primitive value. Caching the wrapper objects can mitigate this, but only to a small extent.

3.1.2 C#

C# is a programming language designed by Microsoft, strongly influenced by Java. C# has taken a different approach to generics [36].

Type information is kept at runtime, which allows casts between generic types to be checked. Like Java, C# had an extensive collections library before generics were added. The collections classes could not be transformed into generic versions in a compatible way; so instead, generic versions were added in a different namespace—the old collections classes continue to be available in the namespace `System.Collections`, while the new classes are in `System.Collections.Generic`. Gradual migration was more difficult with this approach, but it benefits modern code.

Generics in C# support non-uniform representation using runtime code generation (with code sharing for types with the same representation). This approach is not applicable to Go, because one of the goals of Go is to not require runtime code generation.¹

3.1.3 C++

In C++, support for generic classes and functions is provided through *templates*. The compiler *instantiates* templates by substituting the template arguments for the template parameters and compiling the resulting code. Templates are automatically instantiated when they are referenced.

¹<https://docs.google.com/document/pub?id=1IXHI5Jr9k4zDdmUhcZImH59bOUK0G325J1FY6hde1cM> Russ Cox. 2012. “Alternatives to Dynamic Code Generation in Go”.

A limitation of templates is that instantiation occurs at compilation time, so all uses of a template must be known statically. We discuss this topic in more detail in [Section 4.2](#). Templates also raise concerns of “code bloat”—large executables due to many instantiations.

C++ templates did not originally support a way to specify the requirements on the template arguments; rather, the permitted template arguments are those for which the corresponding template body compiles.

The lack of well-specified constraints often results in long and inscrutable error messages. As a particularly egregious example, the following 7-line program produces 1,373 lines of error messages in GCC 16.1:

```
#include <algorithm>
struct Person { char* name; };
int main() {
    Person arr[5];
    std::sort(arr, arr + 5);
    return 0;
}
```

The error in this program is that the type `Person` does not implement `operator<`, so it cannot be used with `std::sort`.

A relatively recent addition to C++ are *concepts*, which address this issue by allowing templates to specify explicit requirements on their parameters. C++ Concepts have a complicated history. An early version was planned for C++11 (then-named “C++0x”), but was withdrawn due to complexity and technical issues [72]. A revised version was added in C++20. The syntax for specifying requirements is based on giving example expressions that need to compile, with special syntax for specifying return types.

3.1.4 Haskell

Haskell is a purely functional programming language. It uses the Hindley–Milner type system for type inference, extended with *type classes* to provide overloading [89].

The implementation of generics (parametric polymorphism) in Haskell uses uniform representation. Haskell does not rely on run-time type information, and does not permit casts; instead, a generic function specifies the operations

required from its type parameters using *type classes* (roughly analogous to interfaces in Java), and *dictionary-passing* is used to pass the implementation of those operations to the generic function. The dictionaries roughly correspond to the virtual method tables of object-oriented languages, but they are stored separately from the object.

3.2 Terminology

Run-time type information (RTTI) makes information about the types of values available at runtime. This information is required for operations such as checked casts (which in Go are called *type assertions*). There are different approaches to handling this information; in Java for example, run-time type information (excluding erased information) is included in the representation of every object² (though there is work under way to change this³). In contrast, Go only records run-time type information when a value is converted to an *interface value* [12].

Uniform representation refers to values of values of different types having the same representations in memory. Uniform representation has the several advantages, including simplicity and support for polymorphism (both in the object-oriented sense and the parametric sense), but it has significant memory overhead. In Java, the compromise is that all *reference types* have uniform representation, while a small number of *primitive types* have non-uniform representation for efficiency. In contrast, languages from the C family (including Go) lean towards memory-efficient, non-uniform representations; for example, it is common to declare an array of elements such as struct values that are packed contiguously in memory, with no reference indirection.

Boxing. One of the challenges with non-uniform representation is to write code that can operate on types with different representations. For example in Java, generics is restricted to reference types; a generic function for reversing an array can accept `String[]`, but not `int[]` (which has a different layout in memory). A solution to allow code to operate on values with different representations is to “box” (wrap) the values into a uniform

Erasure. Some implementations of generics, notably Java, translate generic

²<https://openjdk.org/jeps/450> “JEP 450: Compact Object Headers”. Retrieved 7 July 2026.

³<https://openjdk.org/jeps/401> “JEP 401: Value Objects (Preview)”. Retrieved 7 July 2026.

code to non-generic code using uniform representation or boxing, and do not store run-time type information about type arguments. This approach has limitations; for example, casts to generic types cannot be fully checked.

Dictionary-passing [89] is a technique where a generic function is passed the implementation of operations defined on its type parameters.

Monomorphisation is the technique of translating a generic type/function into multiple non-generic (monomorphic) copies, one for each set of type arguments used throughout a program; C++ templates use monomorphisation. Monomorphisation has the advantage of producing efficient code, but the disadvantage of requiring all instantiations to be known statically. We use the term *specialisation* synonymously with monomorphisation.

First-class generic functions is a term that we will introduce in [Section 4.2](#), in the context of Go. It has connections to several ideas from type theory: higher-rank polymorphism, impredicative polymorphism, and polymorphic recursion.

Chapter 4

History of Generics in Go

The first open-source release of the Go programming language was in November 2009, and the first comment¹ noting the omission of generics was posted within 24 hours [82]. Go was released without generics due to the complexity involved and because the Go team initially did not consider generics essential to the language’s goals.²

In December 2009, Russ Cox of the Go team posted a short blog post titled “The Generic Dilemma” [11] describing the basic approaches for handling generics in a compiler and their drawbacks: (1) leave out generics, which slows programmers; (2) use compile-time specialisation as in C++, which slows compilation; (3) use boxing as in Java, which slows execution. Hence the dilemma: “Do you want slow programmers, slow compilers and bloated binaries, or slow execution times?”

Over time, the need for generics in Go became clear. The annual Go surveys showed that generics are a very important feature for users (as mentioned in Section 1.2). There were many instances where the lack of generics has been a problem; to give just two examples, new channel operations³ could only be added by changing the language rather than writing library functions [13], and members of the Go team grew disillusioned with workarounds such as the one used in the `sort` package.⁴

¹<https://groups.google.com/g/golang-nuts/c/70-pdwUUrBI/m/onMsQspcljCJ>
JE. 2009. The Go Nuts Mailing List.

²https://go.dev/doc/faq#beginning_generics “Frequently Asked Questions (FAQ)”.
The Go Documentation. Retrieved 7 July 2026.

³<https://github.com/golang/go/issues/26282> “proposal: spec: clone and splice, new channel primitives”.

⁴<https://github.com/golang/go/issues/16721> “sort: make sorting easier, add sort.Slice, ...”

After several rejected proposals (Section 4.3), generics were finally added in 2022 with the release of Go 1.18. The implementation can be characterised as compile-time specialisation where instantiations for type arguments of the same “shape” have partial code sharing; this strikes a balance between approaches (2) and (3) from [11].

4.1 Alternatives to generics

There were various ways to work around the absence of generics in Go, as listed for example in the community-maintained document “Summary of Go Generics Discussions”⁵, but they all fell short in some respect. We describe these approaches, their strengths, and their shortcomings.

4.1.1 Code duplication

The most primitive solution is to duplicate code and manually replace the types, as with `ReverseStrings` in Section 1.2.1).

This is the approach originally recommended for small tasks such as reversing a slice, removing an element from a slice, or copying a map.⁶

Advantages:

- Code that uses the specialisations can be statically type checked (it does not require casts).
- The specialisations will have good runtime performance.
- General-purpose (but see Section 4.2).

Disadvantages:

- The resulting code is difficult to maintain. If changes are made to the original function (such as bug fixes), the programmer must remember to update all the copies. This is highly error-prone.
- Code bloat. A complete copy of the code is required for each combination of type arguments used.

⁵<https://docs.google.com/document/d/1vrAy9gMpMoS3uaVphB32uVXX4pi-HnNjkMEgyAHX4N4/view> “Summary of Go Generics Discussions”. Google Docs. Retrieved 7 July 2026.

⁶<https://web.archive.org/web/20260530021722/https://go.dev/wiki/SliceTricks#reversing> “Go Wiki: SliceTricks”. Archived May 2026.

- Replacing types in copied code involves manual work, and the process can be fragile (for example, naive “find and replace” may inadvertently replace occurrences of a type that were not intended as type arguments).

4.1.2 Code generation

The Go toolchain includes support for code generation through the command `go generate`⁷ [67], which executes arbitrary commands (often scripts written in Go) specified using directives in source files. Code generation can be used to automate the code duplication approach—using string substitution, string templating⁸, or by parsing a source file containing a template (using the package `go/parser`⁹) and transforming its AST.

Advantages:

- The code is statically type checked (but the templates are typically not).
- Good runtime performance.
- General-purpose (but see Section 4.2).

Disadvantages:

- Dependencies between generic types require special handling. For example, if a type `HashSet[T]` is implemented using `HashMap[T, bool]`, then to generate `HashSet[string]` one must also generate `HashMap[string, bool]`.
- Code generation has a propensity for code bloat.
- The user must explicitly instantiate the template with each set of type arguments that will be used.
- There is poor interoperability across packages. For example, if two packages instantiate `HashMap[string, bool]` independently, then the generated types will be distinct and incompatible.
- Organisation of the generic code can be awkward. For example, if an entire template source file is instantiated as one unit, then generic functions with a different number of type parameters cannot exist the same template source file.

⁷https://pkg.go.dev/cmd/go#hdr-Generate_Go_files_by_processing_source Package `cmd/go`. The Go Standard Library. Retrieved 7 July 2026.

⁸<https://pkg.go.dev/text/template> Package `text/template`. The Go Standard Library.

⁹<https://pkg.go.dev/go/parser> Package `go/parser`. The Go Standard Library.

- Generated code is less pleasant to work with, since there is an additional step that the programmer must remember to perform between saving a file and running the code. (The `go generate` command is intentionally not executed automatically by `go build` [67].)
- Bespoke code generators can be fragile (in particular if they are string-based, but AST transformations can also require attention to detail).

4.1.3 `interface{}` and type assertions

Another approach is to replace type parameters with the empty interface, `interface{}`, which can hold values of any type. This is the approach used in the package `container/list`¹⁰ (a linked list implementation), the concurrent map from the package `sync`¹¹, and the package `context`¹² which carries deadlines and arbitrary key-value pairs. This approach generally requires the user of the library to insert *type assertions* (casts) to convert results to their actual type (for example when getting elements of the linked list).

Advantages:

- No code duplication.
- Easy to use (with the exception of the type assertions).

Disadvantages:

- The user is required to insert type assertions.
- There is no static type checking. For example, a user of `container/list` cannot ensure that a particular linked list contains only strings; and incorrect type assertions are only detected at runtime.
- Storing a value as `interface{}` adds a small memory overhead, and type assertions add a small performance overhead.
- This approach cannot handle composite types. For example, the type `[]int` cannot be converted directly to `[]interface{}`, because they have different representations in memory; conversion would require an expensive copy operation.

¹⁰<https://pkg.go.dev/container/list> Package `container/list`. The Go Standard Library.

¹¹<https://pkg.go.dev/sync> Package `sync`. The Go Standard Library.

¹²<https://pkg.go.dev/context> Package `context`. The Go Standard Library.

4.1.4 Reflection

Go supports runtime reflection through the package `reflect`¹³, which allows manipulation of objects whose types are not known statically. The example from [Section 1.2.1](#) could be written using reflection as follows:

```
import "reflect"

func ReverseUsingReflect(slice interface{}) interface{} {
    original := reflect.ValueOf(slice)
    if original.Kind() != reflect.Slice {
        panic("ReverseUsingReflect: non-slice type")
    }
    n := original.Len()
    reversed := reflect.MakeSlice(original.Type(), n, n)
    for i := 0; i < n; i++ {
        reversed.Index(n - i - 1).Set(original.Index(i))
    }
    return reversed.Interface()
}

// Usage:
var numbers []int = []int{1, 2, 3}
numbers = ReverseUsingReflect(numbers).([]int) // Type assertion
```

The main disadvantage of this approach is that it is very slow. The other disadvantages are that the code which uses the `reflect` package is much harder to read, and there is no static type checking.

In some cases it is possible to optimise code that uses reflection to be nearly as fast as normal code. For example, in 2016 the convenience function `sort.Slice` was added to the `sort` package;¹⁴ this function uses reflection instead of the adapter `sort.Interface`. To make `sort.Slice` efficient, a new function named `reflect.Swapper` was added to the `reflect` package, which creates an efficient “swapper” function for a given slice. This way the overhead of reflection is only incurred once per call to `sort.Slice` rather than per swap.

The `reflect` package has almost-complete support for the operations available in Go. Some operations, such as iteration over a map (with the same semantics as Go’s `for range` loop), were not available initially but were added

¹³<https://pkg.go.dev/reflect> Package reflect. The Go Standard Library.

¹⁴<https://github.com/golang/go/issues/16721> “sort: make sorting easier, add sort.Slice, ...”

later.¹⁵ Other operations, such as creation of struct types with promoted methods from embedded fields, remain unsupported.^{16,17,18,19}

The limitations of the `reflect` package affect^{20,21} development of Go interpreters such as `gomacro`²² and `Yaegi`²³. There is an interesting connection between a generics and interpreters; execution of generic code can be viewed as execution of code where some types are only known at runtime, while for an interpreter the *entire* code is only known at runtime. The techniques used in Go interpreters could therefore be used to implement generics, but they will generally be less efficient because they target a more general problem and cannot take advantage of information available at compilation time.

The disadvantages of reflection mean that it has a lower preference and that it requires great care when performance is a concern.

Advantages:

- General-purpose (with some limitations).

Disadvantages:

- Very slow (though in some cases it can be optimised).
- There is no static type checking.
- Code that uses the `reflect` package is much harder to read.

A closely related technique is the use of the `unsafe` package²⁴, which allows raw memory access. Operations performed using the `unsafe` package are much faster than the `reflect` package, but there are no type safety checks and is difficult to use correctly (for example due to the interaction with the garbage collector).

¹⁵<https://github.com/golang/go/issues/11104#issuecomment-262624401> “reflect: add a range like interface for Value”.

¹⁶<https://github.com/golang/go/issues/15924> “reflect: StructOf doesn’t generate wrapper methods for embedded fields”.

¹⁷<https://github.com/golang/go/issues/61433> “reflect: reflect.StructOf makes types that violate reflect.Type contracts”.

¹⁸<https://github.com/golang/go/issues/4146> “reflect: MakeInterface”.

¹⁹<https://github.com/golang/go/issues/16522> “reflect: NamedOf”.

²⁰<https://github.com/golang/go/issues/39574> “proposal: reflect: add method Type.Underlying()”.

²¹<https://github.com/golang/go/issues/39717> “proposal: reflect: allow creation of recursive types atomically”.

²²<https://github.com/cosmos72/gomacro> “gomacro - interactive Go interpreter and debugger with generics and macros”. GitHub repository. Retrieved 7 July 2026.

²³<https://github.com/traefik/yaegi> “Yaegi is Another Elegant Go Interpreter”. GitHub repository. Retrieved 7 July 2026.

²⁴<https://pkg.go.dev/unsafe> Package `unsafe`. The Go Standard Library.

4.1.5 Adapters

In some cases it is possible to write “pseudo-generic” functions that process a composite type without knowledge of the element type, by declaring an adapter interface with the required operations and having the caller pass an implementation of this interface.

For example, the package `sort`²⁵ is written so that it only requires three operations from the collection to be sorted: get the length, compare the elements at two indexes, and swap the elements at two indexes. The signatures of these operations do not depend on the element type, so they can be accurately described using a normal Go interface. This interface is declared in the `sort` package as follows:

```
// An implementation of Interface can be sorted by the routines
// in this package. The methods refer to elements of the
// underlying collection by integer index.
type Interface interface {
    // Len is the number of elements in the collection.
    Len() int
    // Less reports whether the element with index i
    // must sort before the element with index j. [...]
    Less(i, j int) bool
    // Swap swaps the elements with indexes i and j.
    Swap(i, j int)
}
```

To sort a slice such as `[]int`, the caller declares an adapter type that wraps `[]int` and implements `sort.Interface`, such as the following (in fact this specific declaration is included in the `sort` package for convenience):

```
type IntSlice []int
func (x IntSlice) Len() int           { return len(x) }
func (x IntSlice) Less(i, j int) bool { return x[i] < x[j] }
func (x IntSlice) Swap(i, j int)      { x[i], x[j] = x[j], x[i] }
```

Then a value `s` of type `[]int` can be sorted by calling `sort.Sort(IntSlice(s))`.

An interesting consequence in this particular example is that the sorting implementation must be in-place, because the adapter does not allow copying elements to a temporary location. For stable sorting this is a challenge, since

²⁵<https://pkg.go.dev/sort> Package `sort`. The Go Standard Library.

classic algorithms such as merge sort are not in-place. One option (which requires $O(n)$ additional space) is to create a slice with the indexes, sort the indexes using a classic out-of-place stable sorting algorithm,²⁶ then permute the original elements in-place according to the sorted indexes. The option chosen by the Go team is to use the in-place merge sort algorithm from [39], which only requires $O(\log n)$ additional space but performs $O(n \log^2 n)$ swaps.

The adapter approach is not general-purpose. There are cases where it is suitable but requires `interface{}` and type assertions; for example the package `container/heap`²⁷ (which implements a heap data structure) extends the adapter interface from the `sort` package with operations `Push` and `Pop`, which require the use of `interface{}`. There are also many cases where the adapter approach is simply unsuitable, for example because the adapter implementation would be so complex that it is not worthwhile.

Advantages:

- The code is statically type checked.
- The performance cost (from the overhead of interface method calls) is fairly low.

Disadvantages:

- This approach is not general-purpose, and there are cases where `interface{}` and type assertions are required.
- The adapter implementations will generally be boilerplate code.
- The limited interface exposed by the adapter may prevent algorithmic improvements (for example, rotating an array of size n by one position using swaps requires $n - 1$ swaps and therefore $3 \times (n - 1)$ assignments, while rotating using assignments and a temporary variable requires only $n + 1$ assignments).
- The limited interface exposed by the adapter may require the code that uses the adapter to be written in an awkward style, harming readability.

²⁶Or using an unstable sorting algorithm with a comparison operation that breaks ties as in Section 1.2.2.

²⁷<https://pkg.go.dev/container/heap> Package `container/heap`. The Go Standard Library.

4.2 First-class generic functions

We now take an aside to introduce a feature we refer to as *first-class generic functions* (by analogy with the terminology from [31]). As the name suggests, this feature makes generic functions first-class values; it would allow the generic function `Reverse` from [Section 1.2.1](#) be used as a value whose type might hypothetically be written as `func[T any] ([]T) []T`, a *generic function type*.

This feature is difficult to implement; of the general-purpose approaches mentioned so far, those that support static type checking (namely, code duplication and code generation) do not support first-class generic functions. The accepted type parameters proposal [85] and the version implemented in the Go compiler also do not support first-class generic functions, because they rely on specialisation.

If supported, first-class generic functions would have several important implications. In type theory terminology, they imply support for *higher-rank polymorphism* [37, 65] (also known as *rank- N types*), which in the context of Go means that generic function types may be used as the types of function parameters; they also imply support for *impredicative polymorphism* [65] (also known as *first-class polymorphism*), which in the context of Go means generic function types may be used as type arguments of generic functions and of generic types; finally, first-class generic functions imply support for *polymorphic recursion* [30], which allows a generic function to call itself with different type arguments.

First-class generic functions also have an important implication to interfaces in Go. They are closely related to generic interface methods—each can be expressed in terms of the other.

To give a practical example of the usefulness of first-class functions, suppose we have several sorting functions:

```
func StableSort[E any](elements []E) []E { ... }
func QuickSort[E any](elements []E) []E { ... }
// The 'cmp' parameter is omitted for simplicity
```

and suppose we wish to run each of these functions on several test cases.

If all the test cases are slices of the same type, this can be done easily:

```

func TestInts(sort func([]int) []int) {
    assertEquals(sort([]int{3, 2, 1}), []int{1, 2, 3})
    assertEquals(sort([]int{2, 1, 3}), []int{1, 2, 3})
    ...
}
// Usage:
TestInts(StableSort[int])
TestInts(QuickSort[int])

```

(The implementation of generics in Go allows `TestInts(StableSort[int])` to be written as `TestInts(StableSort)`, with the type argument inferred, but we write the type argument explicitly for clarity.)

However, if we wish to write a single testing function that includes test cases with different slice types, we would need first-class generic functions:

```

func TestMix(sort func[T any]([]T) []T) { // Hypothetical syntax
    assertEquals(sort[int]([]int{3, 2, 1}), []int{1, 2, 3})
    assertEquals(sort[string]([]string{"z", "a"}), []string{"a", "z"})
    ...
}
// Usage:
TestInts(StableSort)
TestInts(QuickSort)

```

(In the first call to `sort` we use an explicit `int` type argument for clarity, while in the second call we demonstrate type inference.)

In situations where the set of generic functions to be used is fixed, a workaround would be to assign each of them an ID and use this ID to specify which generic function to use; the function `TestMix` would take an ID rather than a generic function, and it would call the corresponding function using a helper such as the following:

```

func CallSortFuncById[E any](funcId int, elements []E) []E {
    switch funcId {
    case 0: return StableSort(elements)
    case 1: return QuickSort(elements)
    ...
    }
}

```

This workaround is not general-purpose: it requires all the generic functions that might be used to be available and known by the calling code in advance, so

it cannot be used to pass a user-declared generic function to a library function or to implement interfaces with generic methods.

Comparing to some other programming languages,

- C++ does not support first-class generic functions (since templates are instantiated at compile-time);
- C# supports generic interface methods;
- Java supports generic interface methods trivially (due to boxing and type erasure);
- Haskell supports typeclasses with polymorphic operations, which is roughly analogous; but the GHC language extensions “RankNTypes” and “ImpredicativeTypes” are required to allow polymorphic types to be used everywhere.

In [Section 4.6](#) we sketch an approach for implementing first-class generic functions in Go, based on dictionary-passing.

4.3 Proposals

Changes to the Go language are made through a proposal process.²⁸ A proposal can be made by a member of the Go team or a member of the community; the proposal and community feedback are reviewed by a committee of Go team members who decide whether to accept or reject the proposal.

The notable proposals for generics were written by Ian Lance Taylor, a member of the Go team. A series of rejected proposals culminated in the accepted proposal (coauthored with Robert Griesemer, also of the Go team). We briefly describe these proposals; a good overview is given in [13].

A design document [79] describes the reasons for adding generics, and the desired features: generic types and functions, constraints on type parameters, and implicit instantiation, without sacrificing too much compilation time and execution time.

The first proposal [77] focused on generic types, with an unusual syntax for generic functions, to be implemented in a similar way to the `reflect` package. It includes support for constraints on type parameters, specified using interfaces.

²⁸<https://go.goglesource.com/proposal/> “Proposing Changes to Go”. Go Project Design Documents repository.

The second proposal [78] uses the syntax `gen [T]` as a prefix for declarations of generic types and functions, and the syntax `Vector[int]` for instantiation. It includes similar constraints on type parameters, and explicitly notes that this does not allow constraints to specify operators (such as `+` and `<`),²⁹ but mentions that this could be solved by having the compiler automatically generate corresponding methods. The proposal describes a very basic type inference algorithm. The implementation described in the proposal again based on reflection (but function specialisation is briefly mentioned as an optional optimisation—characterised as “a form of cross-package inlining”). The proposal mentions some of the details entailed by a reflection-based implementation. For example, local variables whose size is only known at runtime require dynamic memory allocation (rather than a stack frame with a fixed layout), and struct field access and slice access require the computation of offsets based on the sizes of types. Run-time type information is explicitly passed to generic functions to facilitate this.

The second proposal also mentions some of the rejected syntax ideas; in particular, the Java syntax `List<String>` was rejected because it causes ambiguities in Go. Various examples of such ambiguities have been given across the proposals [78, 84, 85]; an example which we think is most compelling is that the function call `f(HashMap<K, V>(x))` could be parsed either as having one argument which is the result of converting `x` to the generic type `HashMap<K, V>`, or as having two `bool` arguments—the first is `HashMap<K` and the second is `V>(x)`. In Java, this ambiguity is avoided by having parentheses around the type in casts (unlike Go, where the parentheses are around the argument in type conversions); and in the case of method calls with explicit type arguments, by placing the type arguments *before* the method name, as in `Collections.<String>emptyList()`. Even so, the syntax creates parsing challenges, requiring arbitrary lookahead to distinguish between generic types and expressions [24, p. 729]. Similar approaches could probably be used in Go, but they are unappealing.

The third proposal [80] is an adaptation of the second proposal intended to address flaws. It switches to specialisation in response to concerns regarding slow execution time. Only the operations that involve type parameters are

²⁹Go does not support operator overloading. Most operators only support built-in types, with the exception of `==` and `!=` (which support struct types and interface types).

specialised—the rest of the code in the body of a generic function is shared, and it invokes the specialised operations through a dictionary passed by the caller. In an attempt to allow operators to be used with type parameters, the proposal specifies that constraints are computed implicitly based on the body of a generic function. For example, if a generic function with type parameter `T` attempts to use the `+` operator on a variable `x` of type `T`, then `T` is constrained to numeric and string types; and if the generic function attempts to call a method of `x` with some arguments, then `T` is constrained to types that have a method of that name that can be called with those argument types.

Implicit constraints are expressive but very problematic—they result in poorly-specified interfaces and are the reason that C++ templates produce inscrutable error messages. Further, as pointed out in [13], there is no way to ensure that the constraints are not changed accidentally; this is especially important in Go due to the backward compatibility promise.

The fourth proposal [81] retains implicit constraints and adds more implementation details. It changes the syntax to `type [T] List struct { ... }`; the proposal notes that `type List[T] struct { ... }` would be nicer but is ambiguous with an array of size `T`. This ambiguity was later resolved in [85] by introducing a compulsory constraint after the type parameter, as in `type List[T any] struct { ... }`. The proposed implementation is again based on specialisation with code sharing; the proposal notes that real compilers are likely to use full specialisation (without code sharing) as an optimisation in some cases.

The fifth proposal [83, 84] uses explicit type parameter constraints, specified using a system named “contracts”. Initially [83] contracts were specified using a system similar to the C++0x Concepts (which were removed in 2009 [72]). A contract is a code snippet with one or more type parameters that demonstrates the required operations by example; the first contract shown in the proposal is the following:

```
contract stringer(x T) {
    var s string = x.String()
}
```

This specifies that `T` must have a method called `String` that takes no arguments and returns `string`. Explicit constraints solve the issues of implicit constraints,

but the initial contract system holds several subtle surprises:

(A) Contracts introduce covariance and contravariance to Go’s type system.

Consider the following contract:

```
contract readerLike(x T) {
    var p []byte
    var n int
    var e error
    n, e = x.Read(p)
}
```

This contract is similar to Go’s `io.Reader` interface³⁰, but it also permits methods with signatures such as `Read([]byte) (int, *net.DNSError)` and `Read(interface{}) (int, error)`.

Covariance and contravariance are useful language features, but adding them incidentally is liable to cause confusion. For example, a naive user might assume that if a type satisfies `readerLike` it must implement `io.Reader`, but that is not the case (because interfaces do not permit covariant return types or contravariant parameter types, even with the contracts proposal). Instead, to specify that `T` should implement `io.Reader`, the user must write `io.Reader(x)` in the contract body.

(B) The type checking rules are unusual and subtle.

The standard problem of type checking can be described as “Given type information, determine whether an expression is valid.” This problem is familiar and well-understood.

In contrast, the problem of type checking with contracts would be described as “Assuming some expression (the contract body) is valid, determine whether another expression is valid.” This is an unusual problem; it effectively requires the standard type checking rules to be applied *in reverse* to the contract body to compute all the constraints it entails, so that they can be used during type checking.

As such, writing contracts requires great care and a thorough understanding of Go’s typing checking rules. The proposal mentions that the simplest way is to copy the function body into the contract, and notes that this becomes equivalent to C++ templates and provides no benefit over implicit constraints. It is possible to construct a finer-grained contract through trial and error by

³⁰<https://pkg.go.dev/io#Reader> Package io. The Go Standard Library.

adding requirements until the function that uses the contract compiles, but there are two risks with this approach: (1) the contract might be too restrictive, forbidding desirable type arguments; and (2) the contract might be too permissive—even though it is sufficient for the current function implementation, a future implementation may have slightly different requirements.

We now give a concrete example of the subtlety and potential for unforeseen consequences. Careful reading of the Go 1.12 language specification shows that the expression `0 << x` is valid if and only if `x` has unsigned integer type.³¹ So one might write `contract unsigned(x T) { 0 << x }`. However, in Go 1.13 the language specification was changed to allow signed integers as shift counts for convenience [26]. On its own, that language change was fully backward-compatible. But with the contracts proposal, that change would have been problematic—there might be contracts that rely on the old behaviour to restrict a type to be an unsigned integer. It is unclear whether signed shifts would have been considered a breaking change in this hypothetical (since technically all existing code would have continued to work), but it certainly would have broken the integrity of contracts that rely on the old behaviour.

A better way to ensure that a type is unsigned would be to use `contract unsigned(x T) { uint64(^T(0)) }`. This works because `^T(0)` is the bitwise complement of 0, and if `T` is signed the result is the constant -1 which is not representable by `uint64`.

The initial version of the fifth proposal explains that it does not include built-in contracts such as `contracts.Arithmetic` because one of the driving goals of the design is to not introduce new names, but that standard names may be introduced if people find them useful.

The fifth proposal was later updated [84]. The syntax and semantics of contracts were significantly changed, resembling interfaces. The updated contract syntax allows a finite set of permissible types to be listed explicitly; this mostly solves the issue of specifying operators, because most of Go’s operators may only be used with built-in types (which can be listed explicitly). The update also added a built-in contract `comparable` (for the `==` and `!=` operators) and a new package `contracts` that would contain standard contracts such as `Ordered`. The idea of explicitly listing permissible types, and the built-in

³¹Assuming `x` is not an untyped constant.

constraints `comparable` and `Ordered`, would carry through to the final accepted proposal.

Both versions of the fifth proposal use parentheses for type parameters. This results in several annoying ambiguities that require additional parentheses to resolve, including two ambiguities that we reported [46] through the feedback submission process³². For example, the declaration `type S struct { List(int) }` is parsed as having a field named `List` of type `int`; we suggested a small language change to allow embedding the type `List(int)` by writing `(List(int))`. A similar ambiguity occurs with interfaces. The proposal was later updated to mention both of these ambiguities.^{33,34}

The additional parentheses required to resolve these ambiguities are awkward, and as stated in [85] some people disliked the number of parentheses in declarations such as `func F(type T)(v T) (r1, r2 T)`, where the parentheses denote type parameters, regular parameters, and multiple return values.

The sixth proposal [85] is the proposal that was accepted. The main change is that the contract system was replaced with interface-based constraints, extended to support explicitly listing permissible types. There is no loss of expressive power compared to contracts [85], however some examples (with multiple interrelated type parameters) are slightly more verbose. The syntax uses square brackets, for example `type List[T any] struct { ... }`. Each type parameter must have a type constraint to avoid a syntax ambiguity; the proposal introduces the predeclared identifier `any` as a short alias for the empty interface, `interface{}`, for use in the common case where there are no constraints on the type parameter. The proposal does not specify a particular implementation, but broadly assumes a specialisation-based implementation.³⁵ The proposal does not allow methods to have type parameters (beyond the type parameters of the receiver type) due to the implementation challenge,

³²<https://go.dev/wiki/Go2GenericsFeedback> “Go Wiki: Go 2 Generics Feedback”. Retrieved 7 July 2026.

