Automated Code Transformation for Context Propagation in Go

Adam Welc
Uber Technologies
San Francisco, USA
adam.welc@uber.com

ABSTRACT

Microservices architecture, which is increasingly being adopted by large technology companies, can accelerate development and deployment of backend code by partitioning a monolithic infrastructure into independent components. At the same time, microservices often compose into massively distributed systems, where analyzing the behavior of an individual service may not be enough to diagnose a performance regression or find a point of failure. Instead, a more global view of the entire computation may be required, where some form of global *context* is used to *trace* relevant information flowing through the system.

In the Go language, the recommended method of propagating this *tracing context* through service's code is to pass it as the first parameter to all functions on call paths where the context is used. This kind of code transformation, in addition to modifying function calls and function signatures, may involve modifications to other language constructs – performing it manually can be very tedious, particularly for large existing services. In this paper we describe an automated code transformation tool supporting this style of context propagation. We describe the design and implementation of the tool and, based on a case study using real production services, demonstrate that the tool can on average eliminate 94% of manual effort required to propagate tracing context through the code of a given service.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Go, automated code transformation, context propagation, services

ACM Reference Format:
Adam Welc. 2021. Automated Code Transformation for Context Propagation in Go. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3468264.3473918

1 INTRODUCTION

In recent years big technology companies with large backend infrastructures started moving from monolithic architectures to those based on microservices. A microservices architecture accelerates the development, deployment, and maintenance of the backend code. At the same time microservices often compose into massively distributed systems where monitoring, testing, and debugging can be a big challenge [16, 21, 23]. The advantages of this architecture come from the fact that services[1] are loosely coupled and can be developed largely independently, assuming the interactions with other services are known and well defined. This loose coupling, however, is also a source of challenges since a single service does not operate in a vacuum.

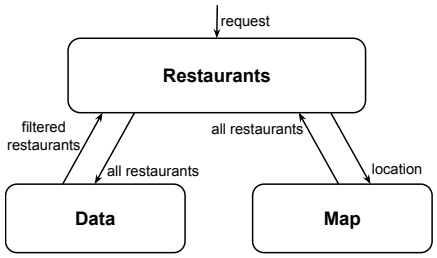

Figure 1: Simple services architecture.

For example, consider the simple services architecture in Figure 1, where the **Restaurants** service returns locations of all restaurants of a given type within a given area. The **Restaurants** service fulfills a request it receives by first obtaining a list of all nearby restaurants at a given location from the **Map** service, and then passing it on to the **Data** service containing detailed restaurant information so that only restaurants of a given type can be identified and returned. Even in this simple architecture an engineer observing that the **Restaurants** service is misbehaving cannot easily tell if the problem is with the **Restaurants** service itself or with other services participating in the computation. In the case of a more complicated microservices architecture, an engineer may not even know which other services are a part of the whole computation process. These types of problems motivated development of distributed tracing solutions, such as provided by Dapper [24] or featured in TChannel [27].

At a high level, as defined by the authors of Dapper [24], distributed tracing "needs to record information about all the work done in a system on behalf of a given initiator" (i.e., the **Restaurants** service in our example). This requires some form of *tracing context*[2] to store relevant information as the initiator's request propagates through the system and the work on its behalf is done by different services. The decision on what kind of information this context must contain is very much dependent on the details (e.g., functionality, architecture) of a given backend system. Context

[1]For brevity, in the remainder of this paper we will call microservices simply *services*.
[2]In the remainder of this paper, we will simply use the term *context*.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3473918

Listing 1: the Data service

```
func handler(ctx context.Context,
             s Service,
             r Request) {
  query := parseRequest(ctx, r)
  res := getResult(query)
  send(ctx, s, res)
}

func getResult(q Query) Result {
  ...
}

func execQuery(q Query) Result {
  return rdbms.exec(q)
}
```

can also be utilized in different ways, for example by a specialized tracing backend (e.g., TChannel uses the Zipkin backend [26]) or by a logging subsystem (to log information aggregated across different services).

On the implementation side the challenge is that the context for a given request must be propagated both across services (so that all services involved in the computation are captured, particularly that a given service may be handling requests on behalf of different initiators) and through the code of each individual service (so that all relevant information can be captured at appropriate places). The former issue can be solved by mandating a specific communication protocol. For example, outbound calls in TChannel [27] are required to pass the context as the first argument so that it is readily available in the receiving service's request handler. A solution for the problem of propagating context through the code of an individual service may vary between different programming languages, but the canonical method in Go, as recommended by Google, is to pass it on "as the first argument to every function on the call path between incoming and outgoing requests" [1]. When implementing a service from scratch, it may be relatively easy to maintain the required discipline, but with a large number of existing services and APIs evolving from context-unaware (i.e., not storing any data into the context) to context-aware (i.e., utilizing context to record some relevant information), this task may be daunting.