³³<https://go-review.googlesource.com/c/proposal/+/188358>. Ian Lance Taylor. Aug 2019. “design: minor updates to contracts design draft”. Go Project Design Documents repository change.

³⁴<https://go-review.googlesource.com/c/proposal/+/221257>. Robert Griesemer. Feb 2020. “design/go2draft-contracts.md: document struct embedding parsing ambiguity”. Go Project Design Documents repository change.

³⁵The section “No parameterized methods” briefly mentions runtime instantiation using just-in-time compilation or a reflection-based approach, but dismisses them as complex and slow.

as discussed in [Section 4.2](#); later a small language proposal³⁶ was accepted to allow concrete method declarations to have type parameters, but interface methods are not permitted to have type parameters (because that is what leads to the implementation challenge).

4.4 Implementation

The Go team prepared three implementation proposals for generics: “Stenciling”³⁷, “Dictionaries”³⁸, and “GC Shape Stenciling”³⁹. All three implementations are based on compile-time specialisation, with varying degrees of code sharing. The third is a hybrid of the first two, and is the implementation that was chosen for Go 1.18.⁴⁰

The stenciling proposal uses straightforward monomorphisation, with no code sharing between function instantiations with different type arguments.

The dictionaries proposal shares the code that does not depend on the type parameters. It is quite different from “pure” dictionary-passing; in traditional dictionary-passing, the dictionaries depend only on the signature of the function (specifically, on the constraints specified for its type parameters); in contrast, the Go dictionaries proposal generates a specialised dictionary for each function based on its body: each operation performed by the function that depends on a type parameter is turned into a helper method, supplied by the caller through the dictionary; the remaining code is shared across all instantiations. As a consequence, the dictionaries proposal does not support first-class generic functions—all instantiations must be known statically.

The GC shape stenciling proposal is a hybrid of the stenciling proposal and the dictionaries proposal. It creates separate instantiations for type

³⁶<https://github.com/golang/go/issues/77273> “spec: generic methods for Go”.

³⁷<https://go.dev/proposal/+217dd4e7ebc3beca/design/generics-implementation-stenciling.md> “Generics implementation - Stenciling”. Go Project Design Documents repository.

³⁸<https://go.dev/proposal/+217dd4e7ebc3beca/design/generics-implementation-dictionaries.md> “Generics implementation - Dictionaries”. Go Project Design Documents repository.

³⁹<https://go.dev/proposal/+88faf0a677867d37/design/generics-implementation-gcshape.md> “Generics implementation - GC Shape Stenciling”. Go Project Design Documents repository.

⁴⁰<https://go.dev/proposal/+30e9a8d11d1e84aa/design/generics-implementation-dictionaries-go1.18.md> “Go 1.18 Implementation of Generics via Dictionaries and Gcshape Stenciling”. Go Project Design Documents repository.

arguments of different “shapes” (size and alignment),⁴¹ and passes dictionaries for operations that depend on the type parameters. The GC shape stenciling proposal avoids much of the complexity of the dictionaries proposal, because the sizes of the type parameters are known statically—the implementation does not need to deal with variable-sized stack frames.

The implementation of generics in Go 1.18 was well-received by the community, despite the limitations, as evidenced by the Go Developer Survey.⁴²

4.5 Literature on generics in Go

There have been a number of academic papers about generics in Go, including in collaboration with the Go team.

Featherweight Go. Inspired by previous work on Featherweight Java, Griesemer et al. [27] formalise a subset⁴³ of Go and extend it with generics (Featherweight Generic Go) using monomorphisation. They prove preservation and progress (type safety) for the reduction rules, and bisimulation (correspondence) for a their monomorphisation translation.

One of the contributions of Griesemer et al. is a predicate to detect unbounded recursive instantiation (which is not supported by monomorphisation). Go’s released version of generics includes an implementation of this check; the source code of the compiler⁴⁴ includes good documentation. The check permits recursive instantiations that use the same type arguments (for example `type List[T any] struct { head *T; tail *List[T] }`), and it permits recursive instantiations that use a permutation of the type arguments (for example `type Map[K, V any] struct { inverse *Map[V, K] }`), but it forbids examples such as the following (from [60, §10.1]):

```
type CompleteTree[T any] struct {
    value *T
    child *CompleteTree[Pair[T, T]]
}
```

⁴¹The updated description in the Go 1.18 design document limits sharing to type arguments that have the same underlying type and to pointer types.

⁴²<https://go.dev/blog/survey2022-q2-results> “Go Developer Survey 2022 Q2 Results”.

⁴³Local types and embedded struct fields are not included for example.

⁴⁴<https://go.golangsource.com/go/+/refs/tags/go1.26.5/src/cmd/compile/internal/types2/mono.go> The Go Programming Language compiler.

This example uses polymorphic recursion with a composite type argument `Pair[T, T]` that references a type parameter. This example is forbidden because monomorphisation would produce an infinite number of instantiations—`CompleteTree[int]` instantiates `CompleteTree[Pair[int, int]]`, which in turn instantiates `CompleteTree[Pair[Pair[int, int], Pair[int, int]]]`, and so on ad infinitum.

The Go compiler reduces the check for unbounded recursive instantiation to the problem of finding a positive-weight cycle in a directed graph. The graph’s nodes are the type parameters; a zero-weight edge $T_1 \longrightarrow T_2$ is added if T_1 is used as a type argument for T_2 , and an edge $T_1 \longrightarrow T_2$ of weight 1 is added if a composite type that references T_1 is used as a type argument for T_2 . Unbounded recursive instantiation can occur if and only if the graph contains a positive-weight cycle. The Go compiler uses a variant of the Bellman–Ford algorithm⁴⁵ to check for positive-weight cycles in $O(V \times E)$, where V is the number of nodes and E is the number of edges. The implementation in the Featherweight Generic Go prototype⁴⁶ is similarly $O(n^2)$.

It is possible to improve this algorithm to $O(n)$. The Bellman–Ford algorithm supports both positive-weight and negative-weight edges, but here there are only positive-weight and zero-weight edges. We can use Tarjan’s algorithm [76] to identify strongly-connected components (SCCs) in $O(V + E)$, and then for each positive-weight edge $T_1 \longrightarrow T_2$ check whether T_1 and T_2 are in the same SCC; if so, that edge is part of a positive-weight cycle. A further improvement is to perform the checks during the execution of Tarjan’s algorithm—once Tarjan’s algorithm finishes processing an edge, it is known whether that edge is part of an SCC. We have proposed using this improved algorithm in the Go compiler.⁴⁷

Generic Go to Go. Ellis et al. [17] propose using dictionary-passing and boxing to support generic interface methods, and prove their translation from Featherweight Generic Go to Featherweight Go is correct. They box *all* method parameters and struct fields as `any`; this is not a suitable implementation strategy for the Go compiler due to the overhead (which would also impact

⁴⁵The Bellman–Ford algorithm computes the single-source shortest paths and can be extended to detect negative-weight cycles (or positive-weight cycles, by negating the weights).

⁴⁶https://github.com/rhu1/fgg/blob/5b86f091eff3f688/internal/fgg/fgg_ismonom.go#L44

⁴⁷<https://github.com/golang/go/issues/80378> Tom Levy. 2026. “cmd/compile: improve mono.go from $O(n^2)$ to $O(n)$ ”. GitHub issue.

non-generic code).

In our opinion dictionary-passing is the right approach for adding generic interface methods to Go, but we found that the specific translation scheme proposed by Ellis et al. is incompatible with the implementation of generic struct types in Go 1.18 because it adds dictionaries as struct fields. The issue requires some explanation.

The Go language specification permits conversion between pointer types whose element types have the same underlying type [20, §Conversions]. For example, the types `type S1 struct { X int }` and `type S2 struct { X int }` have the same underlying type, and so a value of type `*S1` may be converted to a value of type `*S2`. In the Go 1.18 implementation of generics, the underlying type of the instantiation of a generic struct type is the result of substituting the type arguments in place of the type parameters (as a natural consequence of monomorphisation). Hence the following program is valid in Go 1.18:

```
package main
type S struct { X int }
type G[T any] struct { X T }
func main() { _ = (*G[int])(amp{42}) } // Convert *S to *G[int]
```

The Go compatibility promise mandates that this remains valid in all future versions of Go—so for better or worse, the type `G[int]` *must* have the same representation in memory as `S`. In particular, fields may not be added to `G[int]` to store dictionaries or run-time type information.

Fortunately, it is possible to implement dictionary-passing while remaining compatible with Go 1.18. Sulzmann and Wehr [74] describe a dictionary-passing translation from Featherweight Generic Go to the untyped lambda calculus (effectively boxing all values). They pass the dictionaries during method calls, without storing them as struct fields; this is compatible with Go 1.18. (They actually pass *coercion functions* rather than dictionaries, but the result is equivalent.)

Welterweight Go. Hu et al. [33] continue their work from Featherweight Go with an extended model of Go and propose using boxing and adapter functions to implement generic interface methods. The level of detail in the formalisation and the proofs is impressive, but we believe there are issues that make the compilation strategy unsuitable for Go:

- Struct fields that reference type parameters are boxed [33, §5.1]; this is incompatible with the Go 1.18 implementation of generics for the same reason as above.
- Boxing cannot support composite types such as slices, for the reason described in [Section 4.1.3](#).
- The treatment of interface values appears to be slightly different from standard Go in that the size of their method table is not fixed.

We have reported our concerns, but we did not receive a reply in time for inclusion in this thesis.⁴⁸

4.6 Future

The addition of generics in Go 1.18 has been a tremendous improvement—the limitations of the monomorphisation implementation are minor in comparison. Still, as we look forward to the future, we see room for improvement.

In our opinion, a good next step in the evolution of generics in Go would be to add support for first-class generic functions and generic interface methods, with the added benefit of allowing the size of generated code to be reduced. Hu et al. [33] show that extending (a subset of) Go with generic interface methods is sound; we feel that the main question that needs to be addressed is implementation.

The constraints of Go—in particular non-uniform representation, backward compatibility, and no runtime code generation—rule out many approaches; but we believe there is one natural path forward: dictionary-passing with support for non-uniform representation. We sketch this approach and illustrate it with an example.

We consider functions and interfaces separately. Featherweight Go uses a subset of Go that does not include functions (instead functions are simulated using methods of trivial types); this reduces the size of the language without loss of expressiveness, and that is very useful in some contexts; but it couples together functions and interfaces, and we feel that this complicates the analysis

⁴⁸<https://github.com/rhu1/wg/issues/1> Tom Levy. 2026. “Questions about the paper and representations”. GitHub issue.

of first-class generic functions. It also strays from the actual implementation of Go, which supports functions as first-class values.

Non-uniform representation. We observe that non-uniform representation is an inherent part of Go, both due to a strong culture of preferring memory-efficient representations over boxing and (more concretely) due to the backward compatibility constraints imposed by the Go 1.18 implementation of generics. Fortunately, Go supports raw memory access and has a powerful reflection capabilities; these make it possible to operate on non-uniform representations. The challenges are performance and completeness (specifically, due to known limitations of the `reflect` package such as those mentioned in [Section 4.1.4](#); they would need to be addressed). Reflection in Go is very slow, but we are optimistic that acceptable performance can be achieved:

- Our approach is intended to *complement* monomorphisation, not replace it; the compiler may choose between the two (for example using profile-guided optimisation⁴⁹).
- Many operations (such as variable assignment and slice access) only require basic memory access, such as with `memmove`; this is still slower than operations on known types, but has good potential for acceptable performance.
- Generic code is checked at compile-time, so runtime type checks can be eliminated.
- The implementation of `sort.Slice`⁵⁰ shows that reflection can be performant.

Many of the details discussed in the Go dictionaries implementation proposal are relevant (for example stack frame layout, struct field access, and slice access). However, that proposal does not support first-class generic functions because it derives the dictionaries from the body of the generic function; first-class generic functions require a “pure” dictionary-passing approach where the dictionaries are computed solely based on the signature of a generic function. We do not discuss the details of implementing function bodies further; instead we focus on dictionary-passing, beginning with the representation of functions.

⁴⁹<https://go.dev/doc/pgo> “Profile-guided optimization”. The Go Programming Language documentation. Retrieved 7 July 2026.

⁵⁰<https://github.com/golang/go/issues/16721> “sort: make sorting easier, add sort.Slice, ...”

Functions. We have two requirements for the representation of functions: calls from non-generic code to non-generic functions should be efficient, and calls involving generic code should be possible (and reasonably efficient).

We propose to represent a Go function as a pair of function pointers—a “native” (unboxed) version that uses the usual calling conventions, and a version where *all* parameters and return values are boxed as plain pointers (with no run-time type information). This is analogous to the adapter functions of [33].

The caller uses either the native version or the boxed version depending on whether the function signature (as seen by the caller) references type parameters.

In general, one of the two function pointers will be a wrapper for the other;⁵¹ which of the two is the wrapper depends on whether the signature in the function’s declaration references type parameters. If the signature does not reference type parameters, the boxed version is the wrapper; it simply unboxes its arguments, calls the native version, and boxes the return values. If the signature references type parameters,⁵² the native version is the wrapper; this is the more difficult case. Before we describe our approach, we must discuss the representation of closures.

A *function value* is represented in Go as a *closure object*, which is a function pointer followed by a variable-sized *environment*.⁵³ The calling convention is that a pointer to the closure object is passed to the function pointer as an additional argument. It is natural to change this representation to contain the pair of function pointers (native and boxed) followed by the environment. This also allows wrappers for the same (or compatible) signatures to be shared—the wrapper has access to the other function pointer through the closure object and can forward calls to it, so the body of the wrapper only depends on the function signature (specifically, the layout of the parameters and the return values).

The standard approach for creating a function value with a signature that is

⁵¹It is also possible for both function pointers to be wrappers for a third function which uses a mixture of boxed and unboxed parameters (but note that this reduces the opportunities for wrapper sharing). Likewise, direct calls can use a mixture of boxed and unboxed parameters for efficiency.

⁵²For example a generic function, or a local function that references the type parameters of an enclosing generic function.

⁵³<https://go.golangsource.com/go/+/refs/tags/go1.26.5/src/cmd/compile/abi-internal.md#closures> “Go internal ABI specification”. The Go Programming Language documentation.

not known statically is to use `reflect.MakeFunc`⁵⁴. This is relatively slow, and calls are also relatively slow (since the function must read the arguments from the stack frame based on dynamic type information). It also requires some wrangling because `reflect.MakeFunc` returns a closure object rather than a function pointer.

Another approach we reject is to make the native function pointer optional, and have the caller fall back to the boxed function pointer if the native function pointer is nil. The issue is that this significantly increases the size of the generated code for all non-generic calls, due to the boxing and unboxing required in the fallback path.

The approach that we propose is to statically generate all the native wrappers needed by function values. This is possible because all the signatures involved in *calls* to native versions are known statically. There are a few different options for looking up these statically-generated native wrappers:

- Build a static table that maps each known signature to a native wrapper, and look up the native wrapper using the dynamically-computed signature when creating a function value. (If there is no statically-generated native wrapper for the signature, the native function pointer is set to nil—it is guaranteed that all calls will use the boxed function pointer.)
- Lazily set the native function pointer to the native wrapper during the first call. The caller checks whether the native function pointer is nil, and if so sets it to the statically-known native wrapper. This adds an overhead of a few instructions to all non-generic function calls.
- Build a static table that maps each call instruction in the program that calls a native version of a function value to the corresponding native wrapper. Initialise the native function pointers to a special assembly subroutine that looks up the real native wrapper based on the caller's program counter, sets the native function pointer to the real native wrapper to speed up future calls, and forwards the call to the native wrapper.

Generic functions take additional *dictionary* parameters corresponding to each of their type parameters.⁵⁵ We describe these dictionaries next.

⁵⁴<https://pkg.go.dev/reflect#MakeFunc> Package `reflect`. The Go Standard Library.

⁵⁵If a generic function is instantiated but not called immediately, it is handled using partial application.

Dictionaries. We propose extending the runtime representations used in Go to support dictionary-passing.

Go represents *interface values* as a pair of pointers [12].⁵⁶ The first points to an *itable* (interface table) that contains run-time type information and an array of function pointers (virtual method table) corresponding to the methods specified in the interface. The second pointer holds the data for the value—in the case of a struct, that would be the values of its fields.

We propose reusing the itable data structure to implement dictionary-passing. A dictionary can be viewed as a *detached itable* [89]; it allows methods specified in the constraint of a type parameter to be called, including on elements of a composite type such as a slice (without having to box each element individually).

The runtime representation of a type is extended with an array of itables, one per type argument, with method tables corresponding to the type parameter constraints.

To support non-uniform representation, the runtime representation of generic struct types is extended with an array of field offsets.

Methods. Methods are handled similarly to function values, but with some differences because methods of generic types need access to the receiver’s type information; also, methods are represented in Go as function pointers with no access to a closure environment.

The entries in the method table are changed to use the two-pointer function representation, and to take a pointer to the enclosing itable as an additional parameter (providing access the receiver’s type information and to the method table). This additional parameter adds a small overhead to all interface method calls; we later describe an alternative representation that avoids the additional parameter.

If the signature (that is, parameter and return types) of a method declaration does not reference type parameters, the method is compiled into a single native version and the boxed version is a simple wrapper. Otherwise, the method is compiled into a boxed version (with all parameters and return values boxed as plain pointers) and native wrappers are used.

As with functions, all the signatures involved in calls to native versions of

⁵⁶<https://github.com/golang/go/blob/go1.26.5/src/internal/abi/iface.go>

interface methods are known statically, so native wrappers can be generated statically.

The native wrapper needs to know the index of the corresponding boxed function pointer in the method table; this can be addressed by using a fixed index in the native wrapper, and only sharing native wrappers for methods that have the same signature *and* the same method table index.

The native wrappers can be looked up when creating an interface value (either statically or based on the dynamically-computed method signature), or they can be lazily set by the caller, similarly to closures.

Generic interface methods take additional dictionary parameters corresponding to each of their type parameters, similarly to generic functions.

Alternative representation. The runtime representation described above is the one that we feel is most natural, but there may be objections to an additional parameter in every interface method call. There is another approach that avoids this additional parameter, while still providing methods of generic types with access to the receiver’s type information.

As explained in [Section 4.5](#) (Generic Go to Go), run-time type information cannot be added to generic struct types due to backward compatibility. However, the data pointer of an interface value need not use the same representation as the struct.

Hence the additional method parameter can be avoided as follows:

- When a value of a generic type is converted to an interface value, the data pointer is set to a newly-allocated block of memory populated with a pointer to the itable (the itable that the interface value will hold) followed by the value’s actual data.
- Interface method calls use Go’s existing calling convention—only the data pointer is passed, with no additional parameter for the receiver’s type information.
- Methods of generic types fetch the itable pointer from the data pointer and use it to access the receiver’s type information. The value’s actual data is accessed by offsetting the data pointer.
- Likewise, native wrappers fetch the itable pointer from the data pointer and use it to access the method table. (Native wrappers are only needed

for methods of generic types, so they may assume that the itable pointer is present.)

With this approach, interface method calls in non-generic code will not be impacted, but more memory allocations will be required when converting values of generic types to interface values.

Interface conversion and type assertions. When a type parameter is used as a type argument of a generic function that has a different constraint, a new itable is created by filtering the original itable to the relevant methods (as Go currently does for interface conversions).

A type assertion `v.(I)` where `I` is an interface type and `v` is an interface value containing a value of a generic type is performed in a similar way to existing type assertions (the full method table of the type is scanned to check if it implements the interface), except the method signatures need to be computed dynamically based on the type arguments of the generic type.

If using the alternative representation, a type assertion `v.(G)` where `G` is an instantiation of a generic type requires offsetting the data pointer to extract the value, to account for the itable pointer.

Example. To help illustrate the approach, we translate an example from the literature [74, Fig. 2] to a compiled form, which we present as Go-like pseudocode.

The example is shown in Figure 4.1. It demonstrates a direct method call and an interface method call on a value of a generic type. The original example specified constraints on the receiver’s type parameter (a feature permitted by Featherweight Generic Go but not by Go; our approach could be extended to support this feature). We “lifted” the constraints to the type declaration of the receiver to make the example a valid Go program.

The runtime representation used by our compiled translation is shown in Figure 4.2. This representation is a simplified version of Go’s application binary interface^{57,58} modified to use the two-pointer function representation and to store dictionaries for type arguments.

The compiled form is given in Figure 4.3. We use Go-like syntax, but the intent is a close correspondence with low-level machine code. We use several

⁵⁷<https://github.com/golang/go/blob/go1.26.5/src/internal/abi/iface.go>

⁵⁸<https://github.com/golang/go/blob/go1.26.5/src/internal/abi/type.go>

```

type Num    struct { val int }
type Format interface { format() string }

func (this Num) format () string { return strconv.Itoa(this.val) }

func formatSome(x Format) string { return x.format() }

type Pair[T Format, U Format] struct { left T; right U }

func (this Pair[T, U]) format () string {
    return this.left.format() + ...
}

func main() {
    var p Pair[Num, Num] = Pair[Num, Num]{Num{1}, Num{2}}
    p.format()
    formatSome(p)
}

```

Figure 4.1: An example of generics from [74, Fig. 2] (modified)

```

type Func struct { Native Pointer; Boxed Pointer }
type Type struct { Size int; TypeArgs []*ITab; Offsets []int; ... }
type ITab struct { Type *Type; Funcs []Func }
type InterfaceValue struct { ITab *ITab; Data Pointer }

```

Figure 4.2: Runtime representation for dictionary-passing

ad hoc syntactic elements for readability:

- `Pointer` represents a raw pointer (corresponding to Go's `unsafe.Pointer`⁵⁹). For readability, we omit casts and use `Pointer` as if it has whichever type is desired.
- We borrow from C the functions `alloca(size)` and `memmove(dst, src, size)` for allocating and copying memory, and we use `sizeof` and `offsetof` for statically-known sizes and field offsets.
- For readability, we use descriptive string literals to represent the integer constants used as table indexes.
- We use Go's syntax `&Foo{...}` for allocating and initialising a pointer to a struct, and by extension we use `&expr` with arbitrary expressions to indicate boxing of a value as a pointer (by allocating a block of memory of the value's size and copying the value to that block).

There are several details worth noting about the compiled translation:

- The function `formatSome_boxed` demonstrates unboxing of function parameters and boxing of return values.
- The signature of the method `Pair[T, U].format` does not refer to type parameters, so it is compiled into a single native version with a trivial boxed wrapper. If the signature had mentioned `T` or `U` it would have been compiled into a boxed version, and multiple native wrappers would have been generated as required.
- The body of `main` could be simplified because the size of `Num` is known statically, but for illustrative purposes we use the more general form.

We believe that this approach addresses the main implementation challenges of first-class generic functions and generic interface methods, and is an important step towards adding them to the language.

⁵⁹<https://pkg.go.dev/unsafe#Pointer> Package `unsafe`. The Go Standard Library.

```

func Num_format_native(itab *ITab, this Pointer) string {
    return strconv.Itoa(*(this + offsetof(Num, val)))
}

func Num_format_boxed(itab *ITab, this Pointer) Pointer {
    return &Num_format_native(itab, this)
}

func formatSome_native(x InterfaceValue) string {
    return x.ITab.Funcs["format"].Native(x.ITab, x.Data)
}

func formatSome_boxed(x Pointer) Pointer {
    return &formatSome_native(*x)
}

func Pair_format_native(itab *ITab, this Pointer) string {
    var T *ITab = itab.Type.TypeArgs["T"]
    var left Pointer = this + itab.Type.Offsets["left"]
    return T.Funcs["format"].Native(T, left) + ...
}

func Pair_format_boxed(itab *ITab, this Pointer) Pointer {
    return &Pair_format_native(itab, this)
}

var Num_Type *Type = &Type{sizeof(int), {}, {}} // Offsets not needed
var Num_Format_ITab *ITab = &ITab{Num_Type,
    {"format": Func{Num_format_native, Num_format_boxed}}}

var Pair_Num_Num_Type *Type = &Type{Num_Type.Size + Num_Type.Size,
    {"T": Num_Format_ITab, "U": Num_Format_ITab},
    {"left": 0, "right": Num_Type.Size}}
var Pair_Num_Num_Format_ITab *ITab = &ITab{Pair_Num_Num_Type,
    {"format": Func{Pair_format_native, Pair_format_boxed}}}

func main() {
    var p_type *Type = Pair_Num_Num_Type
    var p Pointer = alloca(p_type.Size)
    memmove(p + p_type.Offsets["left"], &Num{1}, Num_Type.Size)
    memmove(p + p_type.Offsets["right"], &Num{2}, Num_Type.Size)
    Pair_format_native(Pair_Num_Num_Format_ITab, p)
    formatSome_native(InterfaceValue{Pair_Num_Num_Format_ITab, p})
}

```

Figure 4.3: Compilation of the example using dictionary-passing

Chapter 5

Lambda Lifting

Preface

This chapter is an extended version of a paper [49] presented in the IEEE/ACM 14th International Conference on Formal Methods in Software Engineering (FormaliSE '26), co-authored with Steve Reeves. The additions are as follows:

- Extended discussions of the Danvy–Schultz algorithm (Section 5.2.1) and CertiCoq (Section 5.2.2).
- A complex lifting example (Section 5.4.3).
- Correctness and time complexity proofs for Algorithm 1 (there was only enough space in the paper for the proofs for Algorithm 1b).
- A second data structure approach based on an augmented AST (in Section 5.4.6).
- *Algorithm 1b'*, which explicitly shows the modifications to Algorithm 1b described in Section 5.4.6.
- A discussion of optimality and possible improvements, including a sketch of a further-improved algorithm (Section 5.4.7).
- An extended discussion of the Fischbach–Hannan specification, including an example demonstrating how “bare” singleton types make the specification unsound (in Section 5.5.1).
- An extended discussion of the Lean proof of Theorem 5.6.1.

- [Figure 5.5](#), showing the Lean translation of the type system.
- A new section about the testing process ([Section 5.8](#)).
- A new section about related problems ([Section 5.9](#)).
- An extended analysis of the worst-case output size ([Appendix A](#)).

Abstract

Lambda lifting is a technique used in compilers to convert nested function definitions to top-level function definitions. A series of papers has led to an $O(n^2)$ algorithm, however it is complex. We present a simple $O(n^2)$ algorithm for lambda lifting and prove its correctness. We also formalise a lambda lifting specification from the literature in Lean 4, and use that to prove some of the properties and test our algorithm on generated test cases. One of our contributions is to formalise the notion of a “complete” and “minimal” lifting, addressing a small issue with the handling of unused functions that to our knowledge affects all previous algorithms.

5.1 Introduction

Improving long-standing algorithms, as in our example here where we show how to make a widely used algorithm faster, carries with it the burden of showing that the improvement still matches the specification. In this paper we show by example how existing, widely used and very well established algorithms and implementations might be re-verified by taking a formal specification regarded as the canonical definition of an algorithm—in this case given as a set of logical rules in natural deduction form—and formalising it in a proof assistant (Lean 4 in our case); then a combination of machine-checked and “pen and paper” proofs can be used to demonstrate that the improved algorithm still meets the definition given by the original specification. We suggest that this might be a generally good thing to do even when making “obvious” improvements to widely used algorithms, since much can be at stake.

As explained in [Section 1.1](#), lambda lifting is a technique used to convert nested functions to top-level functions by introducing additional parameters.

We refer to these additional parameters as “extraneous parameters” (following the terminology of [15, 55]).

For example, consider the following code (written in Python for illustration):

```
def triangular_number(n):
    def sum_from(i):
        if i > n: return 0
        else:      return i + sum_from(i + 1)
    return sum_from(1)
```

It defines a function `triangular_number` that computes the sum of $\{1..n\}$, using a local helper function called `sum_from` that computes the sum of $\{i..n\}$ recursively.

Note that the body of `sum_from` refers to `n`, which is a parameter of the enclosing function.

In order to convert the function `sum_from` to a top-level function, we need to introduce a parameter to `sum_from` allowing the value of `n` to be passed in. The result of lifting is:

```
def sum_from(n, i):
    if i > n: return 0
    else:      return i + sum_from(n, i + 1)
def triangular_number(n):
    return sum_from(n, 1)
```

Note how the argument `n` must be added to all the call sites of `sum_from`. If the function is used as a value (in a language that supports higher-order functions / first-class functions), then in general the added arguments would need to be passed using partial application, producing a function value with the same signature as the original function. In this paper we focus on approaches that avoid partial application (except when functions are used as values), because it requires special support from the language, such as closures, and is not applicable to types in Go for example.

Also note the subtle change to the variable bindings: even though the expression `i > n` in `sum_from` did not change textually, the binding of `n` in that expression has changed—in the original program it was bound to the parameter `n` of `triangular_number`, while in the lifted program it is bound to the extraneous parameter of `sum_from`. If we were to choose a unique name for the extraneous parameter, this change would become apparent:

```

def sum_from(n2, i):
    if i > n2:
        return 0
    else:
        return i + sum_from(n2, i + 1)
def triangular_number(n):
    return sum_from(n, 1)

```

In this version, all references to `n` in `sum_from` are explicitly changed to `n2`. The fact that `triangular_number` passes `n` to `sum_from` ensures that `n` and `n2` will have the same value, preserving the behaviour of the original program.

The extraneous parameters of `sum_from` in this example are simply the free variables of the original definition. But determining the extraneous parameters in general is non-trivial for two reasons:

- A local function may call other local functions. If those functions require extraneous parameters, then corresponding arguments will need to be introduced in the calls to those functions, which will in turn produce new free variables (unless those variables are declared by the calling function). This applies transitively.
- Local functions may be mutually recursive. So an algorithm for determining the extraneous parameters needs to be able to handle programs whose call graph contains cycles.

Further, we wish to find the smallest set of extraneous parameters, to produce more efficient code. (If we did not care about minimality we could simply add all the variables from the enclosing scopes as the extraneous parameters of each function; that would produce valid code, but in general it adds unnecessary parameters.)

Determining the extraneous parameters is the core of lambda lifting. Once that is done, there are a few remaining tasks, but they are all straightforward manipulations of the abstract syntax tree:

1. insert the extraneous parameters in the parameter lists and argument lists;
2. rename the functions and parameters as necessary to avoid naming conflicts;

3. move all local function definitions to the top-level scope (“block floating”).

There are algorithms for determining the minimal extraneous parameters in the literature, but they have a suboptimal time complexity or are complex. The algorithm we present in this paper is simple, and its *worst-case* time complexity is $O(n^2)$ like the fastest existing algorithm [56].

We go further and give a tight time complexity analysis for *all* cases by considering the size of the output program. The worst-case size of the output program is $\Theta(n^2)$ where n is the size of the input program;¹ but if each top-level function only contains a small² number of nested function or a small number of parameters, then the size of the output program is $\Theta(n)$. We show that the time complexity of our algorithm is linear in the size of the output program (Theorem 5.4.10); hence the time complexity of our algorithm is optimal on all cases if the lifted program is to be output. In contrast, the fastest existing algorithm has an optimal *worst-case* time complexity, but there exist non-worst-case inputs on which its time complexity is suboptimal.

5.2 Literature

Church’s lambda calculus [9] is one of the earliest examples of the notion of nested functions; the reduction rules handle nested functions using capture-avoiding substitution.

LISP had nested functions but used what later became known as “dynamic scoping”, which is generally considered undesirable [70] and was regarded as a bug [51].

Algol 60 had nested functions that could be passed as parameters but could not be returned from a function. This was implemented using special stack frame links [88][68, §2.2].

Closures were first described by Landin in [42]. Closures are used by PAL [18] and Scheme [88, 75].

Hughes [34] describes a method for converting λ -expressions containing free variables to supercombinators (functions that contain no free variables). The method relies heavily on partial application. Briefly, an expression such as

¹This is true both when the program size is measured in tokens and when it is measured in bytes; see Appendix A.

²By “small” we mean less than an arbitrary constant.

$\lambda y. y + x$ is translated to αx where α is defined as the function $\alpha x y = y + x$. These ideas are also described in [64] in more detail.

Johnsson [35] describes a different method that does not make use of partial application (except when functions are used as values). As described in [Section 5.1](#), this “lambda lifting” method eliminates free variables by assigning each function a set of extraneous parameters and inserting corresponding extraneous arguments in each call site (and at each reference if the function is used as a value). The implementation is somewhat complicated because inserting the extraneous arguments may produce new free variables, so computing the final sets of extraneous parameters requires computing a variation of the transitive closure. Johnsson’s approach is to set up a system of set equations and solve it using substitution for each `letrec` expression, with time complexity $O(n^3)$.

The code generated by Johnsson-style lifting has different performance characteristics compared to supercombinator/closure conversion [64, 25], and both are still widely used [15]. Johnsson-style lifting also been used to optimise partial evaluators [55].

Fischbach and Hannan give a formal specification of lambda lifting [19] and prove it preserves operational semantics. Their specification is quite general: it permits both Hughes-style and Johnsson-style liftings, as well as other forms of lifting. They also describe an algorithm similar to Johnsson’s and prove its correctness with respect to the specification. However, they do not specify any notion of a “complete” lifting that eliminates all free variables, and they do not prove their algorithm computes the minimal extraneous parameters.

CertiCoq [63] is a compiler for Gallina (the specification language of Coq) that includes a verified lambda lifting transformation; the time complexity is not specified but appears to be at least $O(n^3)$. We discuss it in more detail in [Section 5.2.2](#).

There has been a series of papers on improving the time complexity of Johnsson-style lifting using graph-based approaches. The most obvious approach for computing the extraneous parameters of a function is to take the union of its free variables and the extraneous parameters of the functions it calls. However, a simple top-down recursive traversal of the call graph would fail due to infinite recursion if there are any cycles (mutually recursive functions). It is possible to avoid infinite recursion by stopping the recursion when it reaches

functions that have already been visited, and it can be shown that this will produce the correct extraneous parameters for the function that the recursion initially started from, however the sets of extraneous parameters computed for other functions visited during the recursion may be incomplete. An algorithm that performs independent traversals starting from each function would produce the correct extraneous parameters, but its overall time complexity would be $O(n^3)$.

Danvy and Schultz [15] describe an $O(n^2)$ algorithm which eliminates cycles in the call graph by coalescing strongly-connected components (SCCs) and assigning the same extraneous parameters to all functions within each SCC. However, this approach is flawed; one analysis is given in [55], and we give our own analysis in [Section 5.2.1](#).

Morazán and Schultz [56] describe an $O(n^2)$ algorithm that solves the issue. The algorithm is complex—it uses dominator trees in addition to SCCs.

Newer lambda lifting implementations such as in GHC [25] continue to use $O(n^3)$ algorithms, suggesting a need for a simple, fast, and correct algorithm. Such an algorithm will be useful to future compiler implementors, and has applications beyond nested functions—for example, it can be used to lift local types inside generic functions in the Go programming language (which has fast compilation as a core goal [66]).

5.2.1 The Danvy–Schultz algorithm

The Danvy–Schultz algorithm [15, Figure 9] uses a top-down recursive traversal of an input program consisting of `letrec` expressions. For each `letrec` expression, it builds a call graph (restricted to the functions bound in the `letrec`), coalesces SCCs, then uses the resulting condensation graph to propagate free variables in $O(n^2)$ time³; all functions in a component are assigned the same extraneous parameters.

The issue is that the Danvy–Schultz algorithm does not consider calls to

³As written the algorithm is actually $O(n^3)$, but that can be easily fixed. The flaw in the complexity analysis is that while the number of edges in the *original* graph is linear, the total number of edges between nodes and the functions associated with their successor nodes can be $\propto n^2$ (for example, when the condensation graph is a star containing $n/2$ nodes of size 1 whose successor is a node of size $n/2$). The fix is to change $F_p = \bigcup_{q \in \text{succ}_p} q$ to $F_p = \{c(q) \mid q \in \text{succ}_p\}$ where c is a choice function that selects an arbitrary element from each node q . This change is valid, because all functions in the same node have the same extraneous parameters.

functions in parent scopes. This is illustrated by the following example (using the syntax from [19], where “@” denotes function application):

```
letrec main =  $\lambda$  x .
  letrec e =  $\lambda$  y .
    letrec f =  $\lambda$  z . e @ 3
    in x + f @ 2
  in e @ 1
in main @ 0
```

The function **e** references **x**, so **x** must be added as an extraneous parameter of **e**. The function **f** calls **e**, so **x** must also be added as an extraneous parameter of **f**.

However, the Danvy–Schultz algorithm does not add **x** as an extraneous parameter of **f**. When that algorithm processes the innermost **letrec** (where **f** is defined), the call to **e** is excluded in **foreach** $g \in \text{FF}(f_i) \cap \{f_1, \dots, f_k\}$ **do** because $\text{FF}(f_i) \cap \{f_1, \dots, f_k\} = \{e\} \cap \{f\} = \{\}$, and that call is never considered elsewhere.

The issue can be fixed by identifying calls to functions in parent scopes and adding their extraneous parameters, specifically by changing **let** $V = \dots$ to **let** $V = \dots \cup (\bigcup \{V_g \mid g \in C, (g, V_g) \in S\})$ where $C = \bigcup_{f \in p} \text{FF}(f) \setminus \{f_1, \dots, f_k\}$ are the calls to functions in parent scopes.

As written, this change increases the time complexity $O(n^3)$, as demonstrated by the example to the right.

We use Scheme syntax (as in [55, 56]) for this example and all the following examples, because it is clear and easy to parse; it corresponds directly to the **letrec** syntax used in the specification, except that Scheme allows 0-arity functions (which are convenient for the examples).

The functions **h**, **g1**..**gn** all have n extraneous parameters. When processing each of the functions **f2**..**fn** with our fix, C will be $\{g1, g2, \dots, gn\}$ so

```
(define (main x1 x2 ... xn)
  (define (h) (+ x1 x2 ... xn))
  (define (g1) (h))
  (define (g2) (h))
  ...
  (define (gn) (h))
  (define (f1)
    (define (f2)
      ...
      (define (fn)
        (+ (g1) (g2) ... (gn)))
      (fn)
      ...))
    (f2))
  (f1))
```

the algorithm will need to compute the union of n sets of size n . Assuming set union is $O(n)$, the overall time complexity would therefore be $O(n^3)$.

It is possible to improve the time complexity of our fix to $O(n^2)$ by propagating the extraneous parameters eagerly.

A variation of the Danvy–Schultz algorithm is to compute the SCCs and coalesce them over the call graph of the *entire* program; that is the interpretation taken in [55]. In this variation, free variables are correctly propagated along calls to functions in parent scopes, and the time complexity remains $O(n^2)$, however there are cases where unnecessary extraneous parameters are introduced (as identified in [55]).

To illustrate the whole-program version of the SCC approach, consider the following example:

```
(define (main x)
  (define (f)
    (define (g) (+ x (h)))
    (define (h) (f))
    (g))
  (f))
```

The call graph contains a cycle: `f` calls `g` calls `h` calls `f`. Coalescing those functions into a single node, $\{f, g, h\}$, produces an acyclic graph with a single edge: `main` calls $\{f, g, h\}$. The free variables in $\{f, g, h\}$ are `x`, so `x` is added as an extraneous parameter of `f`, `g`, and `h`. Note that `x` is not added as an extraneous parameter of `main` (even though `main` calls $\{f, g, h\}$) because `x` is declared in `main`.

The extraneous parameters added by the whole-program SCC approach to the above example are minimal, but on the following variation it adds unnecessary extraneous parameters:

```
(define (main)
  (define (f x)
    (define (g) (+ x (h)))
    (define (h) (f 1))
    (g))
  (f 0))
```

In this variation, `x` is declared as a parameter of `f` rather than `main`. The call graph is unchanged, so `f`, `g` and `h` are again coalesced into a single node

with x as a free variable. So x is added as an extraneous parameter of g and h (but not of f , because x is declared in f). However, it is not necessary to add x as an extraneous parameter of h , because x is not free in h and h does not call any function that has x as an extraneous parameter. (Aside: Note that x is again not added as an extraneous parameter of `main`, but this time it is because x is not in the scope of `main`.)

One way to view the issue with the whole-program SCC approach is that the true SCCs of the call graph actually depend on which variable is being considered; this is illustrated in [Section 5.4.3](#). Specifically, for the purpose of determining which functions require a variable x as an extraneous parameter, we should exclude calls made by the function in which x is declared from the call graph, because a variable should never be added as an extraneous parameter of its declaring function. In fact, this leads to a novel algorithm which is relatively simple and fast: for each function f , create a modified call graph that excludes calls made by f and coalesce SCCs, then use a top-down recursive traversal of the resulting condensation graph to propagate the variables declared in f to the functions that require them as extraneous parameters. The computed extraneous parameters are minimal, and the overall time complexity remains $O(n^2)$ despite the multiple traversals over the call graph.

It turns out we can do even better: the algorithm we describe in this paper is even simpler and has a faster best-case running time—it does not require SCCs, and the time complexity is linear in the size of the output program.

5.2.2 CertiCoq

CertiCoq [63] is a compiler for Gallina (the specification language of Coq) that includes a verified lambda lifting transformation.

It does not apply lambda lifting to all the functions—it uses heuristics to choose between lambda lifting and closure conversion.

The lambda lifting transformation itself does not produce minimal extraneous parameters; a separate dead parameter elimination transformation is used instead.

The time complexity of the lambda lifting transformation is not specified, but appears to be at least $O(n^3)$.

The time complexity of the dead parameter elimination algorithm is quadratic

in the size of the program. This is actually $O(n^4)$ where n is the size of the original program, because adding the extraneous parameters can increase the size of the program to $O(n^2)$. The time complexity can be improved to be linear in the size of the program (and hence $O(n^2)$ in terms of the original program) using the techniques from our algorithm.

In fact, this leads to an $O(n^2)$ algorithm for computing the minimal extraneous parameters when all functions are lifted: assign to each function all the variables of the enclosing scopes as extraneous parameters, then perform the improved dead parameter elimination. This is $O(n^2)$, but again is not linear in the size of the output program (unlike our algorithm).

5.3 Syntax

We use the language from [19]. This language is the simply typed lambda calculus, extended with **letrec** expressions allowing mutually-recursive function definitions. The syntax for expressions and types is as follows:

$$\begin{aligned} e &::= c \mid x \mid \lambda x. e \mid e_1 @ e_2 \mid \text{letrec } \overline{f_i = e_i} \text{ in } e \\ \tau &::= \iota \mid \tau \rightarrow \tau \end{aligned}$$

These symbols denote (in order) expressions, pre-defined constants, identifiers, abstractions, function application, **letrec** expressions with mutually-recursive bindings, types, base types, and function types. We refer the reader to [19] for a more detailed explanation. The overline notation indicates repetition: $\overline{f_i = e_i}$ denotes a sequence of bindings $f_1 = e_1; f_2 = e_2; \dots; f_n = e_n$.

We also use the type system from [19, Fig. 1]; we include the rules alongside our Lean translation in [Figure 5.5](#). The rules are standard and unremarkable.

For the lambda lifting specification, τ is extended using singleton types:

$$\tau ::= \iota \mid \tau \rightarrow \tau \mid \{e\}_\tau \rightarrow \tau$$

$\{e\}_\tau$ is a singleton type; the only term of that type is the expression e of type τ . We restrict the expression e in singleton types to be an identifier, as done in the version of the lambda lifting specification in [19, §3.2]. The more general specification in [19, §4] does not have this restriction and allows lifting arbitrary expressions (enabling full laziness), but in this paper our focus is solely

on parameter lifting. The type $\{x\}_{\tau_1} \rightarrow \tau_2$ denotes the type of the function that results when x is inserted as an extraneous parameter of a function of type τ_2 .

Note that the grammar does not allow “bare” singleton types—singleton types may only appear as part of a lifted function type. This restriction is subtle but critical; we discuss this in [Section 5.5.1](#).

We treat $\lambda x_1 \dots \lambda x_n. e$ as a single multi-parameter function (for the purpose of the call graph and analysis of free variables).

We distinguish between three types of bindings:

- A **letrec** binding of the form $f = \lambda x. e$ is a *function binding*.
- A **letrec** binding of the form $f = e$ where e is not an abstraction is a *variable binding*.
- A binding created by a binder λx is a *parameter binding*.

A *variable* is an identifier bound by a variable or parameter binding, and a *function name* is an identifier bound by a function binding.

A function $\lambda \dots$ that occurs as the right-hand side of a function binding is a *named function*, while a function that occurs elsewhere is an *anonymous function*.

The reason we distinguish between function bindings and variable bindings is that they are handled differently in Johnsson-style lifting; in particular, extraneous parameters are never inserted directly in variable bindings. This terminology also facilitates discussion of the simple parameter lifting algorithm in [19, Fig. 5], which is very similar to Johnsson’s algorithm but does not support variable bindings⁴ (that is also the reason we distinguish between variable bindings and parameter bindings).

Shadowing is when an identifier in a binding is already bound in the enclosing scope. The Fischbach–Hannan specification forbids shadowing to avoid some issues [19, p. 513].⁵

⁴Instead, the syntax sugar “let” is defined [19, p. 513] to allow (non-recursive) variable declarations in the examples.

⁵An alternative is to generate unique names for the extraneous parameters and update references accordingly.

For the purpose of our algorithm, we further assume all identifiers are distinct; that can be ensured using a preprocessing step that renames identifiers (α -conversion),⁶ or by assigning the identifiers unique IDs as in [Section 5.4.6](#).

5.4 The algorithm

5.4.1 Definitions

We use the standard definitions for *scope*, *free variables*, and *closed expressions*. We use the term *extraneous parameters* to refer to the parameters added by the lambda lifting process (following the terminology of [15, 55]).

Definition 5.4.1 (Declaring function). The *declaring function* of an identifier binding is the innermost function that encloses the binding, or null if the binding is not enclosed by any function.

Definition 5.4.2 (Referencing). An occurrence of an identifier is *referenced directly* by a function if the function is the innermost function that encloses the occurrence. (We will sometimes abbreviate this to “referenced” for readability.)

Definition 5.4.3 (Function reference graph). The *function reference graph* is analogous to the call graph. The nodes are the functions (including anonymous functions), and there is an edge from f to g if g is referenced directly by f (either because f calls g or because f uses g as a value). When g is an anonymous function, we say that it is referenced directly by f if f is the innermost enclosing function of g .

Definition 5.4.4 (Non-local variables). An occurrence of a variable in a function is *non-local* if the function is not the declaring function of the variable and the occurrence is not a reference to a global variable.

Definition 5.4.5 (Referenced non-local variables). A variable x is a *referenced non-local variable* of a function if there exists an occurrence of x which is referenced directly by the function and is non-local in the function.

⁶Renaming identifiers also avoids name clashes when functions are block-floated, and ensures a `letrec` expression does not contain multiple bindings with the same identifier (the Fischbach–Hannan specification allows that, and it does not cause issues there, but it does cause issues when block-floating is performed).

The referenced non-local variables of a function are a subset of the function’s free variables, restricted to ensure that the extraneous parameters derived from them are minimal:

- We exclude global variables because they remain accessible after lifting, so they never need to be added as extraneous parameters.
- We exclude function references because we are interested in block-floating all functions to the top-level scope, so after block-floating all functions will be in scope (as noted in [19]).
- We only include variables referenced directly by the function to avoid unnecessary extraneous parameters in the presence of unused functions.

To illustrate the last point, consider the following example:

```
(define (main x)
  (define (f)
    (define (g) x)
    1)
  (f))
```

The variable `x` is a non-local variable in `f` but is not referenced directly by `f`. It becomes clear that `f` does not require `x` as an extraneous parameter (unlike `g`) after we perform block-floating:

```
(define (g x) x)
(define (f) 1)
(define (main x) (f))
```

Once `g` is block-floated out of `f`, `f` does not reference `x`; and `f` does not reference any function that has `x` as an extraneous parameter, so `x` does not need to be added as an extraneous parameter of `f`.

Alternatively, the issue can be avoided by eliminating dead code in a preprocessing step.⁷

⁷Algorithm-specific fixes also exist. For example, the simple parameter lifting algorithm in [19, Fig. 5] can be fixed by changing $\bigcup \theta_i \cup \theta$ to θ in the rule (letrec); the proof of Theorem 5 continues to hold trivially—specifically, Lemma 3 continues to hold because $\bigcup \theta_i \cup \theta \subseteq \text{dom}(\Gamma)$ implies $\theta \subseteq \text{dom}(\Gamma)$; and the algorithm continues to produce a complete lifting (excluding anonymous functions)—the premise $\Lambda^* \triangleright e : (\tau, \theta) \Rightarrow m$ guarantees that if f_i is referenced in e then θ will be a superset of θ_i , so any parameters in $\bigcup \theta_i$ that are not already in θ are unnecessary.

Definition 5.4.6 (Solution). The *solution* for a program is a mapping EP from each function to a set of extraneous parameters satisfying the following conditions, for all functions f and g and for all identifiers v :

1. v is a referenced non-local variable of $f \Rightarrow v \in EP[f]$
2. $v \in EP[f] \wedge g$ references $f \wedge g$ is not the declaring function of $v \Rightarrow v \in EP[g]$
3. Nothing else is in EP .

In other words, the referenced non-local variables in the original program must be included as extraneous parameters, and extraneous parameters need to be propagated to callers and other referencing functions (except the declaring function) because they will be inserted at call sites and references; this is somewhat similar to [56]. A variable is never inserted as an extraneous parameter of its declaring function, ensuring that variables never escape their original scope when they are inserted as extraneous parameters and arguments (see Theorem 5.6.4).

This definition is novel, and is one of the key insights behind our algorithm.

We prove in Section 5.6 that the solution gives the minimal extraneous parameters in the class of liftings we are interested in.

(Note that anonymous functions are included in this definition. We assume each function is assigned a distinct map key.)

5.4.2 A simple lifting example

The conditions from the definition of the solution are straightforward to apply. We demonstrate the process using the following example; our analysis mirrors the steps performed by our algorithm.

```
(define (main w x)
  (define (e)
    (define (f) (+ x y (e)))
    (define y 1)
    (define (g) (+ w (e)))
    (f))
  (e))
```

The variable w is a referenced non-local of g , so it is added to $EP[g]$. The function g is unused, so w is not added to other sets.

The variable x is a referenced non-local of f , so it is added to $EP[f]$. The function e calls f and $x \in EP[f]$, so x is added to $EP[e]$. Likewise g calls e so x is added to $EP[g]$. The function $main$ calls e , but $main$ is the declaring function of x so x is not added to $EP[main]$. The function f calls e (a cycle), so x needs to be added to $EP[f]$; but x is already in $EP[f]$ so there is no change.

The variable y is a referenced non-local of f , so it is added to $EP[f]$. The function e calls f , but e is the declaring function of y so y is not added to $EP[e]$.

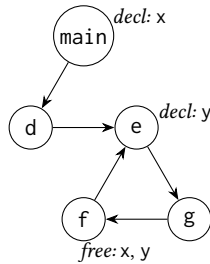
Therefore, $EP = \{main : \{\}, e : \{x\}, f : \{x, y\}, g : \{w, x\}\}$.

5.4.3 A complex lifting example

We now present an example with a more complex call graph and work through it in full (including the block-floating step). This example highlights the significance of the declaring function on propagation, like the previous example; and it gives a good illustration of how the whole-program SCCs depend on the variable being considered.

```
(define (main x)
  (define (d) (e 1))
  (define (e y)
    (define (f) (+ (e 2) x y))
    (define (g) (f))
    (g))
  (d))
```

This example can be visualised using the following call graph diagram:



The variable x is a referenced non-local of f , so it needs to be introduced as a parameter of f . The function g does not refer to x directly, however since

`g` calls `f` and calls to `f` will need `x` as an argument, `x` must be introduced as a parameter of `g`. Likewise `x` must be introduced as a parameter of `e` (because it calls `g`) and to `d` (because it calls `e`). But `x` does not need to be introduced as a parameter of `main`, even though `main` calls `d`, because `main` is the declaring function of `x`.

The variable `y` is a referenced non-local of `f`, so it needs to be introduced as a parameter of `f`, and likewise of `g` (which calls `f`). But `y` does not need to be introduced as a parameter of `e`, even though `e` calls `g`, because `e` is the declaring function of `y`.

Therefore the smallest sets of extraneous parameters are as follows:

$$\text{main} : \{\}, d : \{x\}, e : \{x\}, f : \{x, y\}, g : \{x, y\}$$

The only difference between `x` and `y` is their declaring function, and yet their propagation is very different. When propagating `x` the component `{e, f, g}` needs to be coalesced to eliminate cycles, but when propagating `y` that component must not be coalesced.

When the extraneous parameters are inserted in the parameter lists and call sites, and the function definitions are moved to the top level, it results in the following lifted program:

```
(define (d x) (e x 1))
(define (f x y) (+ (e x 2) x y))
(define (g x y) (f x y))
(define (e x y) (g x y))
(define (main x) (d x))
```

5.4.4 Algorithm 1

We now describe an algorithm and prove it computes the solution defined above.

We present two versions of the algorithm. The first version is simple and intuitive, but the proofs are not fully rigorous in some parts due to inconveniences with reasoning about recursion. So we later present an iterative version of the algorithm with corresponding proofs that are more rigorous.

The key properties of our algorithm are that it traverses the function reference graph in a bottom-up fashion and that it processes each variable

```

1 define DFS(v: Identifier, f: Identifier, traversed_funcs: Set):
2   if f not in traversed_funcs:
3     add f to traversed_funcs
4     for g in functions that reference f:
5       if g is not the declaring function of v:
6         DFS(v, g, traversed_funcs)
7
8 define multi_source_DFS(v: Identifier, starting_funcs: Set) → Set:
9   traversed_funcs := new set
10  for f in starting_funcs:
11    DFS(v, f, traversed_funcs)
12  return traversed_funcs
13
14 for v in all_variables:
15   starting_funcs := functions where v is a referenced non-local
16   traversed_funcs := multi_source_DFS(v, starting_funcs)
17   for f in traversed_funcs:
18     add v to the extraneous parameters of f

```

Algorithm 1

separately. This makes it easy to handle cycles—we simply skip functions that have already been visited. Processing each variable separately might seem inefficient at first glance, because it requires multiple passes over the function reference graph; however that is not the case (algorithms that perform a single pass over the call graph will in general spend more time processing each node, because they need to merge sets containing multiple variables).

For each variable, [Algorithm 1](#) performs a depth-first traversal (DFS) starting from the functions where the variable is a referenced non-local and propagating to the functions that reference them. The set *traversed_funcs* records the functions traversed by the DFS. If the DFS encounters a function that has already been traversed, it skips that function; otherwise it adds the function to the set. As a special case, the DFS does not propagate to the declaring function of the variable. When the traversal ends, the variable is added as an extraneous parameter of each of the traversed functions.

We assume a preprocessing step has been used to compute the (reverse) function reference graph and the referenced non-local variables of the functions.

Theorem 5.4.7. *The output of Algorithm 1 is the solution.*

Proof. Consider a single iteration of the top-level **for** loop on line 14, fixing the variable v .

As a result of lines 17–18, at the end of the iteration we have

$$f \in \text{traversed_funcs} \Leftrightarrow v \in EP[f]$$

for all functions f . So we need to show that the functions in traversed_funcs satisfy the definition of a solution.

Note that the only place that adds elements to traversed_funcs is line 3 in DFS , and that elements are never removed. Observe that DFS has the postcondition $f \in \text{traversed_funcs}$.

We can show that DFS terminates by induction on the number of unvisited functions (functions in the input program that are not in traversed_funcs). If the current function is already in traversed_funcs , DFS returns immediately. Otherwise, DFS adds the current function to traversed_funcs , so the number of unvisited functions is strictly less in the recursive calls, and the inductive hypothesis implies that they will terminate.

Condition (1) from the definition of a solution is satisfied, because multi_source_DFS invokes DFS on each of the starting functions (which are the functions where v is a referenced non-local variable), and DFS adds those functions to traversed_funcs .

Condition (2) from the definition of a solution is satisfied, because whenever a function is added to traversed_funcs , DFS recurses into the functions that reference it (except for the declaring function of v), ensuring that the necessary functions will also be added to traversed_funcs .

Condition (3) is satisfied, because the only place that adds elements to traversed_funcs is line 3 in DFS , and DFS is only called from multi_source_DFS and DFS , and the functions added by those calls are justified by conditions (1) and (2) respectively. $\square$

Theorem 5.4.8. *The time complexity of Algorithm 1 is linear in the size of the output program.*

Proof. We assume the availability of map and set data structures providing $O(1)$ access, $O(1)$ initialisation, and iteration linear in the number of elements; see Section 5.4.6.

The preprocessing step to compute the function reference graph and the referenced non-local variables of the functions can be done using a single linear-time pass over the input program, because each identifier occurrence can be referenced directly by at most one function. This is also linear in the size of the output program, because the output program is at least as long as the input program (they are the same apart from the additional parameters and arguments, and the reordering caused by block floating).

Lines 3–6 of *DFS* will run exactly once for each extraneous parameter. Each iteration of the **for** loop on line 4 corresponds to a unique extraneous argument in some call site or reference, and these arguments will be inserted in the output program, so the total number of iterations is linear in the size of the output program. The **for** loop on lines 17–18 is linear in the number of extraneous parameters. The rest of the code is linear in the size of the input. $\square$

5.4.5 Algorithm 1b

Algorithm 1b is an iterative version of Algorithm 1, produced by converting the recursion to iteration using an explicit stack. It is equivalent to Algorithm 1 apart from some harmless reordering due to the stack being last-in first-out. (Proof sketch of the equivalence: If we trivially refactor Algorithm 1 by moving lines 2–3 from the beginning of *DFS* to immediately before each call to *DFS*, then a call to *DFS* in Algorithm 1 with arguments $(v, f, *)$ corresponds to pushing (f, v) onto the stack in Algorithm 1b.)

The variable *stack* contains pairs (f, v) , denoting that the variable v needs to be propagated to the functions that reference f . Whenever a variable v is added to $EP[f]$, the pair (f, v) is pushed onto *stack*. All the variables are processed one after another using a single **while** loop. This allows conditions (1) and (2) from the definition of the solution to be neatly translated to a loop invariant.

Theorem 5.4.9. *The output of Algorithm 1b is the solution.*

Proof. The loop invariant of the **while** loop on line 8 is as follows. It is almost exactly the same as the conditions from the definition of the solution; the important difference is that we added $(f, v) \notin \text{stack}$ to the second clause, to relax condition (2) for the pairs currently in the stack.

```

1   $EP :=$  mapping from each function to a set of extraneous parameters
2      (all sets initially empty)
3   $stack :=$  new stack
4  for  $v$  in  $all\_variables$ :
5      for  $f$  in functions where  $v$  is a referenced non-local:
6          add  $v$  to  $EP[f]$ 
7          push  $(f, v)$  onto  $stack$ 
8  while  $stack$  is not empty:
9       $(f, v) :=$  pop  $stack$ 
10     for  $g$  in functions that reference  $f$ :
11         if  $g$  is not the declaring function of  $v$  and  $v \notin EP[g]$ :
12             add  $v$  to  $EP[g]$ 
13             push  $(g, v)$  onto  $stack$ 
14 output  $EP$ 

```

Algorithm 1b

For all functions f and g and for all identifiers v , the invariant is the conjunction of the following clauses:

1. v is a referenced non-local variable of $f \Rightarrow v \in EP[f]$
2. $v \in EP[f] \wedge (f, v) \notin stack \wedge g$ references $f \wedge g$ is not the declaring function of $v \Rightarrow v \in EP[g]$
3. The only elements in EP are those required by conditions (1) and (2) from the definition of the solution.

*Proof that the invariant holds immediately before the **while** loop is entered:*

The code before the **while** loop (specifically lines 4–7) adds v to $EP[f]$ for all functions f where v is a referenced non-local variable (and only for those functions), so clauses 1 and 3 are satisfied.

The code before the **while** loop pushes (f, v) onto $stack$ whenever it adds v to $EP[f]$, so clause 2 is satisfied vacuously.

*Proof that the invariant holds at the end of each iteration of the **while** loop:*

The body of the **while** loop does not remove any identifiers from EP , so clause 1 will continue to be satisfied.

Removing the pair (f, v) from $stack$ may cause clause 2 to become false for f , but the following **for** loop (lines 10–13) will immediately restore the invariant

by adding v to $EP[g]$ for all g that reference f (excluding the declaring function of v), so at the end of the iteration clause 2 will hold for f .

Adding v to $EP[g]$ does not violate clause 2 because (g, v) is also pushed onto *stack*.

Observe that if $(f, v) \in \textit{stack}$ then $v \in EP[f]$. This follows because the algorithm always adds v to $EP[f]$ before it pushes (f, v) onto *stack*, and it never removes any identifiers from EP .

At the beginning of the iteration (f, v) was in *stack*, so $v \in EP[f]$ and condition (2) requires $v \in EP[g]$, satisfying clause 3.

Hence the invariant holds at the end of each iteration.

Therefore, the invariant will also hold at the end of the **while** loop. In addition, *stack* will be empty, so $(f, v) \notin \textit{stack}$ will be true for all pairs and hence the loop invariant will be logically equivalent to the conditions for a solution. $\square$

Theorem 5.4.10. *The time complexity of Algorithm 1b is linear in the size of the output program.*

Proof. As with Algorithm 1, we assume the availability of suitable map and set data structures (Section 5.4.6); the preprocessing step to compute the function reference graph and the referenced non-local variables of the functions is the same.

Each **push** operation corresponds to a unique extraneous parameter of some function. They will all be inserted in the output program, so the total number of **push** operations (and hence also **pop** operations) is linear in the size of the output program.

Each iteration of the **for** loop on line 10 corresponds to a unique extraneous argument in some call site or reference, and these arguments will be inserted in the output program, so the total number of iterations is linear in the size of the output program. $\square$

5.4.6 Data structure implementation

Finding map and set data structures that satisfy the assumptions of Theorem 5.4.10 is not trivial. The most practical approach is to use a hash table, offering an *average* time complexity of $O(1)$ and a worst-case time complexity of $O(n)$ for access operations. Hash tables are highly likely to perform well in

practice under some reasonable assumptions (namely that the hash function is sensible, and that the identifiers are of reasonable length and are not specifically crafted to generate collisions); but to tighten the theoretical worst-case time complexity of the algorithm, we consider additional approaches.

A prefix tree offers a per-operation time complexity linear in the length of the key, which in our case is an identifier. If the maximum length of an identifier is a constant, this becomes $O(1)$. Otherwise another approach is needed.

We now describe two ad hoc approaches with per-operation time complexity $O(1)$ that support identifiers of arbitrary length (we only assume that the size of the input program can be represented by a single machine word).

The first approach begins by converting all identifiers that occur in the input program to integer IDs in the range 0 to $|I| - 1$ (where $|I|$ is the number of bindings). This can be done using a prefix tree, with time complexity linear in the size of the input program. (IDs can later be efficiently converted back to the original identifier names by storing the original identifier names in an array indexed by the identifier ID.) A set can then be represented as an array of Booleans indexed by identifier ID, and a map can be represented as an array indexed by identifier ID, with the element being the mapped value. To allow efficient iteration, a linked list should be maintained recording the unique IDs present in the set/map; this also allows efficient clearing. The only issue is that initialisation is $O(|I|)$; to prevent this from affecting the overall running time of the algorithm, some small changes are required (discussed below).