For example, consider the Go pseudo-code for the **Data** service in Listing 1. In Go, context is typically represented by the `Context` type provided in Go's `context` package [11], and in our example it is available as the first argument to the `handler` function called whenever the **Data** service receives a new request `r` from another service `s`. The request is parsed and then, possibly through a long sequence of calls (e.g., involving query optimization, caching, etc.), passed to the database engine (`rdbms`) to obtain the final result. Assume that initially the only information considered relevant for the trace was the query parsing duration (e.g., as a potential bottleneck). Further assume that at a later time the system architects decide to include database engine behavior in the trace by evolving its API to include context. In other words, the `rdbms.exec(q)` call needs to change to `rdbms.exec(ctx, q)`. The problem, of course, is that the `ctx` value must now be propagated all the way from the top-level request handler to the `execQuery` function, which may involve modifying a potentially large number of other calls that may not even be easy to identify via code inspection.

Performing such a code transformation task manually is not very appealing, particularly for a large number of services. At the same time, despite passing context explicitly as the first argument being the Go-recommended approach, there is no tooling to support it. Consequently, we built a code transformation tool that can ease the burden of propagating a context parameter through the code of a service by largely automating the process.

At a high level, the tool takes a list of the service's source files and a list of *leaf* library functions[3] representing a new context-aware API (in our running example from Listing 1 these would be calls to the database engine). The tool then builds a call graph for the service's code, discovers the call paths from the service's *entry points* (e.g., a handler serving incoming upstream requests) to the leaf functions, and finally injects context as the first parameter to all relevant function definitions and function calls. The intention is to use the tool whenever a new context-aware API is introduced, as this may lead to discovering additional call paths where a context parameter is not yet propagated.

On the surface, building such a tool may appear deceptively simple, but as usual the devil is in the details, and in the remainder of this paper we will discuss how Go language features have guided the design and implementation decisions behind an automated code transformation tool that can eliminate the majority of grunt work necessary to propagate context through the code of real production services. In particular, the paper makes the following contributions:

- We present an overview of our tool's design and implementation which utilizes the support available within the Go core toolchain to write static analysis tooling.
- We describe challenges with handling certain Go language features (e.g., interfaces, first-class functions) in the context of automated context propagation and discuss how our tool manages to overcome them.
- We discuss limitations of the tool and, since our tool is "best effort", the kinds of manual changes that may be required after the automated transformation is completed, so that the modified context-aware service code builds correctly and is capable of passing all the original tests.
- We evaluate the effectiveness of our tool based on a set of 17 production services demonstrating that it can on average eliminate 94% or more of the manual work necessary to implement context propagation for existing services. As part of our evaluation we also discuss how important it is to automatically handle specific Go language features.

The tool's code has been open-sourced and made available at https://github.com/uber-research/go-context-propagate.

[3]The Go language features both functions and methods, but due to their similarity in the remainder of this paper we will use the term "function" to describe both actual functions and methods, unless explicitly stated – please see Section 3.2 for more details on a distinction between a function and a method in Go.

2 OVERVIEW

In this section we describe the tool itself, the kind of code transformation it supports and how it is configured, as well as present an overview of its design and implementation. In all the examples (throughout the whole paper) context is encapsulated by the `Context` type provided in Go's `context` package [11], but in general any type can be used to represent context (it is part of the tool's configuration as described in Section 2.2).

2.1 Code Transformation

The first transformation of the service's code that the tool helps to accomplish, is to replace calls to non-context-aware (i.e., without the context argument) leaf library functions (see Section 1 for additional explanation) with the context-aware ones by introducing an additional context argument (at an arbitrary position) to a given existing call, and optionally renaming the function to be called. The tool also automatically discovers all call paths from the service's entry point(s) to the freshly introduced context-aware function calls. This is accomplished in the bottom-up fashion – the tool starts with the leaf functions and keeps transitively discovering their callers until either reaching a caller that already has a parameter of appropriate type in the right position, or until reaching the top of every call path for a given leaf function. Then the tool transforms all calls on these paths (and respective function definitions) to receive an additional context parameter.

For example, consider the following code fragment where the library function `lib.baz` is to be replaced with its context-aware equivalent `lib.ctxBaz`:

```
func foo(p bool) bool {
  return bar(p)
}

func bar(p bool) bool {
  return lib.baz(p)
}
```

The tool would transform this piece code into the following one (relevant changes in **bold**):

```
func foo(ctx context.Context, p bool) bool {
  return bar(ctx, p)
}

func bar(ctx context.Context, p bool) bool {
  return lib.ctxBaz(ctx, p)
}
```

As we can see, a call to `lib.baz` was changed to a call to `lib.ctxBaz` and the signature of `lib.ctxBaz`'s caller (function `bar`) was changed to include an additional context parameter. Finally, the signature of `bar`'s caller and the call to function `bar` itself were similarly modified.