The second approach is to augment the nodes of the abstract syntax tree (AST) with additional fields. For example, the AST node representing function definitions could be augmented with a Boolean field named *traversed* to be used by the DFS instead of the *traversed_funcs* set. This approach may be more attractive if the implementation operates on references to AST nodes and already has a mechanism for resolving identifiers to declarations. As with the first approach, linked lists are required to allow efficient traversal. Initialisation is again $O(|I|)$ per set/map, and their creation is rather inflexible because they are tied to fields.

As written, both Algorithm 1 and Algorithm 1b create a large number of sets. This is problematic when using the two approaches just described, due to

```

1  EP := mapping from each function to a list of extraneous parameters
2  stack := new stack
3  for v in all_variables:
4      for f in functions where v is a referenced non-local:
5          add v to EP[f]
6          push (f, v) onto stack
7  traversed_funcs := new set
8  v_prev := null
9  while stack is not empty:
10     (f, v) := pop stack
11     if v ≠ v_prev:
12         v_prev := v
13         clear traversed_funcs
14         for g in functions where v is a referenced non-local:
15             add g to traversed_funcs
16         for g in functions that reference f:
17             if g is not the declaring function of v and g ∉ traversed_funcs:
18                 add v to EP[g]
19                 add g to traversed_funcs
20                 push (g, v) onto stack
21 output EP

```

Algorithm 1b': Modified version of Algorithm 1b that only uses one set

the initialisation time complexity (and lack of direct support in the case of the second approach). Fortunately, it is possible to reduce this requirement to a fixed number of sets with some small changes:

(A) In Algorithm 1, instead of creating a new set in each invocation of *multi_source_DFS*, we can reuse a single set across all invocations and clear the set between calls.

(B) In Algorithm 1b, we change the values in *EP* from sets to lists; the challenge is to keep the check $v \notin EP[g]$ efficient. Observe that the pairs pushed onto the stack by lines 4–7 are grouped by variable, and that the **while** loop only pushes pairs with the same variable as the most recently popped pair. Hence all pairs with a given variable will be popped from the stack consecutively. So we can use a single set (*traversed_funcs*) to track the visited functions for the current variable, as shown in figure Algorithm 1b': at the beginning of the

while loop, if v is different from the variable of the previously-popped pair (or if this is the first iteration), we clear $traversed_funcs$ and add the functions where v is a referenced non-local to $traversed_funcs$; then later when we add v to $EP[g]$ we also add g to $traversed_funcs$. Then the check $v \notin EP[g]$ can be replaced with $g \notin traversed_funcs$.

(C) The preprocessing step also requires small changes. The natural way to compute the reverse function reference graph is to use a map from each function to a set of functions that reference it, which is problematic. But we can use lists instead of sets, and efficiently exclude duplicate references using a single set as follows: when processing a function f , deduplicate the functions it references by adding them to the set, then for each g in the set add f as a function that references g ; then clear the set. Likewise, the sets of functions where each variable is a referenced non-local can be represented using lists, and duplicate functions can be efficiently excluded using a single set by deduplicating the referenced non-local variables when processing each function.

With these changes, using either approach, the number of maps and sets created by Algorithm 1 and Algorithm 1b is a constant, so the total initialisation time is $O(|I|)$. The time complexity of the **clear** operations is amortized by the insertions. The proofs of [Theorem 5.4.8](#) and [Theorem 5.4.10](#) cover the rest of Algorithm 1 and Algorithm 1b respectively, so their overall time complexity is linear in the size of the output program.

5.4.7 Optimality and improvements

The time complexity of Algorithm 1 and Algorithm 1b is optimal *if the lifted program is to be output*. If the lifted program does *not* need to be output, then the lower bound imposed by the size of the lifted program is removed, admitting algorithms that have a better time complexity on some cases.

For example, the representation of the lifted program could be made more compact by sharing identical extraneous parameter/argument lists. Consider the first example in [Appendix A](#); the total number of extraneous parameters is $\Theta(n^2)$, but the extraneous parameter lists of all the functions are identical (and hence so are the extraneous argument lists), so a compact representation of the lifted program that shares these lists would have size $\Theta(n)$ rather than $\Theta(n^2)$. In that particular example, it is possible to compute this representation

in $O(n)$ using a simple top-down recursive traversal, because the call graph is acyclic and the required set union operations can be performed trivially with a little care (the union of a set with itself is itself, and set equality can be checked efficiently using a heuristic test such as reference equality).

A nice property of Algorithm 1 and Algorithm 1b is that they are able to start from any state; the only information they require is which variables still have to be propagated from which functions (and to which functions). This leads to an elegant hybrid algorithm: first propagate variables using a top-down recursive traversal of the function reference graph, skipping edges that create cycles; then run Algorithm 1 starting from the edges that were skipped to complete the propagation. This hybrid algorithm will be faster on some cases (such as the first example from [Appendix A](#)) because some of the set union operations can be performed trivially.

Some additional optimisations are possible for the set union operations. If the number of variables is small (say ≤ 64), the sets can be represented as bitmasks and set union can be performed using a single bitwise instruction. For larger sets, the union of sets of size m and n would be⁸ $O(m+n)$ using hash sets (average case) or sorted lists⁹, and this is sufficient for keeping the overall time complexity linear in the size of the output program; however the time complexity of set union can be improved to $O(m \log(n/m + 1))$ where $m \leq n$ using persistent balanced trees [3], and that is the optimal time complexity for the general comparison-based merging problem [7].

Another small optimisation is to observe that top-level functions never require extraneous parameters.

A big remaining opportunity for improving the time complexity is to take advantage of SCCs—coalescing them reduces the size of the function reference graph. The algorithm we described at the end of [Section 5.2.1](#) takes advantage of SCCs, but has inefficiencies. We briefly describe an optimised version of that algorithm; the time complexity is at most linear in the size of the lifted program, and in some cases it is even faster. The optimisations are to restrict the traversals to subgraphs in order to skip functions that do not require

⁸Assuming identifiers are converted to integer IDs (see [Section 5.4.6](#)), to allow $O(1)$ comparisons.

⁹The initial referenced non-local variables of each function can be sorted in linear time using a prefix tree.

extraneous parameters, and to coalesce SCCs incrementally by processing the functions in a specific order.

1. Create a reverse function reference graph G , initially empty, whose nodes are sets of functions (representing components).
2. For each function f , add the set $\{f\}$ as a node of G .
3. Perform a post-order traversal of the AST. To process a function f :
 - (a) Let V be the set of variables whose declaring function is f .
 - (b) Compute the set of nodes N whose functions require one or more of the variables in V as extraneous parameters, by performing a depth-first traversal of G (that is, propagating to callers) starting from the nodes containing functions that reference the variables in V .
 - (c) Compute the SCCs of the subgraph of G induced by N . (This can be done using Tarjan’s algorithm during the same traversal as the previous step.)
 - (d) Coalesce the corresponding components in G (and in N); see the implementation notes below.
 - (e) Perform a traversal of the subgraph of G induced by N in topological order (e.g. using Kahn’s algorithm) to compute the extraneous parameters for the variables in V . Compute the extraneous parameters of a node C as the union of the variables in V referenced by functions in C and the extraneous parameters computed for the predecessors (callees) of C .
 - (f) Add edges from nodes containing functions that reference f to the node $\{f\}$.
4. The final extraneous parameters of a function g are the union of the extraneous parameters computed for nodes containing g across the traversals (see the implementation notes below).

To coalesce nodes while allowing the node containing a function to be looked up efficiently, a disjoint-set data structure should be used. When coalescing

multiple nodes into one, their outgoing edges should be combined eagerly; the incoming edges can be resolved lazily using the disjoint-set data structure.

We note that the problem of incrementally maintaining SCCs as edges are added to a graph has been studied in the literature [28], but in our case it is not necessary to use a specialised incremental SCC algorithm—we need to traverse the subgraph regardless (to compute the extraneous parameters in step 3e), so using Tarjan’s algorithm does not impact the overall time complexity.

There are a number of options for computing the unions in the final step. An efficient and elegant approach is to perform the union lazily: when coalescing nodes, record the coalesced node as the “parent” of the original nodes; to enumerate the extraneous parameters of a function g , enumerate those computed for the node $\{g\}$, followed by those computed for the parent node of $\{g\}$, and so on. There is no need to perform deduplication, because the extraneous parameters computed in different traversals are disjoint.

It can be shown that the time complexity of this algorithm is at most linear in the size of the lifted program, using similar reasoning to Algorithm 1 (Tarjan’s algorithm, Kahn’s algorithm, and coalescing are linear-time; disjoint-set operations are near-constant but we treat them as constant time for simplicity).

Giving a more precise time complexity is difficult because it depends on the input; however, we can say that this algorithm takes good advantage of SCCs and can take good advantage of fast set union operations in step 3e. We conjecture that the time complexity of this algorithm is at least as fast as all known algorithms on all cases, including when the lifted program does not need to be output.

5.5 Specification

5.5.1 Fischbach–Hannan lifting specification

To prove that our solution produces a valid lifting, we turn to the Fischbach–Hannan specification.

We briefly describe the specification here, and refer the reader to [19, §3.2] for a more complete discussion. We include the rules of that specification in Figure 5.1, and also in Figure 5.6 for comparison with our Lean translation.

The judgement $\Gamma \triangleright e : \tau \Rightarrow e'$ states that the input expression e can be lifted

$$\begin{array}{c}
\frac{\Gamma(x) = \tau}{\Gamma \triangleright x : \tau \Rightarrow x} \text{ (var)} \qquad \frac{\Sigma(c) = \tau}{\Gamma \triangleright c : \tau \Rightarrow c} \text{ (const)} \\
\\
\frac{\Gamma\{y : \tau_1\} \triangleright e : \tau \Rightarrow e' \quad y \notin \text{dom}(\Gamma) \quad FV(\tau_1 \rightarrow \tau) \subseteq \text{dom}(\Gamma)}{\Gamma \triangleright \lambda y. e : \tau_1 \rightarrow \tau \Rightarrow \lambda y. e'} \text{ (abs)} \\
\\
\frac{\Gamma(x) = \tau_1 \quad \Gamma \triangleright \lambda y. e : \tau \Rightarrow \lambda z. e'}{\Gamma \triangleright \lambda y. e : \{x\}_{\tau_1} \rightarrow \tau \Rightarrow \lambda x. \lambda z. e'} \text{ (lift-abs)} \\
\\
\frac{\Gamma \triangleright e_1 : \tau_1 \rightarrow \tau \Rightarrow e'_1 \quad \Gamma \triangleright e_2 : \tau_1 \Rightarrow e'_2}{\Gamma \triangleright e_1 @ e_2 : \tau \Rightarrow e'_1 @ e'_2} \text{ (app)} \\
\\
\frac{\Gamma \triangleright e : \{x\}_{\tau_1} \rightarrow \tau \Rightarrow e' \quad \Gamma(x) = \tau_1}{\Gamma \triangleright e : \tau \Rightarrow e' @ x} \text{ (lift-app)} \\
\\
\frac{\Gamma^* = \Gamma\{\overline{f_i : \tau_i}\} \quad \Gamma^* \triangleright e_i : \tau_i \Rightarrow e'_i \quad \Gamma^* \triangleright e : \tau \Rightarrow e' \quad f_i \notin \text{dom}(\Gamma) \quad FV(\tau) \subseteq \text{dom}(\Gamma) \quad i \in \{1..n\}}{\Gamma \triangleright \text{letrec } \overline{f_i = e_i} \text{ in } e : \tau \Rightarrow \text{letrec } \overline{f_i = e'_i} \text{ in } e'} \text{ (letrec)}
\end{array}$$

Figure 5.1: The Fischbach–Hannan lifting specification, from [19, §3.2]

to the expression e' of type τ , where Γ is a context mapping identifiers to types. Γ is initially empty for the top-level expression; the notation $\Gamma\{\overline{f_i : \tau_i}\}$ extends Γ with new entries. Σ is a pre-defined signature mapping each constant to its type; it is assumed that Σ does not contain any singleton types. Extraneous parameters and arguments are inserted via the rules (lift-abs) and (lift-app) respectively. The type information recorded in Γ is used to ensure the extraneous arguments correspond to the extraneous parameters (in both type and identifier name). The rules include conditions such as $y \notin \text{dom}(\Gamma)$ to forbid shadowing, and $FV(\tau) \subseteq \text{dom}(\Gamma)$ to prevent extraneous parameters from escaping their scope, to avoid unsoundness (see the examples in [19, §2]). The types used in the rules need not match the original types in e —for example, the type τ_1 of y in the rule (abs) need not match the original type of y in e ; this allows

more general liftings where extraneous arguments are passed indirectly (see the examples in [19, §3.3]).

As stated in [19, §3.2], in the rules (abs) and (app) of the lifting specification, τ_1 cannot be a singleton type. This is because the grammar of τ does not allow “bare” singleton types. This restriction is subtle but critical for the specification’s soundness. Specifically, if the grammar allowed $\{e\}_\tau$ as a term of τ , then unsound liftings such as the following would be permitted by the specification:

$$\{x : \mathbb{N}\} \triangleright \lambda y.0 : \mathbb{N} \Rightarrow (\lambda y.0) @ x$$

This lifting is unsound because the input expression has type $\mathbb{N} \rightarrow \mathbb{N}$ while the lifted expression has type $\mathbb{N}$ (an unrelated type). The lifting can be derived using the rule (lift-app), where the premise $\{x : \mathbb{N}\} \triangleright \lambda y.0 : \{x\}_{\mathbb{N}} \rightarrow \mathbb{N} \Rightarrow \lambda y.0$ is derived using the rule (abs) with $\tau_1 = \{x\}_{\mathbb{N}}$ (a bare singleton type). Disallowing bare singleton types in the grammar of τ prevents such unsound liftings.

Proving that our solution satisfies the Fischbach–Hannan specification is desirable because then the theorems [19, Theorem 2 (Type Correctness)] and [19, Theorem 4 (Operational Correctness)] apply to our lifting. However, we also wish to prove that our solution gives the minimal extraneous parameters within the class of liftings that we are interested in. So we give a specification for “Johnsson-style” lifting, and prove that this is a restriction of the Fischbach–Hannan specification (Theorem 5.6.1); then we prove that the solution satisfies our specification for Johnsson-style lifting.

5.5.2 Johnsson-style lifting

In Figure 5.2 we give a specification of “Johnsson-style” lifting, where each function is assigned a set of extraneous parameters and corresponding extraneous arguments are inserted at each occurrence of the function’s name. This specification is slightly more general than Johnsson’s algorithm:

- We leave the choice of extraneous parameters unspecified.
- We allow function names to be inserted as extraneous parameters. This is unnecessary when all functions are to be block-floated to the top-level scope, because in that case they will all be in scope; but allowing it simplifies the specification and permits selective lifting. However, we

$$\begin{array}{c}
\frac{\Lambda(x) = \theta}{\Lambda \triangleright x \Rightarrow x @ \theta_1 @ \dots @ \theta_n} \text{ (var)} \qquad \frac{}{\Lambda \triangleright c \Rightarrow c} \text{ (const)} \\
\\
\frac{\Lambda\{y : \emptyset\} \triangleright e \Rightarrow e' \quad y \notin \text{dom}(\Lambda) \quad \theta \subseteq \text{dom}(\Lambda)}{\Lambda \triangleright {}^\theta \lambda y. e \Rightarrow (\lambda \theta_1. \dots \lambda \theta_n. \lambda y. e') @ \theta_1 @ \dots @ \theta_n} \text{ (abs)} \\
\\
\frac{\Lambda \triangleright e_1 \Rightarrow e'_1 \quad \Lambda \triangleright e_2 \Rightarrow e'_2}{\Lambda \triangleright e_1 @ e_2 \Rightarrow e'_1 @ e'_2} \text{ (app)} \\
\\
\Lambda^* = \Lambda\{\overline{f_i : ep(e_i)}\} \\
\frac{\Lambda^* \triangleright f_i = e_i \Rightarrow f_i = e'_i \quad \Lambda^* \triangleright e \Rightarrow e' \quad f_i \notin \text{dom}(\Lambda) \quad \forall x \in ep(e_j), x \in \{\overline{f_i}\} \supset \neg \text{abs}(\{\overline{f_i : e_i}\}(x)) \quad i, j \in \{1..n\}}{\Lambda \triangleright \text{letrec } \overline{f_i = e_i} \text{ in } e \Rightarrow \text{letrec } \overline{f_i = e'_i} \text{ in } e'} \text{ (letrec)} \\
\\
\frac{\Lambda \triangleright e \Rightarrow e' \quad \neg \text{abs}(e)}{\Lambda \triangleright f = e \Rightarrow f = e'} \text{ (var-binding)} \\
\\
\frac{\Lambda\{y : \emptyset\} \triangleright e \Rightarrow e' \quad y \notin \text{dom}(\Lambda) \quad \theta \subseteq \text{dom}(\Lambda)}{\Lambda \triangleright f = {}^\theta \lambda y. e \Rightarrow f = \lambda \theta_1. \dots \lambda \theta_n. \lambda y. e'} \text{ (fun-binding)} \\
\\
ep({}^\theta \lambda y. e) = \theta \\
ep(e) = \emptyset
\end{array}$$

Figure 5.2: Johnsson-style lifting

do not allow function names declared in the same **letrec** expression to be inserted as extraneous parameters, because that can lead to infinite types.

- Johnsson's algorithm does not support anonymous functions; instead, it is assumed that each anonymous function $\lambda \dots$ is replaced with the equivalent expression **letrec** $f = \lambda \dots$ in f where f is a fresh identifier. We have chosen to support anonymous functions directly; their extraneous arguments are inserted at the occurrence of the anonymous function,

producing an equivalent result.

- We exclude the block-floating step (following [19]).

Our specification loosely resembles the two-phase specification in [19, Fig. 5 and Fig. 6]. That specification is similar to Johnsson’s algorithm, but it does not support variable bindings and it allows anonymous functions (however it does not eliminate their free variables).

The judgement $\Lambda \triangleright e \Rightarrow e'$ given in Figure 5.2 states that the input expression e is lifted to the expression e' . Each function in the input expression is required to be annotated with an ordered set θ specifying the chosen extraneous parameters, notated ${}^\theta\lambda y.e$ (based on the notation from [19]). For multi-parameter functions, only the first λ should have an annotation; the rest should be implicitly $\emptyset$ (but we do not enforce this). Λ maps function names to their annotated extraneous parameters; variables are mapped to $\emptyset$.

We use the symbol $\supset$ to denote logical implication, and the notation $\text{abs}(e)$ denotes that e is an abstraction. In the rule (letrec), the premise $\forall x \in ep(e_j), \dots$ ensures that function names declared in the current **letrec** expression are not used as extraneous parameters. (If f_i are assumed to be distinct, that premise is equivalent to $ep(e_i) \cap \{f_j \mid \text{abs}(e_j)\} = \emptyset$.)

5.5.3 Complete lifting

We are interested in block-floating all functions (including anonymous functions) to the top-level scope. For the resulting program to be valid, the lifted program must not contain any non-local variable references prior to block-floating. This is equivalent to the definition of a supercombinator from [64] except that free function references are allowed (because after block-floating all functions will be in scope).

We refer to a lifting where the lifted program e' has this property as a “complete lifting”, formalised by the judgement $\triangleright SC(e')$ given in Figure 5.3 (the name SC is a reference to the closely-related term “supercombinator”). Ψ maps each identifier to one of $\{G, L, N\}$ denoting a global variable, local variable, or non-local variable respectively. Function names are mapped to G because after block-floating they will be global.

$$\begin{array}{c}
\frac{\Psi[x] \in \{G, L\}}{\Psi \triangleright SC(x)} \text{ (var)} \qquad \frac{}{\Psi \triangleright SC(c)} \text{ (const)} \\
\\
\frac{\Psi' = \{y : N \mid \Psi[y] = L \text{ else } \Psi[y] \mid y \in \text{dom}(\Psi)\} \quad \Psi^* = \Psi' \{\overline{x_i : L}\} \quad \Psi^* \triangleright SC(e) \quad \neg \text{abs}(e) \quad i \in \{1..n\}}{\Psi \triangleright SC(\lambda x_1. \dots \lambda x_n. e)} \text{ (abs)} \\
\\
\frac{\Psi \triangleright SC(e_1) \quad \Psi \triangleright SC(e_2)}{\Psi \triangleright SC(e_1 @ e_2)} \text{ (app)} \\
\\
\frac{\Psi^* = \Psi \{\overline{f_i : G \text{ if } \text{abs}(e_i) \text{ else } L}\} \quad \Psi^* \triangleright SC(e_i) \quad \Psi^* \triangleright SC(e) \quad i \in \{1..n\}}{\Psi \triangleright SC(\text{letrec } \overline{f_i = e_i} \text{ in } e)} \text{ (letrec)} \\
\\
\frac{\Psi^* = \{\overline{f_i : G}\} \quad \Psi^* \triangleright SC(e_i) \quad \Psi^* \triangleright SC(e) \quad i \in \{1..n\}}{\Psi \triangleright SC(\text{letrec } \overline{f_i = e_i} \text{ in } e)} \text{ (program)}
\end{array}$$

Figure 5.3: Complete lifting

The rule (program) specifies whether an entire program (represented as a `letrec` expression) is a complete lifting; it is almost the same as the rule (letrec) but it interprets variable bindings as global variables.

5.6 Theorems

This section consists of proofs that together show the solution gives the minimal extraneous parameters for a complete Johnsson-style lifting. We have chosen to do some of the proofs in Lean, while others as traditional “pen and paper” proofs. This choice is driven by practical considerations—some obvious proofs can be difficult to formalise in theorem provers due to the level of detail required.

Theorem 5.6.1. (*Johnsson-style lifting satisfies the Fischbach–Hannan specification*). *Let e be an expression annotated with extraneous parameters. Let $\Gamma = \{\}$ and $\Lambda = \{\}$. If $\Gamma \vdash e : \tau$ and $\Lambda \triangleright e \Rightarrow e'$ then $\Gamma \triangleright e : \tau \Rightarrow e'$.*

Proof. We have proven this in Lean. □

This theorem is similar to [19, Theorem 5], but it is more general: it applies to any extraneous parameters permitted by our specification of Johnsson-style lifting, as opposed to the particular extraneous parameters chosen by the algorithm from [19]. Our Lean proof uses the ideas from the “pen and paper” proof in [19] (“corresponding contexts” and “closed contexts”).

The trickiest part of the proof is the handling of **letrec** expressions, since the computation of the lifted types is quasi-circular: the lifted type of a function binding includes the lifted types of its extraneous parameters, which may be variables from the same **letrec** expression; mathematically, we need to compute a fixpoint $\tau s = \text{liftedTypes}(\Gamma, \text{bindings}, \tau s)$. The premise $\forall x \in ep(e_j), \dots$ in the rule (letrec) of Figure 5.2 ensures there is no true circularity by forbidding function names declared in the same **letrec** expression from being used as extraneous parameters.

We compute τs in three steps:

1. Compute the lifted types of variable bindings (these do not depend on other bindings), and insert placeholders for the types of function bindings; call the result τs_p .
2. Compute $\tau s := \text{liftedTypes}(\Gamma, \text{bindings}, \tau s_p)$. That is, compute the true lifted types using the types containing placeholders.
3. Prove $\tau s = \text{liftedTypes}(\Gamma, \text{bindings}, \tau s)$ using the premise $\forall x \in ep(e_j)$, thereby closing the loop.

Proposition 5.6.2. (*Well-nesting.*) *Let e be a closed expression with no shadowing. For all functions f and g and for all identifiers x , if g references f and x is in scope at the definition of f , then x is also in scope at the reference to f in g (and resolves to the same binding).*

Proposition 5.6.3. (*Scope extension.*) *For all functions f and for all identifiers x and y ,¹⁰ if y is referenced directly by f , and x is in scope at that reference, and f is not the declaring function of the binding of x , then x is also in scope at the definition of f (and resolves to the same binding).*

¹⁰ y is unimportant, it only serves to establish a location inside f .

Theorem 5.6.4. *(The extraneous parameters in the solution are in scope). Let e be a closed expression with no shadowing and let EP be the solution. For all functions f and for all identifiers $v \in EP[f]$, v is in scope at the definition of f (and is bound to a non-global variable).*

Proof. We proceed by induction.

Consider an identifier v which is in $EP[f]$ by condition (1); so v is a referenced non-local variable of f . By Proposition 5.6.3 (with $x = v$ and $y = v$), v is in scope at the definition of f .

Now consider an identifier v which is in $EP[g]$ by condition (2); so $v \in EP[f]$, g references f , and g is not the declaring function of v . By the inductive hypothesis, v is in scope at the definition of f . By Proposition 5.6.2, v is in scope at the reference to f in g . By Proposition 5.6.3 (with $y = f$ and $f = g$), v is in scope at the definition of g . $\square$

Definition 5.6.5. The extraneous parameter scoping premises of the Johnsson-style lifting specification are the premise $\theta \subseteq \text{dom}(\Lambda)$ in the rules (abs) and (fun-binding) and the premise $\forall x \in ep(e_j), \dots$ in the rule (letrec).

Corollary 5.6.6. *Let e be a closed expression with no shadowing. Then the solution satisfies the extraneous parameter scoping premises.*

Proof. The premise $\theta \subseteq \text{dom}(\Lambda)$ in the rules (abs) and (fun-binding) asserts that the extraneous parameters of a function are in scope at the function's definition, so it is satisfied. The premise $\forall x \in ep(e_j), \dots$ in the rule (letrec) is satisfied because the extraneous parameters in the solution are all variables, not function names. $\square$

Theorem 5.6.7. *(Sufficient conditions for Johnsson-style lifting). Let e be a closed expression with no shadowing, annotated with extraneous parameters that satisfy the extraneous parameter scoping premises. Then there exists an expression e' such that $\{\} \triangleright e \Rightarrow e'$.*

Proof. We have proven this in Lean, by constructing e' and using the premises of the theorem to prove the corresponding premises of the Johnsson-style lifting specification. $\square$

Corollary 5.6.8. *The solution satisfies Johnsson-style lifting.*

Theorem 5.6.9. *(The solution satisfies SC). Let e^s be a closed expression with no shadowing, annotated with the extraneous parameters given by the solution. If $\{\} \triangleright e^s \Rightarrow e'$ then $\triangleright SC(e')$.*

Proof. Observe that $\text{dom}(\Psi) = \text{dom}(\Lambda)$, since extraneous parameters are only inserted if they are already in scope.

By Theorem 5.6.4, the inserted extraneous parameters only shadow non-global variables; so the resolution of global variables and of functions is the same in e^s and e' .

To prove $\triangleright SC(e')$, we need to prove $\Psi[x] \in \{G, L\}$ for each identifier occurrence x in e' . We have proven in Lean that e' is closed; so $x \in \text{dom}(\Psi)$, and we need only prove $\Psi[x] \neq N$. Assume for contradiction that there exists an identifier occurrence x such that $\Psi[x] = N$.

If x is not an extraneous argument, then $\Psi[x] = N$ implies x is enclosed by some function f with $x \notin EP[f]$. This contradicts condition (1) from the definition of the solution.

Otherwise x is an extraneous argument of some function f , and $\Psi[x] = N$ implies the reference to f is enclosed by some function g with $x \notin EP[g]$. If g is not the declaring function of x (in e^s), we have a contradiction from condition (2). Otherwise g is the declaring function of x ; and by Proposition 5.6.2 (and Theorem 5.6.4), the binding of x is in scope at the reference to f in g , contradicting $\Psi[x] = N$. $\square$

Theorem 5.6.10. *(The solution is minimal). Let e be a closed expression with no shadowing, annotated with the extraneous parameters given in a mapping A . Suppose $\{\} \triangleright e \Rightarrow e'$. Let EP be the solution. If $\triangleright SC(e')$ then $EP[f] \subseteq A[f]$ for all functions f .*

Proof. We show $v \in EP[f]$ implies $v \in A[f]$ by induction over EP .

Observe that if an identifier occurrence in e resolves to a non-global variable, then the corresponding occurrence in e' must also resolve to a non-global variable (since the inserted extraneous parameters only create parameter bindings), and recall that $\text{dom}(\Psi) = \text{dom}(\Lambda)$.

Consider $v \in EP[f]$ by condition (1); so v is a referenced non-local variable of f in e . By the observation, v must resolve to a non-global variable in e' , so $\Psi[v] = L$ (from $\triangleright SC(e')$). The declaring function of v is not f in e , so v must be in $A[f]$.

Now consider $v \in EP[g]$ by condition (2); so $v \in EP[f]$, g references f , and g is not the declaring function of v . By induction, $v \in A[f]$. Since g references f and $v \in A[f]$, v will be inserted as an extraneous argument at the reference to f inside g in e' . By [Theorem 5.6.4](#), v is in scope at the definition of g and resolves to a non-global variable in e , and hence also in e' . So $\Psi[v] = L$. The declaring function of v is not g in e , so v must be in $A[g]$. $\square$

Corollary 5.6.11. *The solution gives the minimal extraneous parameters for a complete Johnsson-style lifting.*

5.7 Lean implementation

To allow us to machine-check some of our proofs we translated the specifications to Lean 4, which is both an interactive theorem prover based on the Calculus of Inductive Constructions and a purely functional programming language influenced by Haskell [16]. The source code of the translations, some of the proofs, an implementation of Algorithm 1, and examples is published at [48]; archival artifacts are available at [47].

In this section we show and explain some extracts from the Lean definitions. The definition of `Expr` maps to e directly:

```
-- Expressions.  e ::= c | x | λx.e | e1@e2 | letrec  $\overline{f_i = e_i}$  in e  -/
inductive Expr where
  | const : String → Expr
  | var : String → Expr
  | abs : String → Expr → Expr
  | app : Expr → Expr → Expr
  | letrec : List (String × Expr) → Expr → Expr
deriving BEq, Repr
```

`Ty` maps to the version of τ without singleton types, and `Tys` maps to the extended version of τ that uses singleton types. The separation allows us to explicitly allow or forbid singleton types. As mentioned earlier, we restrict the expression e in singleton types to be an identifier, and we are careful to disallow “bare” singleton types in `Tys`.

```
-- Types.  τ ::= ι | τ → τ  -/
inductive Ty where
```

```

| base : String → Ty
| arrow : Ty → Ty → Ty
deriving DecidableEq, Repr

/-- Types, extended using singleton types.  $\tau ::= \iota \mid \tau \rightarrow \tau \mid \{e\}_\tau \rightarrow \tau$  -/
inductive Tys where
| base : String → Tys
| arrow : Tys → Tys → Tys
| singlarrow : String → Tys → Tys → Tys
deriving DecidableEq, Repr

```

We use `deriving DecidableEq` for `Ty` to allow our type checker to test whether elements of `Ty` are equal, and likewise for `Tys`.

The name `Ty` was chosen to avoid clashing with Lean’s `Type` keyword. For the same reason we later use `S` instead of Σ for the mapping from pre-defined constants to their types.

Additional definitions are given in [Figure 5.4](#).

`Sig` is the type of Σ , and `Ctx` is the type of Γ (with a variant `Ctxs` that allows singleton types).

We use `Lean.AssocList` to represent these mappings; `lookup` is not computationally efficient, but that is not a concern for proofs. Another option would have been `Std.HashMap`, which has a better average time complexity and supports more operations, but is more difficult to use in proofs due to weaker support for simplification and definitional equality. There is also `Lean.Data.Trie`, which has good time complexity but only supports a few operations and is difficult to work with (we use it in the implementation of our algorithm).

Note that `Lean.AssocList` keeps duplicate keys when entries are inserted using `insertNew` (but not when using `insert`). Operations such as `find?` return the most recent value inserted for a key, but older values are visible to operations that iterate over the entries such as `any`. This is illustrated by the following example:

```

def m := (Lean.AssocList.nil.insertNew 1 5).insertNew 1 6
#eval m.find? 1 --> some 6
#eval m.any (fun k v => (k, v) == (1, 5)) --> true

```

The presence of older values is not an issue for us, because we only use operations such as `find?` that ignore older values. We have chosen to keep older values,

```

import Lean.Data.AssocList
import LambdaLifting.AssocListExtras
open Lean (AssocList)

/- [The definitions of Expr, Ty, and Tys are given inline. They should be inserted here
    to produce complete source code.] -/

/-- Converts a `Ty` to a `Tys`. -/
def Ty.toTys : Ty → Tys
| .base s => .base s
| .arrow τ1 τ2 => .arrow τ1.toTys τ2.toTys

/-- The type of pre-defined signatures (Σ) mapping each constant to its `Ty`. -/
abbrev Sig := AssocList String Ty

/-- Typing context (Γ) mapping identifiers to types. -/
abbrev Ctx := AssocList String Ty
abbrev Ctxs := AssocList String Tys

/-- Safe alternative to `List.zip` that ensures the arguments have the same length. -/
def zip' {α β} {n : Nat} (as : Vector α n) (bs : Vector β n) : List (α × β) :=
  (as.zip bs).toList

abbrev dom := @AssocList.keys -- Domain (identifiers) of a context.
abbrev _root_.Lean.AssocList.extend := @AssocList.insertManyNew -- Extend a context.

/-- Convenience wrapper for `extend` for use in `letrec_rule`. -/
def _root_.Lean.AssocList.extend' {α β} {n : Nat}
  (m : AssocList α β) (as : Vector α n) (bs : Vector β n) :=
  m.extend (zip' as bs)

/-- Set of identifiers appearing as singleton types in τ. -/
-- For simplicity, we don't erase duplicates.
def FV : Tys → List String
| .base _ => []
| .arrow τ1 τ2 => FV τ1 ++ FV τ2
| .singarrow x τ1 τ2 => [x] ++ FV τ1 ++ FV τ2

```

Figure 5.4: Additional basic definitions

to make scope shadowing visible when mappings are displayed for debugging, and because `insertNew` is easier to reason about than `insert` (which contains a conditional).

The operations supported by `Lean.AssocList` are not comprehensive, so we extend it with some basic operations (not shown here):

- we define the operations `keys` and `insertManyNew`;
- we define instances of the type classes `GetElem` and `GetElem?` to support the operators `[]`, `[]?`, and `[]!`
- we define instances of the type classes `Insert` and `Singleton` to support Lean’s `{a, b, c}` syntax (we define them in terms of `insertNew` to respect duplicates and because `insert` requires `BEq`).

We also define the aliases `dom` and `extend` for `AssocList` operations to match the terminology used in the specification.

The premises of the form $\Gamma^* = \Gamma\{\dots\}$ are difficult to translate to Lean, because `let` is awkward to use within type declarations. Instead, we inline the right-hand side in the places where Γ^* is used. The function `extend'` helps make these inlined occurrences more concise.

We use Lean’s `Vector` type to ensure that lists such as `fs` and `ts` have the same length, because `List.zip` silently discards excess elements.

We use `List` to represent sets in FV because it is the most convenient way to represent a set, although it is not computationally efficient. We do not need to erase duplicate elements because Lean’s $\subseteq$ operator ignores duplicates.

Figure 5.5 shows the Lean translation of the type system from [19].

The inductive type `TypeJudgement S  $\Gamma$  e  $\tau$` represents the typing judgement $\Gamma \vdash e : \tau$ with $\Sigma = S$. We also declare a Lean syntax notation (“mixfix operator”) to match the formal notation; the Lean notation differs in that it explicitly mentions `S`, to allow different examples to use different pre-defined constant mappings.

In Lean terminology, the argument `S` of `TypeJudgement` is a *parameter* and is the same across the uses of `TypeJudgement` within the declaration; the remaining arguments are *indices* and vary across the uses [8].

We declare `TypeJudgement` as a Lean `Type` rather than `Prop` (proposition) to allow us to extract the types of subexpressions from a `TypeJudgement`, something

```

notation:26 "(" S ", " Γ ")" " ⊢ " e " : " τ:26 =>
  $(Lean.mkIdent `TypeJudgement) S Γ e τ

/-- Type system judgement, notated as `(S, Γ) ⊢ e : τ`. -/
inductive TypeJudgement (S : Sig) : Ctx → Expr → Ty → Type where

| var_rule (Γ : Ctx) (x : String) (τ : Ty) :
  Γ[x]? = some τ →
  (S, Γ) ⊢ .var x : τ
  
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$


| const_rule (Γ : Ctx) (c : String) (τ : Ty) :
  S[c]? = some τ →
  (S, Γ) ⊢ .const c : τ
  
$$\frac{\Sigma(c) = \tau}{\Gamma \vdash c : \tau}$$


| abs_rule (Γ : Ctx) (x : String) (e : Expr) (τ1 τ : Ty) :
  (S, Γ.extend [(x, τ1)] ⊢ e : τ →
  (S, Γ) ⊢ .abs x e : τ1.arrow τ
  
$$\frac{\Gamma\{x : \tau_1\} \vdash e : \tau}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau}$$


| app_rule (Γ : Ctx) (e1 e2 : Expr) (τ1 τ : Ty) :
  (S, Γ) ⊢ e1 : τ1.arrow τ →
  (S, Γ) ⊢ e2 : τ1 →
  (S, Γ) ⊢ e1.app e2 : τ
  
$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 @ e_2 : \tau}$$


| letrec_rule (Γ : Ctx) (n : Nat)
  (fs : Vector String n) (es : Vector Expr n)
  (τs : Vector Ty n)
  (e : Expr) (τ : Ty) :
  (∀ i, ( _ : i < n) →
    (S, /-Γ*-/ Γ.extend' fs τs) ⊢ es[i] : τs[i]) →
  (S, /-Γ*-/ Γ.extend' fs τs) ⊢ e : τ →
  (S, Γ) ⊢ .letrec (zip' fs es) e : τ
  
$$\frac{\begin{array}{l} \Gamma^* = \Gamma\{\overline{f_i : \tau_i}\} \\ \Gamma^* \vdash e_i : \tau_i \\ \Gamma^* \vdash e : \tau \quad i \in \{1..n\} \end{array}}{\Gamma \vdash \text{letrec } \overline{f_i = e_i} \text{ in } e : \tau}$$


```

Figure 5.5: Type system

that cannot be done with `Prop` due to the run-time irrelevance (erasure) of propositions in Lean [86]. If there are places where a proposition is desired, one can use `Nonempty (TypeJudgement ...)`.

The notation declaration requires some explanation. Lean macros are automatically *hygienic*, meaning identifiers are resolved in the context of the macro declaration rather than its use [43]. This is generally a good thing because it prevents unexpected identifier capture, but in our case it creates problems with ordering: if the notation is declared before the declaration of `TypeJudgement`, then the identifier `TypeJudgement` cannot be resolved; and if the notation is declared after `TypeJudgement`, then the notation is not available for use inside the declaration of `TypeJudgement`.

The workaround we have used is to declare the notation before `TypeJudgement`, but use `$(Lean.mkIdent `TypeJudgement)` rather than `TypeJudgement` to delay the resolution of the identifier (effectively disabling macro hygiene). This has the disadvantage of making the name `TypeJudgement` susceptible to identifier capture. One possible improvement is to use a fully-qualified name, for example ``_root_.SomeNamespace.TypeJudgement`, but even that is not foolproof because Lean permits `_root_` to be shadowed, so it is not guaranteed to refer to the root namespace. The most robust solution is to declare two versions of the notation: an unhygienic version for use inside the declaration of `TypeJudgement`, and a hygienic version for use elsewhere. The first version can be marked `local` inside a section so that the two notations will not cause ambiguity, as follows:

```
section SomeSection
-- Unhygienic notation for internal use.
local notation:26 "(" S " ", " Γ ")" " " ⊢ " e " : " τ:26 =>
  $(Lean.mkIdent `TypeJudgement) S Γ e τ

inductive TypeJudgement (S : Sig) : ... → Type where
  ... (S, Γ) ⊢ e : τ ...
end SomeSection

-- Hygienic notation for external use.
notation:26 "(" S " ", " Γ ")" " " ⊢ " e " : " τ:26 => TypeJudgement S Γ e τ
```

The number 26 is the operator precedence, and was chosen because it is the lowest precedence that is higher than the precedence of Lean’s “ $\rightarrow$ ” operator.

```

notation:26 "(" S " , " Γ ")" " ▷ " e " : " τ " ⇒ " e':26 =>
  $(Lean.mkIdent `LiftJudgement) S Γ e τ e'

/-- Parameter lifting judgement, notated as `(S, Γ) ▷ e : τ ⇒ e'`. -/
inductive LiftJudgement (S : Sig) : Ctxs → Expr → Tys → Expr → Prop where

| var_rule (Γ : Ctxs) (x : String) (τ : Tys) :
  Γ[x]? = some τ →
  (S, Γ) ▷ .var x : τ ⇒ .var x
  
$$\frac{\Gamma(x) = \tau}{\Gamma \triangleright x : \tau \Rightarrow x} \text{ (var)}$$


| const_rule (Γ : Ctxs) (c : String) (τ : Ty) :
  S[c]? = some τ →
  (S, Γ) ▷ .const c : τ.toTys ⇒ .const c
  
$$\frac{\Sigma(c) = \tau}{\Gamma \triangleright c : \tau \Rightarrow c} \text{ (const)}$$


| abs_rule (Γ : Ctxs) (y : String) (e : Expr) (τ1 τ : Tys) (e' : Expr) :
  (S, Γ.extend [(y, τ1)] ▷ e : τ ⇒ e' →
  y ∉ dom Γ →
  FV (τ1.arrow τ) ⊆ dom Γ →
  (S, Γ) ▷ .abs y e : τ1.arrow τ ⇒ .abs y e'
  
$$\frac{\begin{array}{l} \Gamma\{y : \tau_1\} \triangleright e : \tau \Rightarrow e' \\ y \notin \text{dom}(\Gamma) \\ FV(\tau_1 \rightarrow \tau) \subseteq \text{dom}(\Gamma) \end{array}}{\Gamma \triangleright \lambda y. e : \tau_1 \rightarrow \tau \Rightarrow \lambda y. e'} \text{ (abs)}$$


...

```

Figure 5.6: Lean translation of the Fischbach–Hannan lifting specification

Figure 5.6 (which continues on the next page) shows the Lean translation of the Fischbach–Hannan lifting specification. The inductive type `LiftJudgement` $S \Gamma e \tau e'$ represents the lifting judgement $\Gamma \triangleright e : \tau \Rightarrow e'$ with $\Sigma = S$. We also declare a Lean syntax notation as for `TypeJudgement`. The Lean translation is remarkably similar to the original specification (displayed side-by-side for comparison). The differences are that the translation has explicit type declarations (though almost all of them could have been inferred), explicit handling of missing mapping keys (the operator `[]?` returns `Option.none` if the key is missing), and that the premise $i \in \{1..n\}$ of the rule (letrec) must be translated to explicit quantification; as stated earlier, we translated the premise $\Gamma^* = \dots$ using inlining, and our notation mentions S explicitly.

```

| lift_abs_rule (Γ : Ctxs)
  (y : String) (e : Expr) (τ : Tys)
  (z : String) (e' : Expr)
  (x : String) (τ1 : Tys) :
    Γ[x]? = some τ1 →
    (S, Γ) ▷ .abs y e : τ ⇒ .abs z e' →
    (S, Γ) ▷ .abs y e : .singarrow x τ1 τ ⇒ .abs x (.abs z e')
    
$$\frac{\Gamma(x) = \tau_1 \quad \Gamma \triangleright \lambda y. e : \tau \Rightarrow \lambda z. e'}{\Gamma \triangleright \lambda y. e : \{x\}_{\tau_1} \rightarrow \tau \Rightarrow \lambda x. \lambda z. e'} \text{ (lift-abs)}$$


| app_rule (Γ : Ctxs) (e1 e2 : Expr) (τ1 τ : Tys) (e1' e2' : Expr) :
  (S, Γ) ▷ e1 : τ1.arrow τ ⇒ e1' →
  (S, Γ) ▷ e2 : τ1 ⇒ e2' →
  (S, Γ) ▷ e1.app e2 : τ ⇒ e1'.app e2'
  
$$\frac{\Gamma \triangleright e_1 : \tau_1 \rightarrow \tau \Rightarrow e'_1 \quad \Gamma \triangleright e_2 : \tau_1 \Rightarrow e'_2}{\Gamma \triangleright e_1 @ e_2 : \tau \Rightarrow e'_1 @ e'_2} \text{ (app)}$$


| lift_app_rule (Γ : Ctxs) (e : Expr) (τ : Tys)
  (x : String) (τ1 : Tys) (e' : Expr) :
  (S, Γ) ▷ e : .singarrow x τ1 τ ⇒ e' →
  Γ[x]? = some τ1 →
  (S, Γ) ▷ e : τ ⇒ e'.app (.var x)
  
$$\frac{\Gamma \triangleright e : \{x\}_{\tau_1} \rightarrow \tau \Rightarrow e' \quad \Gamma(x) = \tau_1}{\Gamma \triangleright e : \tau \Rightarrow e' @ x} \text{ (lift-app)}$$


| letrec_rule (Γ : Ctxs) (n : Nat)
  (fs : Vector String n)
  (es : Vector Expr n)
  (τs : Vector Tys n)
  (e : Expr) (τ : Tys)
  (es' : Vector Expr n)
  (e' : Expr) :
  (∀ i, ( _ : i < n) →
    (S, /-Γ*-/ Γ.extend' fs τs) ▷ es[i] : τs[i] ⇒ es'[i]) →
  (S, /-Γ*-/ Γ.extend' fs τs) ▷ e : τ ⇒ e' →
  (∀ fi ∈ fs, fi ∉ dom Γ) →
  FV τ ⊆ dom Γ →
  (S, Γ) ▷ .letrec (zip' fs es) e : τ ⇒ .letrec (zip' fs es') e'
  
$$\frac{\begin{array}{l} \Gamma^* = \Gamma \{\overline{f_i : \tau_i}\} \quad \Gamma^* \triangleright e_i : \tau_i \Rightarrow e'_i \\ \Gamma^* \triangleright e : \tau \Rightarrow e' \quad f_i \notin \text{dom}(\Gamma) \\ FV(\tau) \subseteq \text{dom}(\Gamma) \quad i \in \{1..n\} \end{array}}{\Gamma \triangleright \text{letrec } \overline{f_i = e_i} \text{ in } e : \tau \Rightarrow \text{letrec } \overline{f_i = e'_i} \text{ in } e'} \text{ (letrec)}$$


```

Figure 5.6: Lean translation of the Fischbach–Hannan lifting specification (cont.)

5.8 Testing

We tested our algorithm implementation on examples from the literature and on a large suite of generated test cases. We checked that the output of our implementation satisfies the specifications and that it agrees with the output of algorithms from the literature.

5.8.1 Test case generation

We implemented two methods for generating test cases: exhaustive enumeration (for small test cases), and random generation (for large test cases).

Our exhaustive enumeration algorithm takes a context Γ (indicating the identifiers currently in scope and their types), a target size (roughly corresponding to the number of non-leaf AST nodes), and a target type. It returns a lazily-computed list of all expressions of the target size and type.

The primitives used by our enumeration algorithm are the Cartesian product and integer partitioning. When the target size is 0, we simply enumerate the pre-defined constants and the identifiers in scope that have the target type. When the target size is non-zero, we enumerate the expressions for each of the AST node kinds `abs`, `app`, and `letrec`. For example, suppose the target size is $s = 5$ and we are currently considering creating an AST node that requires $k = 3$ child nodes. We first enumerate all k -tuples that sum to $s - 1$, in this case $\{(0, 0, 4), (0, 1, 3), (0, 2, 2), \dots, (1, 0, 3), (1, 1, 2), \dots\}$. Then for each tuple (s_1, s_2, s_3) , we recursively enumerate the expressions of sizes s_1, s_2, s_3 ; call these enumerations C_1, C_2 , and C_3 . The Cartesian product $C_1 \times C_2 \times C_3$ enumerates the possible child nodes for the current AST node (since the total size together with that node will be $1 + s_1 + s_2 + s_3 = s$). For `letrec` expressions, we choose the number of bindings by including it in the size of the AST node and using the same integer partitioning scheme.

In some cases we also need to enumerate types. For example, when enumerating the possible `letrec` bindings, we must choose a type for each binding. Likewise, when we enumerate `app` nodes, we must choose a type for the argument. In such cases, we exhaustively enumerate types using a similar scheme; we include the size of the type in the size of the expression using the same integer partitioning scheme to ensure termination.

For practical reasons, we assume the lifting algorithms treat identifier names as opaque. One can artificially construct an algorithm that succeeds on all cases except when the program contains the identifier “supercalifragilisticexpialidocious”; uncovering such cases would require enumeration of all strings, which is impractical. Instead, we choose the names of identifier using a straightforward scheme: we use sequential base names (a, b, c, ...) for the bindings in a `letrec` expression, and we append the current target size as a suffix to ensure there is no shadowing (since it is forbidden by the Fischbach–Hannan specification). This scheme intentionally does not ensure that identifiers are unique across the entire program. When we test algorithms such as Johnsson’s that require identifiers to be unique, we use a separate transformation step to make the identifiers unique (and then we reverse this transformation on the algorithm’s output); For algorithms that do not make this assumption, including non-unique identifiers in the test cases is desirable because it can unveil bugs. However, we make no attempt to enumerate all possible combinations of clashing identifiers.

Other than the opaque identifier assumption, our enumeration algorithm is completely *black-box*: it enumerates all expressions without making any assumptions about the underlying lifting problem. For example, assuming x and y are variables with the same type, the subexpressions $f @ x @ y$ and $f @ y @ x$ are interchangeable for the purpose of lambda lifting—all that matters is whether a variable is referenced or not; but our enumeration algorithm treats those subexpressions as distinct, and will enumerate both. Unfortunately this results in significant redundancy in the enumerated expressions. One can imagine a higher-level enumeration algorithm that reduces the redundancies by enumerating *call graphs* rather than expressions, and hence allows larger test cases to be enumerated exhaustively. However, there is a risk that such higher-level test cases might miss bugs tied to the specific structure of the AST. This is especially true for the algorithms from the literature, where the core lifting logic is intertwined with the traversal of the AST (in contrast with our algorithm, which operates on an identifier reference graph produced by a separate AST preprocessing step). For this reason we have chosen to use exhaustive enumeration of expressions, and rely on random generation to cover larger test cases.

Our random generation algorithm follows a very similar approach to the

exhaustive enumeration algorithm, but uses pseudo-random choices instead of enumerating all options. The random generation algorithm is *biased* to produce more “interesting” test cases—for example, we increase the likelihood of generating functions in `letrec` bindings and of generating calls to named functions, because non-trivial call graphs (in particular containing cycles) are important for good coverage; and we reduce the likelihood of a large number of bindings in `letrec` expressions, to encourage deeper nesting.

We visualised some of the call graphs and declaration graphs using Graphviz¹¹ to gauge their quality and fine-tune the biases. Figure 5.7 shows the declaration graph for one randomly-generated test case of size 1000; an edge $a \longrightarrow b$ denotes that a is the enclosing function of b . Figure 5.8 show the call graph of the same test case. In both figures, λ denotes an anonymous function. The call graph shows that this test case has a couple of groups of mutually-recursive functions (that is, cycles that are not self-loops).

5.8.2 Correctness checks

For each algorithm and each test case, we checked that:

1. the extraneous parameters chosen by the algorithm satisfy our Johnsson-style lifting specification;
2. the lifted program satisfies the Fischbach–Hannan specification;
3. the lifted program satisfies our “complete lifting” specification; and
4. the extraneous parameters are minimal (with some limitations, discussed below).

To check item 1, we use a straightforward decision procedure to check that the extraneous parameter satisfy the scoping premises, and then we apply Theorem 5.6.7 (Sufficient conditions for Johnsson-style lifting).

For item 2, we apply Theorem 5.6.1 (Johnsson-style lifting satisfies the Fischbach–Hannan specification).

We check item 3 using a straightforward decision procedure.

Checking minimality (item 4) directly is difficult; this is the only property for which our checker does not produce a Lean proof.

¹¹<https://graphviz.org/>

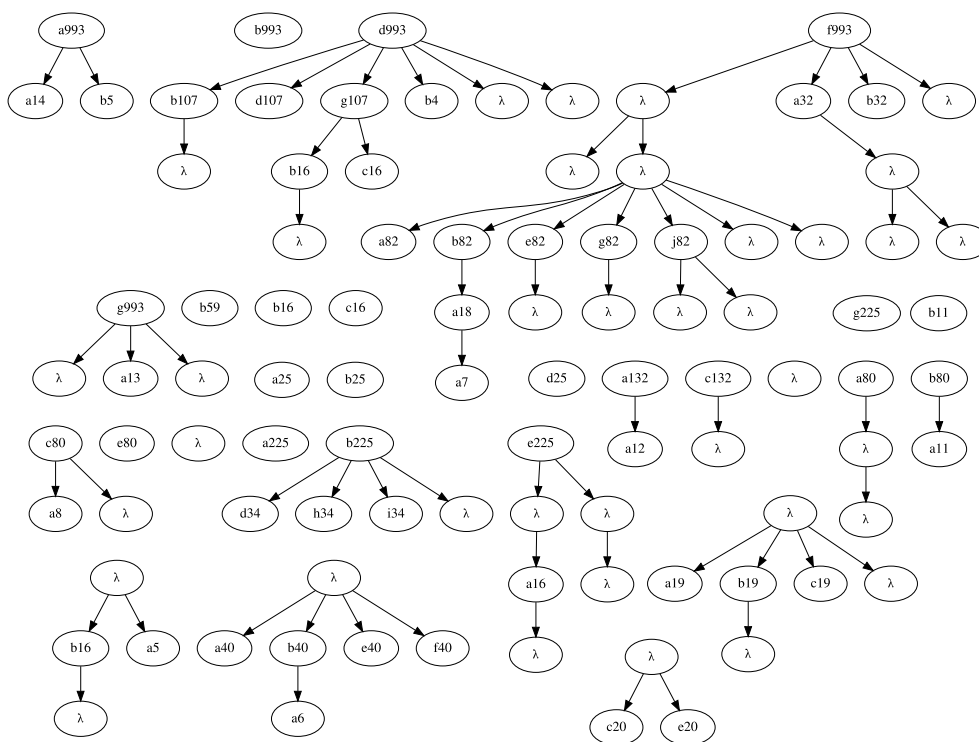


Figure 5.7: Declaration graph of a randomly-generated test case

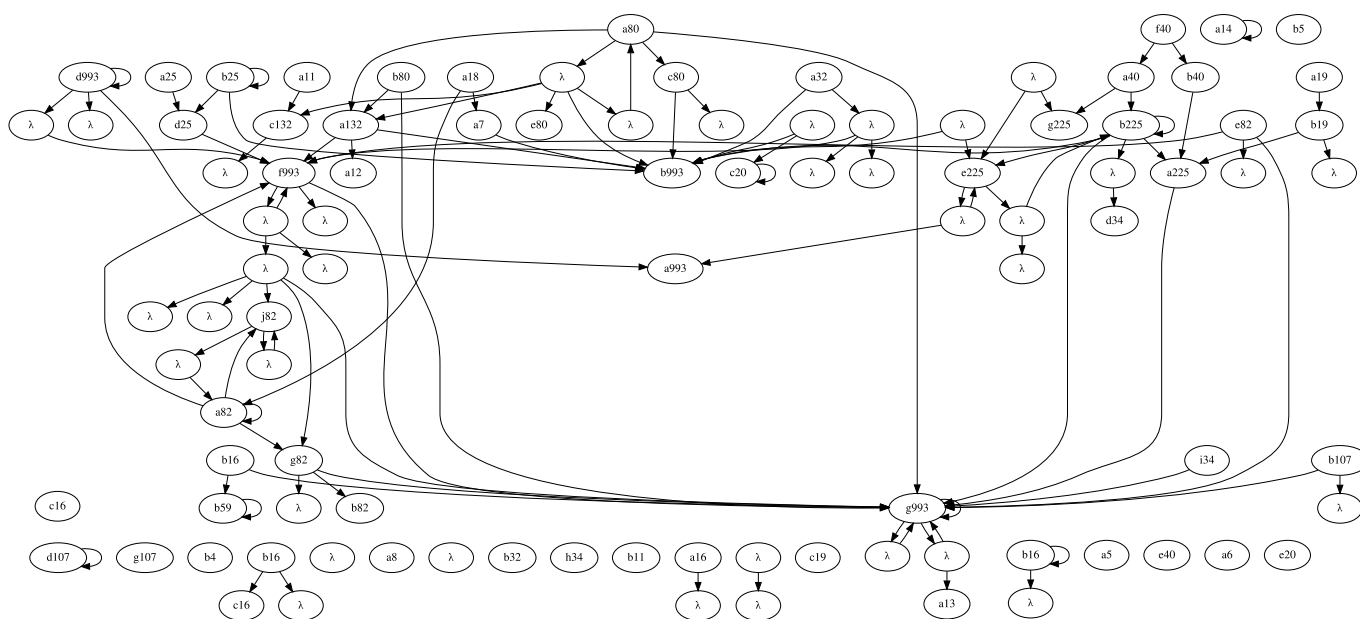


Figure 5.8: Call graph of a randomly-generated test case

The natural approach to check minimality is to enumerate all Johnsson-style liftings with fewer extraneous parameters, check that they do not satisfy the complete lifting specification, and prove that the enumeration is exhaustive. This approach is computationally impractical for large liftings, and proving that the enumeration is exhaustive is difficult.

A proof-based approach would be to translate the proof of [Theorem 5.6.10](#) to Lean (along with the definition of the solution), but that is again difficult.

Instead, our approach is to compare the number of extraneous parameters produced by the different algorithms. This does not show minimality directly, but having agreement between multiple independent algorithms, including one which we proved to be minimal, gives a very high assurance. We also check that the extraneous parameters produced by the different algorithms are the same up to reordering ([Theorem 5.6.10](#) makes this check redundant).

In order to test the algorithms that are not minimal in the presence of dead code, we implemented a basic algorithm that eliminates unused functions. We run this algorithm on the generated test cases to produce modified test cases that do not contain dead code; we then check for minimality on these modified test cases as above.

We also have limited support for the more general liftings allowed by the Fischbach–Hannan specification. We implemented a partial decision procedure that supports a small superset of Johnsson-style liftings (including Hughes-style liftings, hence supporting all the existing algorithms covered by the Fischbach–Hannan specification). Full support for general liftings would be significantly more complex because it requires type inference; alternatively, the lifting algorithm could be tasked with providing type annotations. When our partial decision procedure succeeds, it produces a Lean proof that the lifted program satisfies the Fischbach–Hannan specification; it succeeded on all of our tests (but the algorithms we tested perform Johnsson-style lifting so this is not particularly significant).

5.8.3 Testing Johnsson’s algorithm

Johnsson’s algorithm is presented in [35] as a program written in Lazy ML (an early predecessor of Haskell). It uses lazy evaluation to avoid multiple traversals of the abstract syntax tree—a technique described in [2]. We translated the

program to Lean; since Lean does not support lazy evaluation, we had to restructure the code to use multiple traversals of the AST. To help preserve the original algorithm—compilers for Lazy ML are scarce—we give a direct translation of the Lazy ML program to Haskell in [Appendix B](#).

Running Johnsson’s algorithm on our test cases requires preprocessing:

1. The AST of Johnsson’s algorithm uses multiple parameters instead of currying.
2. Johnsson’s algorithm assumes each anonymous function $\lambda\ldots$ is replaced with the equivalent expression `letrec  $f = \lambda\ldots$  in  $f$` where f is a fresh identifier.
3. Johnsson’s AST requires the function operand of an application expression to be an identifier, not an arbitrary expression. So we convert $e @ e_1 @ \cdots @ e_n$ to `letrec  $f = e$  in  $f(e_1, \dots, e_n)$` where f is a fresh identifier.
4. Johnsson’s algorithm assumes all identifiers are distinct; we rename them to ensure uniqueness.

Johnsson’s algorithm also performs block-floating, which is undesirable for us because our checker expects the extraneous parameters to be specified as annotations on the input expression. Fortunately, it is easy to construct the annotations—we can look up the functions in the output of Johnsson’s algorithm because all identifiers are unique, and their extraneous parameters can be obtained by removing the suffix containing the original parameters. Finally, we translate the result back to the Fischbach–Hannan AST by performing the reverse of the preprocessing steps.

We also implemented a modification to Johnsson’s algorithm to make it produce the minimal extraneous parameters in the presence of dead code, by changing the computation of free variables to exclude identifiers that only occur in unused functions.

5.8.4 Testing the Fischbach–Hannan algorithm

We also tested the Fischbach–Hannan algorithm and the Danvy–Schultz algorithm. For our convenience, we implemented these algorithms in Python rather

than in Lean. To test these implementations we launch them from Lean as subprocesses, and communicate with them over standard input and output; the parent Lean process sends an expression (encoded using JSON), and the child process computes the extraneous parameters and responds with an annotated expression. The Lean process can then perform the usual correctness checks.

The Fischbach–Hannan algorithm is specified both as a set of logical rules in natural deduction form [19, Fig. 5 and Fig. 6] and as an equivalent conventional algorithm [19, Fig. 7]. We translated the latter to Python. Our main difficulty was that the constraint solver subroutine [19, Fig. 8] requires variables to be eliminated in a particular order, otherwise it can fail to terminate; the necessary order can be inferred from the proof of termination [19, Lemma 1].

Some AST transformations are required. As mentioned in Section 5.3 and Section 5.5.2, the Fischbach–Hannan algorithm does not support variable bindings and does not eliminate free variables from anonymous functions; so we have to eliminate both. Eliminating variable bindings is tricky because they may be mutually recursive. The standard way to eliminate non-recursive variable bindings in the lambda calculus is to convert `letrec  $v = e$  in  $b$` to `( $\lambda v. b$ ) @  $e$` , but that is invalid if e references v . Instead, we convert expressions such as

$$\text{letrec } v_1 = e_1; f_2 = \lambda_2 \text{ in } b$$

(where e_1 may refer to f_2 and λ_2 may refer to v_1) to

$$(\lambda v_1. \text{letrec } f_2 = \lambda_2 \text{ in ignore @ } e_1 @ b) @ t_1$$

where `ignore @  $e_1$  @  $b$` returns b (its purpose is to allow us to include e_1 in the transformed program) and t_1 is a trivial expression that has the same type as e_1 .

The transformed program is not semantically equivalent to the original program, because it sets the variables to placeholders instead of the real values; but, importantly, the transformation preserves the behaviour of lambda lifting—the extraneous parameters of the transformed program are the same as those of the original program. Hence we can run the Fischbach–Hannan algorithm on the transformed program to compute the extraneous parameters for the original program.

We also implemented the modification to the Fischbach–Hannan algorithm

Test case kind	Number of test cases
Examples from the literature	15
Randomly-generated ($s \leq 1,000$)	5,000
Randomly-generated ($s = 5,000$)	20
Exhaustive ($s = 5, \Sigma = 2$)	1,929,657
Exhaustive ($s = 6, \Sigma = 0$)	9,781,111
Total	11,715,803

Table 5.1: Number of test cases

we described on page 63 to make the algorithm produce the minimal extraneous parameters in the presence of dead code.

Conveniently, the Danvy–Schultz algorithm requires precisely the same transformations as the Fischbach–Hannan algorithm. Our implementation of the Danvy–Schultz algorithm includes the fix we described in Section 5.2.1.

We also use our subprocess system to test our Haskell translation of Johnson’s algorithm (which is much closer to the original than our Lean translation).

5.8.5 Results

The number of test cases of each kind are shown in Table 5.1. We use 5,000 randomly-generated test cases with sizes up to 1,000, and 20 test cases with a size of 5,000. For exhaustive enumeration we have two configurations. The first configuration limits the size to 5 and allows one base type (`Int`) and two pre-defined constants, `0 : Int` and `succ : Int → Int`. A size limit of 5 is not enough to uncover the dead code issue, so we include a second configuration where the size limit is 6, with one base type and no pre-defined constants. The second configuration uncovers the dead code issue (as do the random test cases); it contains the largest number of test cases and takes up 85% of the overall running time.

As shown in Table 5.2, our algorithm passes all the test cases, as do the algorithms from the literature that we modified to handle dead code—meaning the algorithms produce the same extraneous parameters and pass all our correctness checks.

The symbol † indicates our modified version of an algorithm from the

Algorithm	Pass?	Pass when dead code is eliminated?
Ours	✓	✓
Johnsson (Lean)	✗	✓
Johnsson (Lean)†	✓	✓
Johnsson (Haskell)	✗	✓
Johnsson (Haskell)†	✓	✓
Fischbach–Hannan	✗	✓
Fischbach–Hannan†	✓	✓
Danvy–Schultz	✗	✗
Danvy–Schultz†	✗	✓

Table 5.2: Results

literature. In the case of Johnsson’s algorithm and the Fischbach–Hannan algorithm, our modifications are to produce minimal extraneous parameters in the presence of dead code. In the case of Danvy–Schultz, our modification is to fix the issue described in [Section 5.2.1](#); we did not attempt to make the algorithm minimal in the presence of dead code.

The tests did uncover two issues, but they were unrelated to the lifting algorithms themselves:

1. Python’s recursion limit must be increased using `sys.setrecursionlimit()`¹², otherwise parsing large JSON objects using the `json` module¹³ fails for example.
2. We had a bug in our Lean code for converting an expression to a string that can be parsed from Haskell—we used `reprStr`¹⁴, but it performs line wrapping on long lines and that broke our line-based subprocess communication protocol.

The running times to test each algorithm implementation individually are shown in [Table 5.3](#).^{15,16} We stress that this table *must not* be taken as a fair

¹²<https://docs.python.org/3/library/sys.html#sys.setrecursionlimit>

¹³<https://docs.python.org/3/library/json.html>

¹⁴<https://lean-lang.org/doc/api/Init/Data/Repr.html#reprStr>

¹⁵Without the minimality check, since it requires two implementations to compare.

¹⁶The tests were executed on a desktop machine with an AMD Ryzen 5 PRO 8500GE processor running Ubuntu 24.04.4, using Lean 4.13, GHC 9.14 with `-O2`, and Python 3.12.3.

Algorithm	CPU time (minutes)	Wall clock time (minutes)
Ours	2.1	2.1
Johnsson (Lean)	2.5	2.5
Johnsson (Lean) [†]	2.5	2.5
Johnsson (Haskell)	5.4	16.0
Johnsson (Haskell) [†]	5.4	16.5
Fischbach–Hannan	19.4	101.8
Fischbach–Hannan [†]	19.7	98.3
Danvy–Schultz	20.0	102.3
Danvy–Schultz [†]	20.2	111.1

Table 5.3: Running time to test each implementation

benchmark of the algorithms—it only serves to indicate the time required to test our implementations of the algorithms. The main reasons this does not constitute a fair benchmark are:

- Our AST transformations for the Fischbach–Hannan and Danvy–Schultz algorithms (elimination of variable bindings and anonymous functions) significantly increase the size of the AST. The total size of the input expressions for those implementations is 24.4 GB, in contrast with 2.9 GB for the Haskell implementation of Johnsson’s algorithm. (We designed the transformations to be correct, general, and simple—not to produce a small AST.)
- Subprocess communication adds considerable overhead for the non-Lean implementations. We report the “user CPU time” to exclude time spent in IO, but the time spent in serialization and deserialization of expressions is still included.
- The majority of the test cases are very small, so the running time does not reflect the time complexity.
- The implementations are written in different languages with different

For the implementations that ran in less than an hour, we used the minimum time from 5 runs (in all cases it was close to the median); for the other implementations, we only performed one run.

Algorithm group	CPU time (minutes)	Wall clock time (minutes)
Minimal algorithms	26.7	121.7
All algorithms	74.1	389.8

Table 5.4: Total running time

performance characteristics—for example Python is interpreted, while Lean and Haskell are compiled.

- We only made minimal effort to optimise the implementations; our focus was on writing faithful translations.

What we *can* say is the following:

- If the lifted program is to be output, then our algorithm has an optimal time complexity. So any faster implementation will be faster by at most constant factor. Further, our algorithm does not use any particularly expensive operations, so we expect such a constant factor to be small.
- If the lifted program does not need to be output, then there is room for improvement as discussed in [Section 5.4.7](#).

[Table 5.4](#) shows the running time when the algorithms are executed as a group, to give an idea of how long it takes to run the full test suite. “Minimal algorithms” refers to the four implementations that pass without dead code elimination. “All algorithms” are the eight implementations that pass when we enable dead code elimination (which we did only for that group).

Requiring several hours to run the full test suite may seem excessive, but the exhaustively-enumerated test cases can be easily disabled for quick testing, and in most cases it is not necessary to run all the implementations. (It would also be fairly easy to run the implementations concurrently; we tested a prototype and it reduced the wall clock time for running all algorithms as a group from 389.8 minutes to 137.3 minutes.)

For the dead code issue, the first counterexample found by our exhaustive enumeration algorithm (with a size limit of 6 and with no pre-defined constants) is expression #303,079:

```
letrec a5 = a5 in ( $\lambda$  x4 . letrec a2 =  $\lambda$  x1 . a5 in x4) @ a5
```

In this expression, the anonymous function $\lambda \ x4$ contains an unused function $a2$ that references $a5$.

The test case includes the application $@ \ a5$ because when we run our exhaustive enumeration algorithm we set the target type of the outermost expression to `Int`. Expressions of other types are still explored, but as subexpressions—in general, an expression e of type τ will be explored as part of $(\lambda \ x \ . \ 0) @ \ e$, where x is of type τ and the outermost expression is of type `Int`. (If there are no pre-declared constants, 0 can be replaced with `letrec z = z in z`.)

Overall, we feel confident that the implementations that passed the test suite are correct. The test suite covers all the small edge cases and has good coverage of large test cases, the test suite was able to catch the bugs that we expected, and the results agreed with our initial analysis of the algorithms.

Short of formally proving the correctness of the algorithms, there is still room for more work. The random generation required a surprising amount of tweaking to produce “interesting” test cases, which was slightly concerning. It shows the importance of inspecting generated test cases for desired properties, and that the number of generated test cases can be a misleading metric if only a small portion are useful. The AST-based generators that we implemented would be complemented by generators that operate at the call graph level, since that allows better control over properties such as cycles.

Having multiple independent algorithms produce the same output on many test cases is reassuring, however it not a guarantee of correctness—for example, it is possible for very different algorithms to make the same incorrect assumption, as with the dead code issue. Having a specification that can be checked against provides greater assurance. In our case, the only property which we did not check using a specification was minimality (for the reasons described in Section 5.8.2), but we are content with our “pen and paper” proof of minimality.

5.8.6 Integration with Plausible

Plausible¹⁷ is a property testing framework for Lean 4 based on the Haskell QuickCheck library [10].

We did not use Plausible for our tests because our checks are more natural to write in an imperative style (it allows finer-grained control over the sizes of the

¹⁷<https://github.com/leanprover-community/plausible>

random test cases, inclusion of exhaustively-enumerated and manual examples, custom progress indication, and the use of `IO` to test algorithms implemented in other programming languages and executed using a subprocess).

However, we did implement sample integration with `Plausible`, by providing an instance for `Plausible.Arbitrary Exprt` that wraps our random test case generator. This allows writing tests such as the following (which checks that the randomly-generated test cases are well-typed):

```
#eval Plausible.Testable.check <|
  ∀ e : Exprt, (e.computeType defaultSig {}).isOk
-- Output: Unable to find a counter-example
```

Our sample integration also provides an instance for `Plausible.Shrinkable Exprt`, which is used to reduce the size of counterexamples to make them more readable. Implementing a shrinker for an inductive type is mostly tedious; the main part is to make copies of a value where in each copy one of the children is replaced by a shrunk version. The shrinker is only responsible for performing one step; the testing framework invokes it repeatedly.

In our case, we need to ensure that the shrunk expression is still well-typed; a useful trick is that any expression can be replaced by `letrec x = x in x`. The only pitfall is that we must detect occurrences of this replacement expression and suppress further replacements in them, to ensure that repeated shrink operations reduce the size of the example (otherwise `Plausible` fails to terminate; we reported the issue and submitted a fix that introduces a limit for the number of shrink steps^{18,19}).

Our shrinker has room for improvement. For example, it could try to remove individual `letrec` bindings; that would require identifying and eliminating all references to the removed binding. But we feel that our current implementation is satisfactory: it can reduce an expression all the way to a trivial expression, which is an important property; and reducing to a strictly minimal counterexample is not essential (in fact, the entire shrinking process could be considered optional, because `Plausible` initially tests small examples and gradually increases the size²⁰).

¹⁸<https://github.com/leanprover-community/plausible/issues/90> Tom Levy. 2026. “Non-termination of shrinking process”. GitHub issue.

¹⁹<https://github.com/leanprover-community/plausible/pull/93> Tom Levy. 2026. “fix: introduce Configuration.maxShrinks to ensure termination”. GitHub pull request.

²⁰<https://github.com/leanprover-community/plausible/blob/v4.28.0/Plausible/Testable.lean#L529>

5.9 Related problems

A related problem is the handling of local types inside generic functions in the Go programming language. Converting such local types to global types by inserting extraneous type parameters is an important compilation step.

The problem of determining the minimal extraneous type parameters corresponds to a special case of Johnson-style lambda lifting. Local types correspond to functions and type parameters correspond to variables, with the following restrictions:

- local types cannot be nested inside other local types;
- local types cannot refer to functions (in particular, they cannot refer to their enclosing generic function); and
- a local type cannot refer to local types declared later in the function (due to Go’s scoping rules).

Algorithm 1 could be used to find the minimal extraneous type parameters, but these restrictions prevent mutual recursion, so a simple top-down recursive traversal is sufficient; we describe this approach in [Section 6.1](#).

We also note that there exists a slightly more general graph problem corresponding to lambda lifting. In lambda lifting, the scoping rules impose restrictions on the call graph; for example, a function f cannot be referenced directly by its “grandparent” (the declaring function of the declaring function of f), because f would not be in scope. If we consider a general directed graph without such restrictions, the problem becomes slightly more general. This problem can be stated formally as follows. The definition corresponds closely to the definition of the solution (5.4.6); I corresponds to the set of variables, N corresponds to the “referenced non-local” relation, and D corresponds to the “declaring function” relation.

Definition 5.9.1. Let I be a set, let $G = (V, E)$ be a directed graph, and let N and D be subsets of $I \times V$. Then S is the subset of $I \times V$ satisfying the following conditions, for all $i \in I, f \in V, g \in V$:

1. $(i, f) \in N \Rightarrow (i, f) \in S$
2. $(i, f) \in S \wedge (g, f) \in E \wedge (i, g) \notin D \Rightarrow (i, g) \in S$

3. Nothing else is in S .

Algorithm 1 can be used to compute S , because it does not make any assumptions about the structure of the graph.

Chapter 6

Local Types in Go

As introduced in [Section 1.2.2](#), local types declarations inside generic functions in Go are converted to global types; the process is analogous to lambda lifting, and introduces *implicit type parameters* to the type declarations.

In the case of lambda lifting, the choice of extraneous parameters has no visible effect for the programmer (other than performance). But in the case of local types in Go, the choice of implicit type parameters *does* have a visible effect: it affects type identity.

Consider the following program:

```
package main
func F[T any]() any {
    type S struct{}
    return S{}
}
func main() {
    println(F[int]() == F[string]())
    // Output: false
}
```

Currently, the compiler introduces `T` as an implicit type parameter of `S` (even though `S` does not refer to `T`), effectively converting the declaration to `type S[T any] struct{}`. The value returned by `F[int]()` has type `S[int]`, while the value returned by `F[string]()` has type `S[string]`; these types are distinct, so the values are also distinct. If the compiler did not introduce the type parameter `T`, the output would be `true`.

There has been discussion about this behaviour¹ and the Go developers

¹<https://github.com/golang/go/issues/58573> “spec: are local types in different instan-

decided to keep it, stating “each distinct instantiation of a generic function produces a distinct non-generic function [with] its own distinct local types”.

We argue that this reasoning is flawed, because it uses the behaviour of a particular compiler implementation strategy (namely template-based instantiation) to dictate the semantics of the language; we feel that the correct approach is to consider the semantics first, in isolation from any particular compiler implementation, to the greatest extent practicable.

Type parameters that are referenced by a local type (directly or indirectly) clearly must be added as implicit type parameters, but we argue that there is no reason to add unreferenced type parameters. It is our opinion that adding unreferenced type parameters is surprising and unlikely to match the programmer’s intent. In particular, it means that a snippet of code (that does not use generics and does not mention type parameters) can behave differently depending on whether it is written inside a generic or a non-generic function—the meaning of a trivial type declaration such as `type S struct{}` becomes dependent on the type parameters of the enclosing function, despite having no obvious connection to them.

Another concrete reason to exclude unreferenced type parameters is that they can cause spurious unbounded recursive instantiation cycles (discussed in [Section 4.5](#)). The following example is rejected, because the current implementation adds `T` as implicit type parameter of `S`; if `T` was not added as an implicit type parameter, there would be no unbounded recursive instantiation in this example.

```
package main
func F[T any]() any {
    type S struct{
        return F[S]()
    }
}
```

The problem of determining the minimal required set of implicit type parameters can be solved efficiently using [Algorithm 1](#) (in fact, an even simpler algorithm is available), so the implementation is not a barrier (we describe it explicitly in [Section 6.1](#)). The main obstacle in changing the Go compiler to use the minimal set of implicit type parameters is backward compatibility.

tiations of a generic function identical?”

Go has a strong backward compatibility promise, laid out in the document “Go 1 and the Future of Go Programs”² (2013). The intention is that programs which work with Go 1. x for some x will continue to work with Go 1. y for all $y > x$, with exceptions for breaking changes required to address security issues, correction of bugs, changes to unspecified behaviour, and some other classes of allowed changes. In 2023 this promise was strengthened [14]—it was announced that there will never be a “Go 2” that breaks backward compatibility outright (notably, generics were at one point planned as a Go 2 feature but were eventually added as part of Go 1.18 without breaking backward compatibility); a mechanism called GODEBUG was expanded and formalised to allow programs to opt out of breaking changes (such changes must still abide by the Go 1 compatibility promise).

The Go programming language specification [20] does not state the behaviour of local types inside generic functions explicitly. It says “Two instantiated [generic] types are identical if their defined types and all type arguments are identical” without mentioning implicit type parameters. The sentence “instantiating a [generic] function produces a new non-generic function” does imply indirectly that (distinct) instantiations of a generic function have distinct local types; there is an open issue³ to clarify the specification.

6.1 Algorithm

As explained in [Section 5.9](#), the problem of determining the minimal required set of implicit type parameters corresponds to a special case of Johnsson-style lambda lifting. In this section we give a definition of the minimal implicit type parameters (corresponding to a simplified version of definition 5.4.6) and describe how they can be computed.

Definition. A local type *requires* a type parameter from the enclosing scope if:

- it references the type parameter directly, or
- it references a local type that *requires* the type parameter (transitivity).

²<https://go.dev/doc/go1compat> “Go 1 and the Future of Go Programs”. 2013. The Go Blog.

³<https://github.com/golang/go/issues/65152> “spec: add examples ... regarding the identity ... of types ...”

The minimal implicit type parameters of a local type are the type parameters from the enclosing scope that it *requires*.

The scoping rules of Go prevent a local type from referring to local types declared later in the function. This prevents mutual recursion and greatly simplifies the problem of computing the required type parameters. (If a future release of Go were to allow local types to be mutually recursive, then [Algorithm 1](#) would be appropriate.) A local type can still refer to itself; such references need to be ignored.

Algorithm: Determine the type parameters required by each local type

For each local type t , in declaration order:

Set the implicit type parameters of t to the union of:

- the type parameters from the enclosing scope referenced by t , and
- the implicit type parameters of the local types referenced by t (excluding t itself).

This algorithm does not specify the order of the implicit type parameters. We recommend using the order that they appear in the enclosing declaration. The order is not particularly important; it is only observable when inspecting the local type’s generated name (for example when using reflection, or when printing the Go-syntax representation of a value using `fmt.Printf("%#v", ...)`⁴). In particular, Go’s reflection API intentionally does not include methods to retrieve the type arguments of a generic type.⁵

The worst-case time complexity of this algorithm is $O(K \times R)$, where K is the number of type parameters and R is the number of references to local types and to type parameters, assuming set union is $O(n)$. In “typical” generic functions, K and/or R will be very small. Some possible optimisations for the set union operations are described in [Section 5.4.7](#).

6.2 Backward compatibility

To get an idea of the effect of this change on existing code, we performed an analysis on publicly-available Go repositories. An initial search of the Go

⁴<https://pkg.go.dev/fmt> Package `fmt`. The Go Standard Library.

⁵<https://github.com/golang/go/issues/54393#issuecomment-3206816185> “proposal: reflect: add generic type arg info to `reflect.Type`”. GitHub issue comment.

repository itself (which contains the standard library, compiler, and toolchain) revealed only one local type inside a generic function;⁶ that local type uses all of the type parameters from the enclosing scope, so it would not be affected by a change to use minimal implicit type parameters.

Encouraged by this, we implemented a simple static analyser⁷ and ran it on all publicly available Go source code. We used the Go Module Mirror,⁸ a service operated by Google that provides a cache of Go modules. The documentation of the Go Module Mirror documentation mentions that a separate “cached-only” endpoint is available for bulk static analysis; we used that endpoint.

We found 2,295 packages (out of 2,501,545) containing local types inside generic functions that would potentially be affected by the change.⁹ This is of course too many to analyse by hand; we looked at a few examples, and the impression we got was that local types are commonly used simply to limit the scope of a type declaration, and having implicit type parameters introduced as a side effect is surprising and unintended. There were also many occurrences in test files that can be disregarded.

As future work, we suggest using a more sophisticated analyser that performs escape analysis to determine whether these local types would actually be affected by a switch to minimal implicit type parameters.

⁶https://github.com/golang/go/blob/6db72bb92b2ab68/src/net/cgo_unix.go#L42

⁷<https://github.com/tom93/go-local-type-analyzer>

⁸<https://proxy.golang.org/> “Go Module Mirror, Index, and Checksum Database”.

⁹The total download size of the compressed modules was 6.1 TB. There were 10,376 modules that failed to download (due to HTTP 404 Not Found or HTTP 403 Forbidden errors). We used the latest version of each module as at 1 July 2026.

Chapter 7

Embedded Type Parameters in Go

The interaction between generics and embedded fields is complex. The early Go compiler prototypes allowed type parameters to be used as embedded field types, but we show that those implementations were unsound. The released version of generics in Go 1.18 forbids embedded type parameters due to other reasons; the release notes state that “whether these will ever be permitted is unclear at present”¹.

In this chapter we propose sound promotion rules for embedded type parameters that allow promotion of the methods of corresponding type arguments.

7.1 Introduction

The accepted generics proposal, “Type Parameters in Go” [85], explicitly allows type parameters to be used as embedded field types [85, §Embedded type parameter]:^{2,3}

When a generic type is a struct, and the type parameter is embedded as a field in the struct, any methods of the type parameter’s constraint are promoted to be methods of the struct. (For purposes

¹<https://go.dev/doc/go1.18#generics> “Go 1.18 Release Notes”. The Go Programming Language documentation.

²<https://go-review.googlesource.com/c/proposal/+251444> Ian Lance Taylor. Aug 2020. “design: type parameters: promote embedded methods from constraint”. Go Project Design Documents repository change.

³<https://go-review.googlesource.com/c/proposal/+283113> Ian Lance Taylor. Jan 2021. “design: type parameters: clarify depth of embedded type constraint methods”. Go Project Design Documents repository change.

of selector resolution, these methods are treated as being at depth 0 of the type parameter, even if in the actual type argument the methods were themselves promoted from an embedded type.)

However, the proposal does not explicitly state whether the methods of concrete type arguments are promoted.

The conservative interpretation is that *only* methods specified in the type parameter’s constraint are promoted.

Another interpretation is that the quote refers to contexts where the type argument is unknown, and when the generic type is instantiated with a concrete type argument *all* the methods and fields of the type argument are promoted. This was the behaviour in the early compiler prototypes; it is a natural consequence of monomorphisation, and has more expressive power, but it raises tricky questions regarding promotion which the proposal does not address.

The issue can be summarised using three questions:

1. Are methods of type arguments promoted?
2. When some of the type arguments are not concrete types, are methods of other embedded fields promoted?
3. Can a method of a concrete type argument be promoted when another embedded field has a method with the same name?

If the answer to all three questions is “yes”, then the type system is unsound. This is demonstrated by [Figure 7.1](#).

If the answer to the above questions is “yes”, then the type checker would accept this program. The assignment `var i I = g` is permitted due to (2); specifically, `G[T].S2.S.M` is assumed to be promoted (with return type `int`). The assignment `var _ J = g` is permitted due to (3); specifically, `G[X].T.M` (with return type `string`) is promoted because it is shallower than `G[X].S2.S.M`. This is unsound because Go does not allow a type to have multiple methods with the same name; the value of `g` (which is the same in the two statements) can only have one method `M` and it must have a single consistent return type.

Aside: In Go, a single underscore character is a special placeholder used in variable declarations and assignments. It is useful because the Go compiler reports unused local variable declarations as an error.

```

package main

type S struct{} // struct with method `M() int`.
func (S) M() int { return 0 }

type S2 struct{ // struct with method `M() int` at depth 1.
    S // Normal embedded field.
}

type I interface{ M() int }

type G[T any] struct {
    S2 // Normal embedded field.
    T  // Embedded type parameter.
}

type X struct{} // struct with method `M() string`.
func (X) M() string { return "" }

type J interface { M() string }

func F[T any](g G[T]) int {
    var i I = g // Is g.S2.S.M promoted?
    return i.M()
}

func main() {
    var g G[X]
    var _ J = g // Is g.T.M promoted?
    F(g)
}

```

Figure 7.1: Embedded type parameters and unsound promotion

In one of the development versions of the compiler⁴ (prior to the Go 1.18 release), this program was accepted by the type checker, but the call `i.M()` would fail at runtime with the message “fatal error: unreachable method called. linker bug?”.

The earlier experimental prototype “go2go”⁵ (which translates programs that use the proposed generics syntax to non-generic Go programs) also accepted the program.

Embedded type parameters were disallowed in the compiler prior to the release of generics in Go 1.18 due to other reasons, including complications with reflection^{6,7} (and the language specification was also updated later⁸). However, to the best of our knowledge there has been no prior discussion of the unsoundness.

We begin by describing a few simple (but less expressive) approaches for making the type system sound; we then describe a system of promotion rules that we feel would be the “correct” approach if embedded type parameters are to be added to Go. We derived the rules by carefully considering the possible options and selecting the path that is sound, useful, and consistent with Go’s existing promotion rules. The rules are complex; we give several examples to illustrate the finer points.

7.2 Possible solutions

There are several ways to make the type system sound.

Approach 1: Do not allow type parameters to be embedded in structs.

This is the simplest solution, and is what the Go compiler does in the released version of generics. The cost is reduced expressive power. We give a practical example in [Section 7.3](#) that does not work with this approach.

⁴<https://go.googlesource.com/go>, commit d6753fd491c101e7 ([dev.typeparams] cmd/compile/internal/types2: implement TypeParam.Constraint, 2021-07-23); compilation command: `go run -gcflags=-G=3 main.go`

⁵<https://go.googlesource.com/go/+refs/heads/dev.go2go/README.go2go.md>, branch “dev.go2go”, commit 55626ee50b284ae8; compilation command: `go tool go2go run main.go2`

⁶<https://github.com/golang/go/issues/43621> “go/types, cmd/compile/internal/types2: disallow embedding type parameters”.

⁷<https://go-review.googlesource.com/c/go/+339651> “[dev.typeparams] go/types: embedded type cannot be a (pointer to) a type parameter”.

⁸<https://go-review.googlesource.com/c/go/+603958> “spec: clarify prose for embedded struct fields”.

Approach 2: Only promote methods specified in the type parameter’s constraint; do not promote other methods and fields of type arguments.

Compared to Approach 1, this approach may add some convenience for programmers, but it does not increase expressive power. Specifically, under this approach, the effect of embedding a type parameter T whose constraint is I can be emulated, without embedded type parameters, using two fields: an embedded field of type I and a named field of type T . Both fields are to be initialised to the same value. The purpose of the first field is to promote the methods specified in I , and the purpose of the second field is to allow the value to be accessed with accurate type information. Our example does not work with this approach either.

Approach 3: Only allow promoted fields and methods of a generic type to be accessed in contexts where all the relevant type parameters are instantiated with concrete type arguments. (The relevant type parameters are those that are embedded in the generic type, directly or indirectly.)

Promotion from normal embedded fields is Go’s only analogy to subclassing,⁹ and for the sake of language feature orthogonality we think it should also be available in generic code—including in the problematic case of a generic type with embedded type parameters; so we are against this approach.

Approach 4: Allow fields and methods of type arguments to be promoted, but give them low precedence to ensure the results are consistent. The rules are complex; we describe them in [Section 7.5](#).

7.3 Motivation

Embedded type parameters allow a form of programming that corresponds to the “mixin” pattern [5]. This allows a type to *wrap* another type and override some of its methods. The same effect can be achieved without embedded type parameters, but it is awkward to implement—the programmer must manually set up a circular pointer structure, and manually write method stubs that forward calls to the wrapper (or use very careful manipulation of struct embedding depths).

⁹https://go.dev/doc/faq#Is_Go_an_object-oriented_language “Frequently Asked Questions (FAQ)”. The Go Documentation. Retrieved 7 July 2026.

The key features of this pattern are:

- The wrapper type overrides methods of an arbitrary original type.
- The wrapper type may call methods of the original type.
- The methods of the original type that are not overridden remain accessible.

For example, consider the struct `transform.Reader` from the package `golang.org/x/text/transform`¹⁰. It wraps a given `io.Reader` and transforms the bytes read using a given transformation function.

A limitation of `transform.Reader` is that it does not expose any methods of the wrapped value. So if the programmer needs to call other methods (for example `Close`), they must keep a separate reference to the wrapped value and call the methods using that reference.

Even if `transform.Reader` were to use a (normal) embedded field of type `io.Reader`, only the methods specified in `io.Reader` would be promoted¹¹—regardless of the actual type of the wrapped value. An embedded field of type `io.ReadCloser`¹² would expose `Close`, but no other methods (and it would require *all* wrapped values to implement `Close`).

In contrast, with embedded type parameters, the other methods of the wrapped type become visible (assuming our proposed promotion rules are adopted).

Figure 7.2 shows how embedded type parameters can be used to write an improved version of `transform.Reader`, which overrides the `Read` method (like the original `transform.Reader`) but also exposes the methods of the wrapped type.

The example declares a variable named `conn` which uses the improved `TransformReader` to wrap `*net.TCPConn`¹³. This variable has both the method `Read` (which resolves to `TransformReader.Read`) and all the other methods of `TCPConn` (such as `LocalAddr`). This means that the programmer does not need to maintain a separate reference to the underlying `*net.TCPConn`.

Note that in some cases it does not make sense to expose methods of the wrapped type; for example, with `TransformReader`, if the wrapped type has a

¹⁰<https://pkg.go.dev/golang.org/x/text/transform>
Package `golang.org/x/text/transform`. The Go Project.

¹¹The only method is `Read`, which is overridden—so in fact no methods would be promoted.

¹²<https://pkg.go.dev/io#ReadCloser> Package `io`. The Go Standard Library.

¹³<https://pkg.go.dev/net#TCPConn> Package `net`. The Go Standard Library.

```

// Some existing types that we use without modification.
type TCPConn struct { ... } // From package "net".
func (c *TCPConn) Read(b []byte) (int, error) { ... }
func (c *TCPConn) LocalAddr() net.Addr { ... }
type Transformer interface { // From package "golang.org/x/text/transform".
    Transform(dst, src []byte, atEOF bool) (nDst, nSrc int, err error)
    ...
}

// New type: TransformReader, an improved version of transform.Reader.
type TransformReader[R io.Reader] struct {
    R // Embedded type parameter.
    t Transformer
    ... // The rest is the same as transform.Reader.
}

func (tr *TransformReader[R]) Read(b []byte) (int, error) {
    // Read bytes from tr.R, transform then using tr.t, and write the result
    // to b. The implementation is the same as transform.Reader.Read().
}

func NewTransformReader[R io.Reader](r R, t Transformer) *TransformReader[R] {
    return &TransformReader[R]{
        R: r,
        t: t,
        ... // The rest is the same as transform.NewReader.
    }
}

// Example: Apply TransformReader to TCPConn.
func main() {
    var underlyingConn *TCPConn = ...
    var t Transformer = ...
    conn := NewTransformReader(underlyingConn, t)

    conn.Read(...) // Resolves to TransformReader.Read.
    _ = conn.LocalAddr() // Valid (unlike with transform.Reader).
}

```

Figure 7.2: Motivational example for embedded type parameters

`ReadByte()` method, that method must not be called directly because doing so would bypass the transformer. However, in general it can be useful for methods to be exposed; for example, when the wrapped type is `TCPConn`, it is useful to expose the methods `LocalAddr`, `RemoteAddr`, and `SetDeadline`.

Embedded type parameters would be a powerful language feature. In object-oriented terminology, they might be described as *type parameter subclassing*. In the analogue of the previous example, the type `TransformReader` would be a generic class that inherits from a type parameter. `TransformReader` would describe a *family* of classes where the superclass is a parameter—for example, one instantiation of `TransformReader` might set `TCPConn` as the superclass. Standard object-oriented languages such as Java and C# do not have an equivalent feature, but template-based languages such as C++ do.

7.4 Design

The proposed rules have been derived through the following design principles and decisions for generic types with embedded type parameters:

Principle 1: When a generic type with embedded type parameters is instantiated with type arguments, fields and methods from the type arguments should be promoted (subject to disambiguation rules).

Rationale: This feature adds expressive power to the language. It makes the example in [Section 7.3](#) possible. Without this feature, embedded type parameters do not add expressive power (see the discussion of Approach 2 in [Section 7.2](#)).

Principle 2: If a generic type has both embedded type parameters and normal embedded fields, promotion from the normal embedded fields should take precedence (regardless of the depths).

Rationale: This principle makes it possible to determine promotion from normal embedded fields while there is incomplete type information (that is, in contexts where the type arguments are not concrete). See [Example 4](#) for a case where this principle allows a method to be promoted. This principle makes embedded type parameters useful in generic code, giving language feature orthogonality (see the discussion of Approach 3 in [Section 7.2](#)).

Principle 3: When a generic type that has an embedded type parameter is instantiated with a type argument, the resulting instantiation should be substitutable in place of the type argument to the extent possible.

Background: In non-generic code, when a struct type A embeds another struct type B, the method set of A will be a superset of the method set of B (unless A overrides the methods with incompatible signatures). This does not allow values of type A to be assigned to variables of type B, but it does allow other forms of substitution: values of type A will implement the same interfaces as values of type B, and in many cases it is trivial to take code that uses B and rewrite it to use A (one exception is struct literals, which require the embedded field to be initialised explicitly; also, if A is embedded in some struct, the depths of promoted fields and methods will be one greater than with B).

Practically, this principle implies fields and methods should be promoted from embedded type arguments, with depth one greater than in the type argument, consistently with non-generic embedding. There are some exceptions:

- The accepted type parameters proposal says methods specified in type parameter constraints are treated as being at depth 0 of the type parameter, which is reasonable (see Design Decision 5 below for more details).
- If there are normal embedded fields with conflicting fields or methods, promotion from those fields must take precedence in accordance with Principle 2.

Principle 4: When there is incomplete type information (because not all type arguments are concrete), reasonable steps should still be taken to determine the fields and methods that are provably promoted.

Rationale: There are cases where it is possible to determine promotion even though not all type information is available (see [Example 4](#)).

Determining promotions in these cases adds complexity (in the rules below, in the form of *depth intervals* and *wildcard records*). The alternative is to decide and specify in which cases promotion should be allowed, and that is not easy either (see [Example 5](#) for a simple case where there is no ambiguity and promotion is desirable but it is difficult to specify the required conditions).

Design Decision 5: Treat constraint methods as being at depth 0 of a type parameter, including when type parameters are embedded indirectly.

As quoted earlier, the type parameters proposal says [85]:

When a generic type is a struct, and the type parameter is embedded as a field in the struct, any methods of the type parameter’s constraint are promoted to be methods of the struct. (For purposes of selector resolution, these methods are treated as being at depth 0 of the type parameter, even if in the actual type argument the methods were themselves promoted from an embedded type.)

Treating the constraint methods as being at depth 0 of the type parameter is important because it makes promotion more predictable. Without this rule, the depth of the methods would depend on the type arguments, so it may vary for different type arguments (potentially affecting promotion) and will be unknown in contexts where the type arguments are unknown.

However, it is not clear if the rule in the type parameters proposal is intended to apply only to type parameters embedded directly in the struct, or also to type parameters embedded indirectly (see [Example 7](#) for what we mean by “indirect embedding”).

The promotion rules we propose treat the constraint methods as being at depth 0, both when type parameters are embedded directly and when they are embedded indirectly. This decision aligns with Principle 3, because it allows an embedded type parameter T to be replaced with an instantiation $W[T]$ where W embeds T , while retaining the property that constraint methods are treated as being at depth 0 of T . However, this decision results in an unusual situation (“non-monotonic promotion”) where a method of a deeply nested embedded field is promoted to an outer struct even though it is not promoted to intermediate struct ([Example 9](#)).

The decision to also apply the “depth 0” rule for type parameters embedded indirectly could be reversed with a small update to the proposed promotion rules.

Design Decision 6: Keep legal selectors consistent with the method set and set of fields when there is incomplete type information.

Background: In the existing language specification, a selector expression $x.m$ is legal if m is in the method set of the type of x or if m is a field of the type of x (with two exceptions for shorthands, mentioned in the definition of *legal selector* below).

With the type parameters proposal, there can be contexts with incomplete type information where there is only one known field/method with a particular name (so there is no apparent ambiguity), but the type checker cannot determine whether that field/method will be promoted for all possible type arguments (see [Example 3](#) and accompanying discussion). In such contexts, it is conceivable for the language to allow the selector to be used.

Our proposed promotion rules do not allow selectors to be used in such contexts, to keep the set of legal selectors consistent with the method set and the set of fields. Another way of stating this is that when there is incomplete type information, a selector is legal only if it would be legal and would resolve to the same field/method for all possible type arguments.

Rationale: The promotion rules are complex, and having different rules for legal selectors and for the method set / set of fields could create confusion. More importantly, having a selector resolve to a different field/method (on the same value) in different locations in the source code depending on the available type information would be surprising (see [Example 3](#)).

7.5 Proposed rules

Our proposed promotion rules augment Go's existing depth-based promotion rules with a new notion of *precedence*. The basic idea is to assign *low precedence* to fields and methods that originate from type arguments, to allow them to be promoted while preventing them from overriding fields and methods from normal embedded fields. This way, promotion of fields and methods from normal embedded fields can be determined reliably in contexts where the type arguments are unknown.

These rules follow from the design principles and decisions, and are expressed here in the form of an algorithm that determines the promoted fields and methods. This imperative style may be undesirable for the language specification, but it was easier to reason about the rules and write them correctly in this style.

The algorithm assumes all local type declarations have been converted to top-level type declarations in a preprocessing step (by introducing implicit type parameters, which are discussed in [Chapter 6](#)). This assumption greatly

simplifies the handling of environments.

As written, the algorithm recurses infinitely on valid types that embed themselves, such as `type S struct { *S }`. We discuss a fix for this in [Section 7.6](#).

We add several definitions to help with the description of the algorithm.

7.5.1 Definitions

A *member* is a field or a method.

A *member record* is a tuple (*member*, *depth*, *precedence*, *precedence scope tag*, *ambiguity flag*).

The *depth* of a member record is the number of nested embedded fields traversed. It is either an integer D or an interval $[L, \infty)$ where L is an integer. Depth intervals are used to represent the range of possible depths when the exact depth is unknown.

The *precedence* of a member record is either *normal* or *low*.

The *precedence scope tag* of a member record is used to temporarily change the precedence of the member to low. It identifies the scope in which the low precedence applies. It is present if and only if the precedence is low.

The *ambiguity flag* of a member record is either true or false. It is set to true to mark a member as unusable when there are other members with the same name at the same depth and precedence.

A *wildcard record* is similar to a *member record*, but does not have a member; instead it *matches* member names according to rules introduced in the algorithm. It is used when there is incomplete type information to indicate the possible presence of unknown members. The depth of a wildcard record is always an interval, and the ambiguity flag of a wildcard record is unused.

A record is said to *originate* from a type. The depth of a record is an interval if and only if it originates from a type parameter, and that is the only case where the originating type is of relevance.

The *extended member set* of a type is a collection of *member records* and *wildcard records*. It is an internal data structure computed by the algorithm and used to determine the *member set* and *method set* of the type (defined below). It is called “extended” because, unlike the *member set*, the *extended member set* may contain wildcard records and member records whose ambiguity

flag is set to true. Note that it may contain multiple member records with the same name; such records will always be marked as ambiguous.

The *member set* of a type is the collection of the usable fields and methods of the type. It consists of the members from records in the type's *extended member set* (determined using Algorithm 2) that are member records and whose ambiguity flag is set to false.

The *method set* of a type has the same meaning as in the existing language specification [20, §Method sets]. It is defined here in terms of the member set as follows:

- The *method set* of a non-pointer type T consists of the methods in the member set of T that have receiver type T (that is, fields are excluded and methods with pointer receiver type $*T$ are excluded).
- The *method set* of a pointer type $*T$ consists of the methods in the member set of $*T$ (that is, it includes both the methods with receiver type $*T$ and the methods with receiver type T).

The term *legal selector* also has the same meaning as in the existing language specification and is defined here in terms of the member set: a selector expression $x.m$ is *legal* if the member set of the type of x includes a member named m . The exceptions from the existing language specification [20, §Selectors and §Calls] continue to apply:

- If the type of x is a defined pointer type and $(*x).f$ is a valid selector expression denoting a field (but not a method), $x.f$ is shorthand for $(*x).f$.
- If x is addressable and $\&x$'s method set contains m , $x.m$ is shorthand for $(\&x).m$.

A member record can *shadow* a record, which causes the second record to be ignored. A member record m_1 *shadows* a record m_2 if:

- m_1 has normal precedence and m_2 has low precedence; or
- m_1 and m_2 have the same precedence, and either
 - m_1 has integer depth D_1 , m_2 has integer depth D_2 , and $D_1 < D_2$; or

- m_1 has integer depth D_1 , m_2 has depth interval $[L_2, \infty)$, and $D_1 < L_2$;
or
- m_1 has depth interval $[L_1, \infty)$, m_2 has depth interval $[L_2, \infty)$, m_1 and m_2 *originate* from the same type parameter, and $L_1 < L_2$ (see [Example 7](#)).

Note that if m_1 and m_2 have the same precedence, m_1 has a depth interval, and m_2 does not originate from the same type parameter, then m_1 does not shadow m_2 regardless of the depth of m_2 (see [Example 6](#)).

The *incremented depth* of an integer depth D is the integer $D + 1$, and the *incremented depth* of a depth interval $[L, \infty)$ is the interval $[L + 1, \infty)$.

The *adjusted receiver type* of a method M promoted from an embedded field of type T or $*T$ in a struct type S is defined as follows (consistently with [20, §Struct types]):

- If the embedded field type is $*T$, the adjusted receiver type is S .
- If the method’s receiver type is T , the adjusted receiver type is S .
- If the embedded field type is T and the method’s receiver type is $*T$, the adjusted receiver type is $*S$.

An *environment* is a mapping from type parameters to extended member sets.

A *standalone type expression* is a type expression that is being considered in isolation to determine its member set. The distinction between standalone type expressions and substructures is important, because the extended member set of a type expression may differ depending on whether it is being considered in isolation or in a context where type parameters are bound to type arguments. This is due to the presence of *free type parameters*, whose extended member sets depend on the *environment*.

A *free type parameter* is any occurrence of a type parameter in a standalone type expression. The presence of free type parameters results in situations where the type checker has incomplete type information; the only information available about free type parameters is the constraint declared on the type parameter (if there is one). See [Example 3](#) for a situation where a method is not promoted due to the presence of a free type parameter.

A *concrete type* is a type expression that does not contain free type parameters. For example `int` and `List[int]` are concrete types, but if `T` is a type parameter then `T` and `List[T]` are not concrete types.

7.5.2 Algorithm 2

This is the algorithm for determining the extended member sets of types. It is split into four parts.

- Algorithm 2 is the entry point. It determines the extended member set of a standalone type expression. It handles free type parameters.
- Subalgorithm 2A determines the extended member set of a type expression by inspecting it. It handles the simpler cases such as interfaces and pointer types, and uses Subalgorithm 2B or Subalgorithm 2C for struct types.
- Subalgorithm 2B handles struct types. It contains the core promotion rules.
- Subalgorithm 2C handles instantiations of generic struct types. It is responsible for assigning low precedence to records that originate from type arguments.

Algorithm 2: Determine the extended member set of a standalone type expression T

1. Create an environment E where each type parameter in the type expression T (if any) is mapped to an extended member set as follows:
 - If the type parameter has constraint `any`, bind the type parameter to an extended member set consisting of a single wildcard record that *matches* all member names, with depth interval $[0, \infty)$ and normal precedence, marked as originating from the type parameter (see [Example 3](#)).
 - Otherwise, let C be the constraint of the type parameter. Bind the type parameter to an extended member set consisting of:
 - member records corresponding to the methods of C , with depth interval $[0, \infty)$ and normal precedence, marked as originating from the type parameter; and

- a wildcard record that *matches* all member names except the methods of C , with depth interval $[0, \infty)$ and normal precedence, marked as originating from the type parameter (see [Example 5](#)).
2. Determine the extended member set of the type expression T in the environment E using Subalgorithm 2A.

Subalgorithm 2A: Determine the extended member set of a type expression T in an environment E

- If T is a type parameter, the extended member set is the one bound to T in the environment E .
- If the underlying type of T is an interface type, the extended member set consists of member records corresponding to the methods of the interface type, with depth 0 and normal precedence. (This includes cases where T is the instantiation of a generic type whose underlying type is an interface type.)
- If the underlying type of T is a struct type, then
 - If T is a struct type literal `struct {...}` (“anonymous struct”), determine the extended member set of the struct type T in the environment E using Subalgorithm 2B.
 - If T is a non-generic defined type, determine the extended member set of the struct type T in an empty environment using Subalgorithm 2B. (Note: This step assumes local type declarations have been converted to top-level declarations.)
 - If T is the instantiation of a generic type, determine the extended member set of the instantiation in the environment E using Subalgorithm 2C. (Note: This step assumes local type declarations have been converted to top-level declarations.)
- If T is a non-generic defined type or the instantiation of a generic type (and the underlying type of T is not a struct type), the extended member set consists of member records corresponding to the methods declared with receiver type T or $*T$, with depth 0 and normal precedence.

- If T is a pointer type literal $\star B$, and the base type B is not an interface type and not a pointer type and not a type parameter, recursively determine the extended member set of the type expression B in the environment E using Subalgorithm 2A. The result is the extended member set of T .
- Otherwise, the extended member set is empty.

Subalgorithm 2B: Determine the extended member set of a struct type S in an environment E , where S is one of the following:

- A struct type literal (“anonymous struct”).
 - A non-generic defined type. In this case the environment E must be empty.
 - An uninstantiated generic type. In this case the environment E must correspond to the type parameters of the generic type.
1. Let L be a collection of member records and wildcard records with elements added as follows:
 - (a) If S is not a struct type literal, add member records corresponding to the methods declared with receiver type S or $\star S$, with depth 0 and normal precedence.
 - (b) Add member records corresponding to the fields (including embedded fields) declared in the underlying struct type, with depth 0 and normal precedence.
 - (c) If the underlying struct type has embedded fields, then for each embedded field type F :
 - i. Determine the extended member set of the type expression F in the environment E using Subalgorithm 2A, and let L_F denote that set.
 - ii. Add the records in L_F to the collection L , with *incremented depth* and (if the record is for a method) *adjusted receiver type*.
 2. Remove from L all member records that are *shadowed* by some other member record with the same name.

3. For each distinct member name in L :
 - (a) If there are multiple member records with that name, set the ambiguity flag of those member records to true.
 - (b) Otherwise there is a unique member record with that name. If there is a wildcard record in L that *matches* the name and is not *shadowed* by the unique member record, set the ambiguity flag of the unique member record to true (see [Example 3](#)).
4. The resulting collection L is the extended member set.

Subalgorithm 2C: Determine the extended member set of an instantiation $G[A_1, \dots, A_n]$ in an environment E , where G is a generic struct type

1. Let $P_1..P_n$ be the type parameters of G and let $C_1..C_n$ be their constraints. Let K be a new unique tag, to identify this instantiation.
2. Determine the extended member sets of the type expressions $A_1..A_n$ in the environment E using Subalgorithm 2A. Let $L_1..L_n$ denote those sets.
3. If any of the member records in $L_1..L_n$ are methods specified in the corresponding constraints $C_1..C_n$, set their depth to 0. (Note that their precedence will already be normal, see [Example 8](#).)
4. If any of the other member records or wildcard records in $L_1..L_n$ have normal precedence, set their precedence to low and set their precedence scope tag to K .
5. Create a new environment E_P where $P_1..P_n$ are bound to $L_1..L_n$.
6. Determine the extended member set of G in the environment E_P using Subalgorithm 2B.
7. If any of the records in the extended member set have a precedence scope tag set to K , set their precedence to normal and clear their precedence scope tag.

7.6 Infinite recursion

As written, the algorithm recurses infinitely on valid types that embed themselves, such as `type S struct { *S }`. This issue can be fixed using the observation that when determining the extended member set of a type expression T , embedded (possibly indirectly) copies of T can be assigned empty embedded member sets. This is because any records propagated from the embedded copies of T would be shadowed by the same records of the outer T , and similarly any effects of records propagated from the embedded copies of T (such as marking other records as ambiguous) would be subsumed by the records of the outer T . Go currently prohibits unbounded recursive instantiation, so the number of type expressions that the algorithm may need to consider is finite and this observation is sufficient to prevent infinite recursion.

However, our proposal for first-class generic functions allows the restriction on unbounded recursive instantiations to be lifted,¹⁴ since instantiations can be constructed dynamically. Supporting unbounded recursive instantiations in Algorithm 2 requires additional work; we sketch our suggested approach:

Firstly, we must determine which types embed their type parameters. This corresponds to the problem of dead parameter elimination (mentioned in Section 5.2.2): a type corresponds to a function, type parameters correspond to function parameters, an embedded type parameter corresponds to direct use of a function parameter, and an embedded type corresponds to a function call. A type parameter is embedded (directly or indirectly) if the corresponding function parameter is not a dead parameter. The dead parameter elimination algorithm can be extended to additionally report the minimum depth at which the type parameter is embedded.

Next, to determine the method set of a generic type instantiation $G[A_1, \dots, A_n]$, we add a temporary synthetic definition `type G' struct { G[A1, ..., An] }`¹⁵ and perform Algorithm 2 using a bottom-up propagation, similar to Algorithm 1, starting from all member records with depth 0 and wildcard records with depth $[0, \infty)$ across *all* types, and propagating to the types that embed them (directly or indirectly). The extended member set computed for the embedded field of

¹⁴Provided pointer indirection is used; the size of a struct must still be finite.

¹⁵If $A_1..A_n$ contain free type parameters, say $T_1..T_m$, we instead use the definition `type G'[T1 C1, ..., Tm Cm] struct { G[A1, ..., An] }` where $C_1..C_m$ are the constraints of $T_1..T_m$ in the current scope.

G' gives the desired extended member set. We use a breadth-first propagation (according to member record precedence and depth) rather than depth-first, so that the first propagation(s) to a type gives the highest precedence and shallowest depth—subsequent propagations to that type with greater depth can be ignored because they will be shadowed. The shadowing rules ensure that the propagation terminates; in particular, a wildcard record does not shadow wildcard records originating from other types, but it *does* shadow wildcard records originating from the same type—and the number of originating types is finite.

If embedded type parameters and unbounded recursive instantiation are both to be permitted, this approach would need to be given a more thorough treatment (by explicitly rewriting Algorithm 2 to use bottom-up propagation).

7.7 Embedding in anonymous structs

We propose that embedded type parameters should not be permitted in anonymous structs, because they raise several problems and promotion for anonymous structs is very limited.

The first issue is that generic code can end up creating a struct with multiple fields of the same name (something which is currently forbidden); for example, if a function `func F[T any]() struct { x int; T }` is instantiated with a type named `x` as a type argument, the return type would be `struct { x int; x }`, which has two fields named `x`. For named generic types, the field name of an embedded type parameter is the name of the type parameter (not the name of the type argument); but we cannot use the same approach for anonymous structs, because an anonymous struct is not linked to any declaration or context—it cannot “remember” that the embedded field was originally written as a type parameter. To put this another way, the return type of the instantiation `F[string]` is identical to `struct { x int; string }`, and the original use of `T` can have no bearing on the resulting anonymous struct type.

For similar reasons, promotion is very limited. Principle 2 cannot be followed for anonymous struct types, because we cannot distinguish between normal embedded fields and embedded fields that were originally specified using type parameters.

In generic contexts, it seems that no methods can be promoted at all, because the embedded type parameter might have a conflicting method; it is also possible that the embedded field which results from the embedded type parameter to have the same name as a method—another kind of conflict. (We note that Algorithm 2 assumes all field names are distinct and known.)

In light of these issues, we recommend forbidding embedded type parameters in anonymous structs.

7.8 Examples

As mentioned in the introduction, the rules are complex and have several subtleties. We give examples to illustrate these finer points.¹⁶

Example 1: Simple promotion of a type argument's method

This is a trivial example that shows a method of a concrete type argument being promoted.

```
type SA struct{ } // struct with method M (used as a type argument).
func (SA) M() string { return "" }

type W[T any] struct { // "Wrapper" struct that embeds a type parameter.
    T // Embedded type parameter.
}

func Example1() {
    var w W[SA]
    w.M() // Valid, resolves to w.T.M.
}
```

This example is valid under our proposed promotion rules, because the method SA.M is promoted.

Example 2: Methods of type arguments have low precedence

This example demonstrates that methods of type arguments are not promoted (and do not cause ambiguity) if there are normal promoted methods with the same name.

¹⁶Some declarations are used in multiple examples; we repeat them explicitly in each example, unchanged.

```

type S struct{ } // struct with method M.
func (S) M() string { return "" }

type SA struct{ } // struct with method M (used as a type argument).
func (SA) M() string { return "" }

type I interface { M() string }

type W2[T any] struct {
    // "Wrapper" struct that embeds S and a type parameter.
    S // Normal embedded field.
    T // Embedded type parameter.
}

func Example2() {
    var w W2[SA]
    var _ I = w // Valid.
    w.M() // Resolves to w.S.M.
}

```

There is no ambiguity between `w.S.M` and `w.T.M`, because `w.T.M` has low precedence. The method `w.S.M` is promoted, and `W2[SA]` implements the interface `I`.

Example 3: Ambiguity due to incomplete type information

This example demonstrates that it can be impossible to determine promotion when there is incomplete type information.

The discussion below explains how wildcard records can cause member records to be marked as ambiguous, and why the type checker must not assume that methods will be promoted in some situations. It also explains Design Decision 6 (to keep the set of legal selectors consistent with the method set and set of fields).

```

type SA struct{ }
func (SA) M() string { return "" }

type I interface { M() string }

type Pair[T1, T2 any] struct { // Embeds two type parameters.
    T1 // Embedded type parameter.
    T2 // Embedded type parameter.
}

```

```

func Example3[X any]() {
  var p Pair[SA, X]
  // Naively, we might expect method M to be promoted from SA
  // to Pair[SA, X], but that is not guaranteed for all X.
  var _ I = p // ERROR, because if X has a method M at depth 0
  // there would be a conflict and SA.M would not be promoted.
  p.M() // ERROR: illegal selector according to the proposed
  // rules, even though it is unambiguous.
}

```

To decide whether the line `var _ I = p` is valid, the type checker needs to decide whether `Pair[SA, X]` (the type of `p`) implements the interface `I`. This is done by determining the method set of `Pair[SA, X]`, which in the proposed rules is obtained from the extended member set of `Pair[SA, X]` determined using Algorithm 2.

`X` is a free type parameter with constraint `any`, so Algorithm 2 would assign to it an extended member set consisting of a single wildcard record to indicate that the members of `X` are unknown. This wildcard record will cause the member record `M` originating from `SA` (via the embedded field `T1`) to become ambiguous. Specifically, in Subalgorithm 2B, step 3(b), the wildcard record will *match* the name “`M`”, and since `SA.M` does not *shadow* the wildcard record, the type checker cannot rule out ambiguities and must mark `SA.M` as ambiguous. This is because it is possible that the function `Example3` will be instantiated with `X` bound to a type argument that has a method `M` at depth 0; in that case there would be an ambiguity, and `M` would not be included in the method set of `Pair[SA, X]`—hence `Pair[SA, X]` is not guaranteed to implement `I` for all `X`.

The selector expression `p.M()` is invalid according to the proposed rules, specifically due to Design Decision 6. A hypothetical alternative system of promotion rules could allow that selector expression, since there is only one possible interpretation here; but note that it would be inconsistent with the method set of `p`. Also note that with such a hypothetical system of rules, a selector could resolve to different methods on the same value in different locations in the source code, depending on the available type information. That would be surprising; one consequence is that if such a hypothetical system is used, the compiler must take care to ensure the meaning of a selector in a generic function does not change for certain instantiations, as may happen

with a naive template-based implementation that uses textual substitution to instantiate generic functions (a simple solution would be to expand selectors before substituting the type arguments—here `p.M` expands to `p.T1.M`).

Example 4: Promotion when there is incomplete type information

This example demonstrates that Principle 2 allows a method of a normal embedded field to be promoted despite incomplete type information.

```
type S struct{  
  func (S) M() string { return "" }  
  
type I interface { M() string }  
  
type W2[T any] struct {  
  S // Normal embedded field.  
  T // Embedded type parameter.  
}  
  
func Example4[X any]() {  
  var w W2[X]  
  var _ I = w // Valid.  
  w.M() // Resolves to w.S.M.  
}
```

Principle 2 guarantees that `w.S.M` will be promoted—if `X` had a method `M` it would be assigned low precedence, so it would not conflict with `w.S.M`.

Example 5: Methods specified in contextual constraints can be promoted

This example demonstrates promotion of a method specified in the type constraint of a free type parameter (a “contextual constraint”), illustrating Principle 4.

```
type I interface { M() string }  
  
type W[T any] struct {  
  T // Embedded type parameter.  
}  
  
func Example5[X I]() {  
  var w W[X]  
  var _ I = w // Valid.  
}
```

It is known that X will have a method M . The depth of M in X is not known, but it is known that M will be promoted for any concrete type argument passed for X .

Example 6: Methods specified in contextual constraints have unknown depth

This example demonstrates that methods specified in contextual constraints have unknown depth (unlike methods specified in the type constraints of the generic type, which have depth 0).

```

type SA struct{
func (SA) M() string { return "" }

type SA2 struct { // struct with method M at depth 1.
    SA // Normal embedded field.
}

type I interface { M() string }

type Pair[T1, T2 any] struct {
    T1 // Embedded type parameter.
    T2 // Embedded type parameter.
}

func Example6[X I]() {
    var p Pair[SA2, X]
    var _ I = p // ERROR: method M is ambiguous
}

```

The method $p.T1.SA.M$ (where $T1 = SA2$) has low precedence and depth 2, and the method $p.T2.M$ (where $T2 = X$) has low precedence and unknown depth (this is represented in Algorithm 2 as a depth interval). Because the depth of $p.T2.M$ is unknown in this context, it is not possible to determine which method (if any) will be promoted.

Example 7: Disambiguation when the same type parameter is embedded twice at different depths

This example demonstrates the special case where it is possible to disambiguate between two member records that both have depth intervals.

As shown in [Example 6](#), a member record with unknown depth cannot normally shadow other member records with the same precedence. There is

one exception, which is when the other member record *originates* from the same type parameter. In that case, although the exact depth of the member within the originating type parameter is unknown, its depth must be the same in both cases, so it is possible to compare the depths.

```
type I interface { M() string }

type W[T any] struct {
    T // Embedded type parameter.
}

type W7[T any] struct { // Embeds T twice at different depths.
    T    // Embedded type parameter. (Embeds T directly.)
    W[T] // Embedded field. (Embeds T indirectly.)
}

func Example7[T I]() {
    var w W7[T]
    w.M() // Valid. Resolves to w.T.M.
}
```

In this example `w.T.M` is provably shallower than `w.W.T.M`, even though the depth of `M` in a type argument passed for `T` is unknown; so `w.T.M` is promoted, in accordance with Principle 4.

Example 8: Generic types are type-checked even if they are never instantiated

This example shows a general property of type checking, which is that generic type definitions are type checked even if they are never instantiated.

```
type I interface { M() string }

type GI[T I] struct{ }

type G8[U any] struct {
    GI[U] // ERROR: U is not known to implement constraint I
}
```

This property also gives insight into the note in Subalgorithm 2C step 3. It shows that when a generic type (here `GI`) is instantiated inside another generic type (`G8`), it is a type error if the type arguments of the instantiation (`U`) are not known to implement the corresponding constraints (`I`) based only on the type

constraints of the outer generic type (G8)—that is, regardless of any concrete type arguments supplied in instantiations of the outer generic type.

Algorithm 2 only sets the precedence of a record to low (Subalgorithm 2C, step 4) if it is not specified in the constraint corresponding to the type argument. In a well-typed program, such low-precedence records will not be relied on to satisfy other constraints, so they will not match the condition of Subalgorithm 2C step 3—and by the contrapositive, the records that *do* match the condition will have normal precedence.

Example 9: Non-monotonic promotion

This example demonstrates an unusual situation where a method of a deeply-nested embedded field is promoted to an outer struct even though it is not promoted to intermediate struct.

This situation is a consequence of Design Decision 5 (treat constraint methods as being at depth 0 when type parameters are embedded indirectly).

```
type S struct{ } // struct with method M.
func (S) M() string { return "" }

type S2 struct { // struct with method M at depth 1.
    S
}

type SA struct{ } // struct with method M.
func (SA) M() string { return "" }

type SA2 struct { // struct with method M at depth 1.
    SA
}

type SA3 struct { // struct with method M at depth 2.
    SA2
}

type I interface { M() string }

type Pair[T1, T2 any] struct {
    T1 // Embedded type parameter.
    T2 // Embedded type parameter.
}
```

```

type G9[T I] struct {
    Pair[S2, T]
}

func main() {
    var g G9[SA3]
    g.M()      // Valid. Resolves to g.Pair.T2.SA2.SA.M.
    g.Pair.M() // Valid. Resolves to g.Pair.T1.S.M.
}

```

The call `g.M()` resolves to `g.Pair.T2.SA2.SA.M` (where $T_2 = T = SA_3$), because the constraint I causes the depth of M in SA_3 to be treated as 0.

In contrast, the call `g.Pair.M()` resolves to `g.Pair.T1.S.M` (where $T_1 = S_2$), because `g.Pair` has type `Pair[S2, SA3]` and `Pair[S2, SA3].T1.S.M` is shallower than `Pair[S2, SA3].T2.SA2.SA.M`.

This is surprising, but it is a natural consequence of Design Decision 5. That decision could be reversed fairly easily.

7.9 Testing

We implemented Algorithm 2 in Go and tested it against a Lean specification on random examples.¹⁷

We specify soundness as follows: Given a type expression τ that references a type parameter T whose constraint is C , for any type A that implements C , the method set of the result of substituting A for T in τ must be a superset of the method set of τ .

This captures an important type safety principle: as more information is revealed about a type, its method set may only grow—it is unsound for a promotion algorithm to include a method in the method set if that method would not in fact be included in the method set of a type consistent with the available information.

¹⁷<https://github.com/tom93/go-embedded-types>

Chapter 8

Conclusion

In [Chapter 4](#) we examined the history of generics in Go; we also sketched a possible direction for extending the implementation to support first-class generic functions, and described an $O(n)$ algorithm for detecting unbounded recursive instantiation.

In [Chapter 5](#), we formally specified “Johnsson-style” lambda lifting, including the requirements for a “complete” lifting where all functions are converted to top-level functions. We proved our specification satisfies the existing Fischbach–Hannan lifting specification (allowing us to apply their correctness results). We described an algorithm for computing extraneous parameters for lambda lifting, and proved it produces the minimal extraneous parameters that satisfy our specification. The worst-case time complexity of our algorithm is $O(n^2)$, like the fastest existing algorithm; further, in the output-sensitive complexity model, the time complexity of our algorithm is linear in the size of the output—an improvement over existing algorithms. Our algorithm is simple and produces the minimal extraneous parameters in the presence of unused functions; these are further improvements over the existing algorithms. We used the checker approach ([Chapter 2](#)) to test our algorithm on a large suite of generated test cases.

In [Chapter 6](#) we applied the ideas from lambda lifting to local types in Go. We discussed the reasons for using the minimal type parameters and the issue of backward compatibility.

In [Chapter 7](#) we described a promotion algorithm for embedded type parameters in Go. We again used the checker approach to test our algorithm. This algorithm allows embedded type parameters with promotion to be added

to Go.

8.1 Future work

One area for future work is to formalise the proof of correctness for our lambda lifting algorithm in Lean.

Our analysis of local types in the corpus of Go code was incomplete because the number of uses of local types in generic functions made manual inspection prohibitive. Future work may use more sophisticated static analysis to determine whether changing the compiler to use minimal type parameters would break any existing code.

Our description of the promotion algorithm does not handle types that embed themselves for simplicity; to support them, the algorithm would have to be adjusted as discussed in [Section 7.6](#).

The design of our promotion algorithm is shaped by the fact that Go does not support method overloading and that each type has a fixed method set. In contrast, languages such as C# support overloading and explicit interface implementation.¹ The C# approach is powerful but it is tied to nominal typing, while Go uses structural typing for interfaces. If Go’s type system were extended to support explicit interface implementation, the design of our promotion algorithm would need to be revisited.

¹<https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/explicit-interface-implementation> “Explicit Interface Implementation”. C# Programming Guide. Retrieved 7 July 2026.

References

- [1] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. In *Lectures on Runtime Verification: Introductory and Advanced Topics*, Ezio Bartocci and Yliès Falcone (Eds.). Springer International Publishing, Cham, 1–33. doi:[10.1007/978-3-319-75632-5_1](https://doi.org/10.1007/978-3-319-75632-5_1)
- [2] R. S. Bird. 1984. Using circular programs to eliminate multiple traversals of data. *Acta Informatica* 21, 3 (01 Oct. 1984), 239–250. doi:[10.1007/BF00264249](https://doi.org/10.1007/BF00264249)
- [3] Guy Blelloch, Daniel Ferizovic, and Yihan Sun. 2022. Joinable Parallel Balanced Binary Trees. *ACM Trans. Parallel Comput.* 9, 2, Article 7 (April 2022), 41 pages. doi:[10.1145/3512769](https://doi.org/10.1145/3512769)
- [4] Manuel Blum and Sampath Kanna. 1989. Designing programs that check their work. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing* (Seattle, Washington, USA) (*STOC '89*). ACM, New York, NY, USA, 86–97. doi:[10.1145/73007.73015](https://doi.org/10.1145/73007.73015)
- [5] Gilad Bracha and William Cook. 1990. Mixin-based inheritance. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications* (Ottawa, Canada) (*OOPSLA/ECOOP '90*). ACM, New York, NY, USA, 303–311. doi:[10.1145/97945.97982](https://doi.org/10.1145/97945.97982)
- [6] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. 1998. Making the future safe for the past: adding genericity to the Java programming language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and*

- Applications* (Vancouver, British Columbia, Canada) (*OOPSLA '98*). ACM, New York, NY, USA, 183–200. doi:[10.1145/286936.286957](https://doi.org/10.1145/286936.286957)
- [7] Mark R. Brown and Robert E. Tarjan. 1979. A Fast Merging Algorithm. *J. ACM* 26, 2 (April 1979), 211–226. doi:[10.1145/322123.322127](https://doi.org/10.1145/322123.322127)
 - [8] David Thrane Christiansen. 2025. Functional Programming in Lean. https://docs.lean-lang.org/functional_programming_in_lean/Programming-with-Dependent-Types/Indexed-Families/ Section “Indexed Families”.
 - [9] Alonzo Church. 1941. *The Calculi of Lambda-Conversion*. Princeton University Press, USA.
 - [10] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York, NY, USA, 268–279. doi:[10.1145/351240.351266](https://doi.org/10.1145/351240.351266)
 - [11] Russ Cox. 2009. The Generic Dilemma. <https://research.swtch.com/generic>
 - [12] Russ Cox. 2009. Go Data Structures: Interfaces. <https://research.swtch.com/interfaces>
 - [13] Russ Cox. 2018. Generics — Problem Overview. <https://go.goglesource.com/proposal/+/00fd2f6529173869/design/go2draft-generics-overview.md>
 - [14] Russ Cox. 2023. Backward Compatibility, Go 1.21, and Go 2. The Go Blog. <https://go.dev/blog/compat>
 - [15] Olivier Danvy and Ulrik P. Schultz. 2002. Lambda-Lifting in Quadratic Time. In *Functional and Logic Programming*, Zhenjiang Hu and Mario Rodríguez-Artalejo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 134–151. doi:[10.1007/3-540-45788-7_8](https://doi.org/10.1007/3-540-45788-7_8)
 - [16] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*,

- André Platzter and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 625–635. doi:[10.1007/978-3-030-79876-5_37](https://doi.org/10.1007/978-3-030-79876-5_37)
- [17] Stephen Ellis, Shuofei Zhu, Nobuko Yoshida, and Linhai Song. 2022. Generic Go to Go: Dictionary-Passing, Monomorphisation, and Hybrid. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 168 (Oct. 2022), 29 pages. doi:[10.1145/3563331](https://doi.org/10.1145/3563331)
 - [18] Arthur Evans. 1968. PAL—a language designed for teaching programming linguistics. In *Proceedings of the 1968 23rd ACM National Conference (ACM '68)*. ACM, New York, NY, USA, 395–403. doi:[10.1145/800186.810604](https://doi.org/10.1145/800186.810604)
 - [19] Adam Fischbach and John Hannan. 2003. Specification and correctness of lambda lifting. *Journal of Functional Programming* 13, 3 (2003), 509–543. doi:[10.1017/S0956796802004604](https://doi.org/10.1017/S0956796802004604)
 - [20] The Go Authors. 2026. The Go Programming Language Specification. <https://go.dev/ref/spec> Language version go1.26. Archived at https://go.golangsource.com/go/+/refs/tags/go1.26.0/doc/go_spec.html.
 - [21] Wolfgang Goerigk, Thilo Gaul, and Wolf Zimmermann. 1999. Correct Programs without Proof? On Checker-Based Program Verification. In *Tool Support for System Specification, Development and Verification*, Rudolf Berghammer and Yassine Lakhnech (Eds.). Springer Vienna, Vienna, 108–122. doi:[10.1007/978-3-7091-6355-9_8](https://doi.org/10.1007/978-3-7091-6355-9_8)
 - [22] Brian Goetz. 2020. Background: How We Got the Generics We Have. <https://openjdk.org/projects/valhalla/design-notes/in-defense-of-erasure>
 - [23] Herman Heine Goldstine and John von Neumann. 1947. Planning and coding of problems for an electronic computing instrument. *Report on the Mathematical and Logical aspects of an Electronic Computing Instrument* (1947).
 - [24] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2026. *The Java[®] Language Specification, Java SE 26 Edition*. Oracle America, Inc.

- [25] Sebastian Graf and Simon Peyton Jones. 2019. Selective Lambda Lifting. arXiv:1910.11717 [cs.PL] <https://arxiv.org/abs/1910.11717>
- [26] Robert Griesemer. 2019. Proposal: Permit Signed Integers as Shift Counts for Go 2. Go language change proposal. <https://go.googlesource.com/proposal/+/aed55a0715d6e267/design/19113-signed-shift-counts.md>
- [27] Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. 2020. Featherweight Go. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 149 (Nov. 2020), 29 pages. doi:10.1145/3428217
- [28] Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert E. Tarjan. 2012. Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance. *ACM Trans. Algorithms* 8, 1, Article 3 (Jan. 2012), 33 pages. doi:10.1145/2071379.2071382
- [29] David Harel and Yishai Feldman. 2014. *Algorithmics: The Spirit of Computing* (3rd ed.). Springer, Berlin, Germany.
- [30] Fritz Henglein. 1993. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.* 15, 2 (April 1993), 253–289. doi:10.1145/169701.169692
- [31] Ralf Hinze, Johan Jeuring, and Andres Löb. 2007. Comparing Approaches to Generic Programming in Haskell. In *Datatype-Generic Programming*, Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 72–149.
- [32] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. doi:10.1145/363235.363259
- [33] Raymond Hu, Julien Lange, Bernardo Toninho, Philip Wadler, Robert Griesemer, and Keith Randall. 2026. Welterweight Go: Boxing, Structural Subtyping, and Generics. *Proc. ACM Program. Lang.* 10, POPL, Article 79 (Jan. 2026), 28 pages. doi:10.1145/3776721

- [34] R. J. M. Hughes. 1982. Super Combinators—A New Implementation Method for Applicative Languages. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming* (Pittsburgh, Pennsylvania, USA) (*LFP '82*). ACM, New York, NY, USA, 1–10. doi:[10.1145/800068.802129](https://doi.org/10.1145/800068.802129)
- [35] Thomas Johnsson. 1985. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, Jean-Pierre Jouannaud (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–203. doi:[10.1007/3-540-15975-4_37](https://doi.org/10.1007/3-540-15975-4_37)
- [36] Andrew Kennedy and Don Syme. 2001. Design and implementation of generics for the .NET Common language runtime. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) (*PLDI '01*). ACM, New York, NY, USA, 1–12. doi:[10.1145/378795.378797](https://doi.org/10.1145/378795.378797)
- [37] Assaf J. Kfoury and Jerzy Tiuryn. 1992. Type reconstruction in finite rank fragments of the second-order λ -calculus. *Information and Computation* 98, 2 (1992), 228–257. doi:[10.1016/0890-5401\(92\)90020-G](https://doi.org/10.1016/0890-5401(92)90020-G)
- [38] Nishant Kheterpal, J. Tanner Slagel, Elanor Tang, Serra Dane, and Jean-Baptiste Jeannin. 2026. Automatic Certification of the Active Corner Method for Collision Avoidance. In *NASA Formal Methods* (Los Angeles, California, USA), Jyotirmoy Deshmukh, Klaus Havelund, and Alessandro Pinto (Eds.). Springer Nature Switzerland, Cham, 527–548. doi:[10.1007/978-3-032-28079-4_25](https://doi.org/10.1007/978-3-032-28079-4_25)
- [39] Pok-Son Kim and Arne Kutzner. 2004. Stable Minimum Storage Merging by Symmetric Comparisons. In *Algorithms – ESA 2004*, Susanne Albers and Tomasz Radzik (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 714–723.
- [40] Donald E. Knuth. 1976. Big Omicron and big Omega and big Theta. *SIGACT News* 8, 2 (April 1976), 18–24. doi:[10.1145/1008328.1008329](https://doi.org/10.1145/1008328.1008329)
- [41] Dieter Kratsch, Ross M. McConnell, Kurt Mehlhorn, and Jeremy P. Spinrad. 2006. Certifying Algorithms for Recognizing Interval Graphs

- and Permutation Graphs. *SIAM J. Comput.* 36, 2 (2006), 326–353. doi:[10.1137/S0097539703437855](https://doi.org/10.1137/S0097539703437855)
- [42] P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (Jan. 1964), 308–320. doi:[10.1093/comjnl/6.4.308](https://doi.org/10.1093/comjnl/6.4.308)
- [43] The Lean Developers. 2025. The Lean Language Reference. [https://lean-lang.org/doc/reference/4.23.0/Notations-and-Macros/Macros/](https://lean-lang.org/doc/reference/4.23.0/Notations-and-Macros/Macros/Section%20%22Macros%22) Section “Macros”. Lean version 4.23.0.
- [44] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. 2005. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming* 55, 1 (2005), 185–208. doi:[10.1016/j.scico.2004.05.015](https://doi.org/10.1016/j.scico.2004.05.015) Formal Methods for Components and Objects: Pragmatic aspects and applications.
- [45] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. doi:[10.1145/1538788.1538814](https://doi.org/10.1145/1538788.1538814)
- [46] Tom Levy. 2019. Go 2 Generics Feedback. GitHub Gist. <https://gist.github.com/tom93/fd8c500b8a1d2d94f227e1e108d5315b>
- [47] Tom Levy and Steve Reeves. 2026. Lambda Lifting in Lean – source code artifact. Zenodo. doi:[10.5281/zenodo.18349369](https://doi.org/10.5281/zenodo.18349369)
- [48] Tom Levy and Steve Reeves. 2026. Lambda Lifting in Lean – source code repository. <https://gitlab.com/tom93/lambda-lifting-lean/-/tree/FormaliSE-2026> (mirrors: <https://github.com/tom93/lambda-lifting-lean/tree/FormaliSE-2026> and <https://gitlab.com/stever00/lambda-lifting-lean>).
- [49] Tom Levy and Steve Reeves. 2026. Simple Lambda Lifting: Formalisation in Lean and a new efficient algorithm. In *IEEE/ACM 14th International Conference on Formal Methods in Software Engineering (FormaliSE ’26)* (Rio de Janeiro, Brazil, April 12–13, 2026). ACM, New York, NY, USA, 67–78. doi:[10.1145/3793656.3793692](https://doi.org/10.1145/3793656.3793692)
- [50] David MacQueen, Robert Harper, and John Reppy. 2020. The history of Standard ML. *Proc. ACM Program. Lang.* 4, HOPL, Article 86 (June 2020), 100 pages. doi:[10.1145/3386336](https://doi.org/10.1145/3386336)

- [51] John McCarthy. 1978. History of LISP. *SIGPLAN Not.* 13, 8 (Aug. 1978), 217–223. doi:[10.1145/960118.808387](https://doi.org/10.1145/960118.808387)
- [52] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. 2011. Certifying algorithms. *Computer Science Review* 5, 2 (2011), 119–161. doi:[10.1016/j.cosrev.2010.09.009](https://doi.org/10.1016/j.cosrev.2010.09.009)
- [53] Kurt Mehlhorn and Stefan Näher. 1998. From algorithms to working programs: On the use of program checking in LEDA. In *Mathematical Foundations of Computer Science 1998*, Luboš Brim, Jozef Gruska, and Jiří Zlatuška (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 84–93. doi:[10.1007/BFb0055759](https://doi.org/10.1007/BFb0055759)
- [54] Bertrand Meyer. 1992. Applying “Design by Contract”. *Computer* 25, 10 (1992), 40–51. doi:[10.1109/2.161279](https://doi.org/10.1109/2.161279)
- [55] Marco T. Morazán and Barbara Mucha. 2006. Improved Graph-Based Lambda Lifting. In *Software Engineering Research and Practice* (Las Vegas, Nevada, USA). CSREA Press, 896–902.
- [56] Marco T. Morazán and Ulrik P. Schultz. 2008. Optimal Lambda Lifting in Quadratic Time. In *Implementation and Application of Functional Languages*, Olaf Chitil, Zoltán Horváth, and Viktória Zsók (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 37–56. doi:[10.1007/978-3-540-85373-2_3](https://doi.org/10.1007/978-3-540-85373-2_3)
- [57] F. L. Morris and C. B. Jones. 1984. An Early Program Proof by Alan Turing. *Annals of the History of Computing* 6, 2 (1984), 139–143. doi:[10.1109/MAHC.1984.10017](https://doi.org/10.1109/MAHC.1984.10017)
- [58] Joel Moses. 1970. The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem. *SIGSAM Bull.* 15 (July 1970), 13–27. doi:[10.1145/1093410.1093411](https://doi.org/10.1145/1093410.1093411)
- [59] George C. Necula and Peter Lee. 1998. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) (*PLDI '98*). ACM, New York, NY, USA, 333–344. doi:[10.1145/277650.277752](https://doi.org/10.1145/277650.277752)

- [60] Chris Okasaki. 1998. *Purely Functional Data Structures*. Cambridge University Press. doi:[10.1017/CB09780511530104](https://doi.org/10.1017/CB09780511530104)
- [61] Sam Owre, John M. Rushby, and Natarajan Shankar. 1992. PVS: A prototype verification system. In *Automated Deduction – CADE 11*, Deepak Kapur (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 748–752. doi:[10.1007/3-540-55602-8_217](https://doi.org/10.1007/3-540-55602-8_217)
- [62] Sam Owre and Natarajan Shankar. 2008. A Brief Overview of PVS. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 22–27. doi:[10.1007/978-3-540-71067-7_5](https://doi.org/10.1007/978-3-540-71067-7_5)
- [63] Zoe Paraskevopoulou. 2020. *Verified Optimizations for Functional Languages*. PhD Dissertation. Princeton University, NJ. <https://www.cs.princeton.edu/techreports/2020/006.pdf>
- [64] Simon L. Peyton Jones. 1987. *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc., USA.
- [65] Benjamin C. Pierce. 2002. *Types and Programming Languages*. The MIT Press.
- [66] Rob Pike. 2012. Go at Google. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity* (Tucson, Arizona, USA) (*SPLASH '12*). ACM, New York, NY, USA, 5–6. doi:[10.1145/2384716.2384720](https://doi.org/10.1145/2384716.2384720) (<https://go.dev/talks/2012/splash.article>).
- [67] Rob Pike. 2014. Generating code. The Go Blog. <https://go.dev/blog/generate>
- [68] Brian Randell and Lawford J. Russell. 1964. *Algol-60 Implementation*. Academic Press, Inc., USA.
- [69] Markus Schmaus. 2023. Learning Lean 4 as a programming language part 5 – A different style of proofs. <https://unreasonableeffectiveness.com/learning-lean-4-as-a-programming-language-5-a-different-style-of-proofs/>

- [70] Guy L. Steele, Jr. and Gerald Jay Sussman. 1978. *The Art of the Interpreter or, The Modularity Complex*. Technical Report. Massachusetts Institute of Technology, USA.
- [71] Christopher Strachey. 1967. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation* 13, 1 (1967). doi:10.1023/A:1010000313106
- [72] Bjarne Stroustrup. 2009. The C++0x “Remove Concepts” Decision. *Dr. Dobb’s* (July 2009). Archived at <https://web.archive.org/web/20121009010252/https://www.drdobbs.com/cpp/the-c0x-remove-concepts-decision/218600111>.
- [73] Gregory F. Sullivan and Gerald M. Masson. 1990. Using certification trails to achieve software fault tolerance. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*. 423–431. doi:10.1109/FTCS.1990.89397
- [74] Martin Sulzmann and Stefan Wehr. 2023. A type-directed, dictionary-passing translation of method overloading and structural subtyping in Featherweight Generic Go. *Journal of Functional Programming* 33, e8 (2023), 63 pages. doi:10.1017/S0956796823000047
- [75] Gerald Jay Sussman and Guy L. Steele, Jr. 1998. SCHEME: An Interpreter for Extended Lambda Calculus. *Higher-Order and Symbolic Computation* 11, 4 (01 Dec. 1998), 405–439. doi:10.1023/A:1010035624696
- [76] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160. doi:10.1137/0201010
- [77] Ian Lance Taylor. 2010. Type Functions. Go language change proposal. <https://go.googlesource.com/proposal/+/b571c3273d2c6988/design/15292/2010-06-type-functions.md>
- [78] Ian Lance Taylor. 2011. Generalized Types. Go language change proposal. <https://go.googlesource.com/proposal/+/2c6bc83f234085c8/design/15292/2011-03-gen.md>

- [79] Ian Lance Taylor. 2011. Proposal: Go should have generics. Go language change proposal. <https://go.googlesource.com/proposal/+/2c6bc83f234085c8/design/15292-generics.md> Updated 2016.
- [80] Ian Lance Taylor. 2013. Generalized Types In Go. Go language change proposal. <https://go.googlesource.com/proposal/+/2c6bc83f234085c8/design/15292/2013-10-gen.md>
- [81] Ian Lance Taylor. 2013. Type Parameters in Go. Go language change proposal. <https://go.googlesource.com/proposal/+/2c6bc83f234085c8/design/15292/2013-12-type-params.md>
- [82] Ian Lance Taylor. 2019. Why Generics? <https://go.dev/blog/why-generics> GopherCon 2019.
- [83] Ian Lance Taylor and Robert Griesemer. 2018. Contracts — Draft Design. Go language change proposal. <https://go.googlesource.com/proposal/+/4a530dae40977758/design/go2draft-contracts.md>
- [84] Ian Lance Taylor and Robert Griesemer. 2019. Contracts — Draft Design. Go language change proposal. <https://go.googlesource.com/proposal/+/78bd52518d53994d/design/go2draft-contracts.md>
- [85] Ian Lance Taylor and Robert Griesemer. 2021. Type Parameters Proposal. Go language change proposal. <https://go.googlesource.com/proposal/+/8b5daee3198c7b/design/43651-type-parameters.md>
- [86] The Lean Developers. 2025. The Lean Language Reference. <https://lean-lang.org/doc/reference/4.23.0/The-Type-System/Propositions/> Section “Propositions”. Lean version 4.23.0.
- [87] Alan Turing. 1949. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*. Reproduced with typographical corrections in [57].
- [88] D. A. Turner. 2013. Some History of Functional Programming Languages. In *Trends in Functional Programming*, Hans-Wolfgang Loidl and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20. doi:10.1007/978-3-642-40447-4_1

- [89] Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) (*POPL '89*). ACM, New York, NY, USA, 60–76. doi:[10.1145/75277.75283](https://doi.org/10.1145/75277.75283)
- [90] Hal Wasserman and Manuel Blum. 1997. Software reliability via run-time result-checking. *J. ACM* 44, 6 (Nov. 1997), 826–849. doi:[10.1145/268999.269003](https://doi.org/10.1145/268999.269003)
- [91] Joseph Weizenbaum. 1968. The FUNARG problem explained. (1968). Unpublished memorandum.

Appendix A

Worst-case lambda lifting output size

We show that the worst-case size of the lifted program is $\Theta(n^2)$ tokens¹, where n is the number of tokens in the input program. We also show the same when considering the size of the program in bytes.

The lifted program is produced by inserting extraneous parameters in each function definition and extraneous arguments in each call site and reference (then performing “block floating”, which does not affect the size of the program). The total number of function definitions and function references in a program is $O(n)$; for each one, the number of extraneous parameters/arguments is $O(n)$. So the number of tokens in the lifted program is $O(n^2)$.

The following example shows that there exist cases where the number of tokens in the lifted program is $\Theta(n^2)$:

```
(define (f x1 x2 ... xk)
  (define (g) (+ x1 x2 ... xk))
  (define (h1) (g))
  (define (h2) (g))
  ...
  (define (hk) (g))
  (+ (h1) (h2) ... (hk)))
```

The number of tokens in this example is $n = \Theta(k)$. The function g contains k free variables, which need to be added as extraneous parameters. The functions $h1..hk$ also require these extraneous parameters because they call g . So the

¹We use Knuth’s big Theta notation [40] to indicate tight bounds; $\Theta(f)$ means “of exactly order f ”, in contrast with $O(f)$ which means “of order at most f ”.

total number of extraneous parameters in the lifted program is $\sim k^2$. There are $2k$ calls, so the number of extraneous arguments is $2k^2$. Hence the number of tokens in the lifted program is $\Theta(k^2) = \Theta(n^2)$.

We now show that the worst-case size of the lifted program in bytes is $\Theta(s^2)$, where s is the size of the input program in bytes and identifiers are allowed to have arbitrary length.

The total number of function definitions and function references in a program is $O(s)$; for each one, the sum of the lengths of the extraneous parameter/argument names is $O(s)$, since they are distinct and appear in the input program. So the size of the lifted program in bytes is $O(s^2)$.

There exist cases where the size of the lifted program in bytes is $\Theta(s^2)$, but the above example is not such a case. In that example $k = \Theta(s/\log s)$, because for large k the length of the majority of the identifiers is $\sim \log_{10} k$ (which can be approximated by $\log_{10} s$). The sum of the lengths of the parameter names $x1..xk$ is $\propto s$, so the size of the lifted program in bytes is $\Theta(s^2/\log s)$.

We can easily construct a different example where the size of the lifted program in bytes is $\Theta(s^2)$:

```
(define (f x1 x2 ... xk)
  (define (g) (+ x1 x2 ... xk))
  (+ (g) (g) (g) ... (g)))
```

For any desired program size s , we can choose k and the number of calls to g so that the sum of the lengths of the parameter names is $\propto s$, the number of calls to g is $\propto s$, and the size of the example program in bytes is $\sim s$. The total length of the extraneous arguments is $\propto s^2$, so the size of the lifted program in bytes is $\Theta(s^2)$.

It is also interesting to consider just the extraneous parameters, and exclude the extraneous arguments from the size. If we require all the functions in the input program to have distinct names, then the worst-case size of the extraneous parameters in bytes is $\Theta(s^2/\log s)$. This is because the average length of k distinct identifiers is at least $\Omega(\log k)$, implying the number of functions can be at most $O(s/\log s)$. On the other hand, if we allow functions in different scopes to have the same name, then the worst-case size of the extraneous parameters in bytes is $\Theta(s^2)$.

A size of $\Theta(s^2)$ can be achieved using a tree of nested functions where

the leaf functions have single-character names, their parent functions have two-character names, and so on for any desired depth. If m is the number of leaf functions and $|\Sigma|$ is the number of characters in the alphabet ($|\Sigma| \geq 2$), then there will be $m/|\Sigma|$ functions with two-character names, $(m/|\Sigma|)/|\Sigma|^2$ with three-character names, and so on. The following example demonstrates the structure for $m = 26^{3+2+1}$:

```
(define (f x1 x2 ... xk)
  (define (g) (+ x1 x2 ... xk))
  (define (AAA)
    (define (AA)
      (define (A) (g))
      (define (B) (g))
      ...
      (define (Z) (g))
      (+ (A) (B) ... (Z)))
    (define (AB)
      (define (A) (g))
      (define (B) (g))
      ...
      (define (Z) (g))
      (+ (A) (B) ... (Z)))
    ...
    (define (ZZ)
      ...
      (+ (AA) (AB) ... (ZZ)))
    (define (AAB)
      ...)
    ...
    (define (ZZZ)
      ...)
    (+ (AAA) (AAB) ... (ZZZ)))
```

The number of non-leaf functions is $\leq m$, because the series $\sum_{i=1}^{\infty} m/|\Sigma|^i$ (which is an over-approximation) converges to $m/(|\Sigma| - 1)$ when $|\Sigma| \geq 2$. The sum of the lengths of the function names is $\leq 4m$, because the over-approximation $\sum_{i=0}^{\infty} (i+1)m/|\Sigma|^i$ converges to $m|\Sigma|^2/(|\Sigma| - 1)^2$ when $|\Sigma| \geq 2$. So this scheme allows the number of functions to be $\propto s$, and hence the total size of the extraneous parameters in bytes is $\propto s^2$.

Appendix B

Johnsson's algorithm

To help preserve Johnsson's algorithm [35], we give a direct translation of the algorithm from Lazy ML to Haskell.

The translation does not use idiomatic Haskell style because we tried to match the original Lazy ML syntax. Note that the algorithm does not produce the minimal lifting in the presence of dead code (see [Section 5.8.3](#)).

Johnsson's algorithm, translated to Haskell

```
import Data.List

-- Abstract syntax
type Id = String
data Expr = LETREC(List(Def), Expr) | APPL(Id, List(Expr))
    deriving (Show, Read)
data Def = VAR(Id, Expr) | FUN(Id, List(Id), Expr)
    deriving (Show, Read)

-- Set operations
_Mkset :: List(Id) -> List(Id)
_U      :: List(Id) -> List(Id) -> List(Id)    -- union
_Is     :: List(Id) -> List(Id) -> List(Id)    -- intersection
_Mem    :: Id -> List(Id) -> Bool              -- membership
```

```
{-
```

"The function `Lambdalift` contains three auxiliary functions `Le`, `La` and `Ld` which are the real workhorses of the algorithm. `Le` takes an expression, `La` an expression list and `Ld` a definition list as argument. They also take the previously mentioned `vars`, `funcs` and `sol` as arguments.

`vars` is the set of variables in the current scope, i.e. they can occur as free variables in the `letrec` expression. At the start of the lambda lifting process this set is empty.

`funcs` is similarly the set of functions in the current scope; for these functions we already know which variables that has to be abstracted out of the functions. Initially this set is also empty.

`sol` is the solutions of the set equations for the function names in the set `funcs`.

Each of the functions `Le`, `La` and `Ld` returns a tuple:

- the new expression (expression list, definition list) where the lifted functions have been removed,
- the lifted function definitions,
- the set of identifiers occurring in the expression (expression list, definition list).

Furthermore, the function `Ld` has three additional components in the returned tuple:

- the variables defined in the definition list,
- the functions defined in the definition list,
- a list of pairs of function names and the identifiers occurring in each function definition.

Note the unconventional use of recursion, which is particularly conspicuous in the case `Le(LETREC(d,e))`. Components 4-6 of what `Ld` returns is used to compute the set equations for the functions defined in this definition list. These equations are solved, the extended set of solutions `nsol` passed further down in the tree, where `Le`, `La` and `Ld` uses it to compute the first and second component of the tuple. Circular programs such as this one requires lazy evaluation to work. A discussion of such programming techniques can be found in [Bird84]."

```
-}
```

```
-- Lambda lifting program
```

```
_Lambdalift expr =
```

```
  let _Xset sol s z = itlist(\f p->_U (assocdef f sol []) p) s z in
  let solve      []      = []
      solve ((f,s,e):[]) = [(f,s)]
```

```

solve ((f,s,e):l) =
  let so = solve(map(\t->let (f1,s1,e1) = t
                           in if _Mem f e1
                              then (f1, _U s s1, _U e e1)
                              else t) l)
  in (f, itlist (\x p->_U (assocdef x so []) p) e s):so in
let _Le (LETREC(d,e)) vars funs sol =
  let (ed, dd, id, dvars, dfuns, ss) = _Ld d nvars nfunns nsol
      (ee, de, ie) = _Le e nvars nfunns nsol
      nvars = _U vars dvars
      nfunns = _U funns dfuns
      nsol = solve(map(\(f, _S)->
                        (f,
                         _Xset sol (_Is _S funns) (_Is _S nvars),
                         _Is _S dfuns))
                      ss) ++ sol
  in ((if null ed then ee else LETREC(ed,ee)),
      de ++ dd, _U id ie)
_Le (APPL(f,_E)) vars funs sol =
  let (eE, dE, iE) = _La _E vars funns sol
  in (APPL(f,map(\i->APPL(i,[]))(assocdef f sol [])) ++ eE),
      dE, _U [f] iE)

_La [] vars funns sol = ([], [], [])
_La (e:l) vars funns sol =
  let (ee, de, ie) = _Le e vars funns sol
      (el, dl, il) = _La l vars funns sol
  in (ee:el, de++dl, _U ie il)

_Ld [] vars funns sol = ([], [], [], [], [], [])
_Ld (VAR(x,e):l) vars funns sol =
  let (ee, de, ie) = _Le e vars funns sol
      (el,dl,il,dvars,dfuns,ss) = _Ld l vars funns sol
  in (VAR(x,ee):el, de++dl, _U ie il, _U [x] dvars, dfuns, ss)
_Ld (FUN(f,_I,e):l) vars funns sol =
  let (ee,de,ie) = _Le e (_U vars (_Mkset _I)) funns sol
      (el,dl,il,dvars,dfuns,ss) = _Ld l vars funns sol
  in (el, FUN(f,assocdef f sol [] ++ _I, ee):de++dl,
      _U ie il, dvars, _U [f] dfuns, (f, ie):ss)

in let (newexpr, liftedfunns, _) = _Le expr [] [] []
in LETREC(liftedfunns, newexpr)

```

```

-- Library functions
-- (translation note: null and map are provided by Haskell;
-- itlist corresponds to foldr with a different parameter order)
--null [] = True
--null (_,_) = False
--map f [] = []
--map f (x:l) = f x: map f l
itlist f [] z = z
itlist f (x:l) z = f x (itlist f l z)
assocdef x [] def = def
assocdef k ((x,y):l) def = if k == x then y else assocdef k l def

-- Haskell implementation of the set operations
_Mkset l = l
_U l1 l2 = union l1 l2
_Is l1 l2 = intersect l1 l2
_Mem id l = id `elem` l

-- An example from Johnsson (1985).
-- let i = 5 in letrec f =  $\lambda x.f (i+i)$  in f (i*i)
example =
  LETREC(
    -- "let i = 5"
    [VAR("i", ref("5"))],
    -- "in letrec f =  $\lambda x.f (i+i)$  in f (i*i)"
    LETREC(
      [FUN("f", ["x"],
        -- "f (i+i)"
        APPL("f", [APPL("+", [ref("i"), ref("i")])])]),
      -- "in f (i*i)"
      APPL("f", [APPL("*", [ref("i"), ref("i")])])])
  )
where
  -- Variable references (and int literals) are represented as APPL(Id, [])
  ref :: Id -> Expr
  ref var = APPL(var, [])

main = do
  putStrLn ("input: " ++ show example)
  let lifted = _Lambdalift example
  putStrLn ("lifted: " ++ show lifted)

```