Based on this simple example, building this type of code transformation tool seems like an easy task, but in reality it was quite challenging – in the following parts of the paper we will outline both the difficulties we had to face and the solutions we developed.

2.2 Configuration

The tool is configured using a JSON file containing information required to drive the code transformation. The simplest config file, describing the transformation example presented in Section 2.1 looks as follows:

```
{
  "CtxPkgPath"  : "context",
  "CtxPkgName"  : "context",
  "CtxParamName": "ctx",
  "CtxParamType": "Context",
  "LibPkgPath"  : "lib",
  "LibPkgName"  : "lib",
  "LibFns"      : [{"Name": "baz",
                    "NewName": "ctxBaz"}],
  "LoadPaths"   : ["app"]
}
```

The meaning of the fields in this file is as follows:

- `CtxPkgPath` and `CtxPkgName` – path and name of the package where context type is defined
- `CtxParamName` – name of the context parameter, used both as parameter for modified function definitions and as an argument for modified function calls
- `CtxParamType` – name of the context type
- `LibPkgPath` and `LibPkgName` – path and name of the package defining the leaf library functions
- `LibFns` – an array of leaf function specifications[4] where a specification includes a given function's unique per-package name (`Name` field) and an optional field `NewName` for cases when function renaming is required
- `LoadPaths` – a list of paths where the service's code resides, relative to the GOPATH [14] variable pointing to the service's development workspace

The JSON config file is the only input to the tool, but it is assumed that all dependencies of the service being transformed (e.g., external packages it uses) are available to the tool (i.e., they are available on the GOPATH) as all sources of a given service need to be type-checked before the transformation itself can take place.

2.3 Design and Implementation

In this section we present an overview of our tool's design and implementation – additional implementation details concerning specific Go language features are described in Section 3.

2.3.1 Design. We took a very pragmatic approach to designing this tool. Since the tool needs to find call paths from the service's entry points to the leaf calls, by definition it needs to analyze the program inter-procedurally. At the same time, we decided to try avoiding expensive whole program analyses and algorithms, such as data-flow analysis, partially because we needed the tool to work on sizeable code bases and partially because we felt like these may be only partially effective for a code transformation in the static setting (e.g., certain data flows can only be precisely identified at runtime as they depend on decisions made at runtime). In other words, we tried to strike the right balance between the complexity of

[4] Specification of *leaf methods* is also supported, but we omit it for brevity.

the tool that influences transformation times and the effectiveness of the tool in reducing the amount of grunt work needed to make services context-aware.

Ultimately, our main focus ended up being on the control flow analysis and the ability to iterate over and inspect certain language constructs (e.g., globally defined types, functions, etc.). This helped us keep the transformation times low and make the tool quite effective. The code transformation supported by the tool is, however, "best-effort" – we say that the transformation process is automated, rather than saying it is (fully) automatic, as due to the static nature of the analyses it uses (e.g., statically constructed call graph algorithm) the tool sometimes generates code that requires manual intervention to correct it. Of course, the goal is to minimize the amount of manual code changes that are required – please see Section 4 for a description of the tool's limitations and Section 5 for the details of the tool's evaluation.

The workflow including optional manual modifications is supported by the intended deployment model for the tool – transformed version of the code is compared with the original one to create a set of diffs submitted to the code review system where they can be further modified. Even if manual intervention is not necessary, this allows for the final verification of the modified code by the service developers before it is used in production.

Our second design goal was to make the tool relatively simple to use by automating as much of the code transformation process as possible. Upon encountering a more problematic situation (e.g., having to deal with third-party code), the tool attempts to make an educated guess on what the correct action should be and informs the developers about the decision it made, so that the code can be altered manually if need be – please see Section 4.2 for a description of the tool's heuristics.

2.3.2 Implementation. The code transformation process consists of two phases: analysis and transformation. The implementation utilizes support for building static analysis tools coming with the Go language itself in the form of several utility packages.

The analysis phase consists of the following steps:

- loading source code and translating it to abstract syntax trees (ASTs) – with the help of the `go/packages` package
- translating the ASTs to the SSA form [2] – with the help of the `golang.org/x/tools/go/ssa/ssautil` package
- type-checking and inspecting the SSA form – with the help of the `golang.org/x/tools/go/ssa` package
- creating a call graph (nodes representing functions and edges representing function calls) from the SSA form – with the help of the `golang.org/x/tools/go/callgraph` package utilizing the Rapid Type Analysis (RTA) algorithm provided by the `golang.org/x/tools/go/callgraph/rta` package
- traversing the call graph bottom up, starting with the leaf calls, to identify and record portions of the service's code (i.e., function signatures, call sites, data types, etc.) that need to be modified to support context propagation – this is where the "core" transformation logic is implemented (see Section 3)

The transformation phase consists of of the following steps:

- traversing and modifying the ASTs according to the information collected during the analysis phase – with the help of the `golang.org/x/tools/go/ast/astutil` package
- transforming ASTs back to the source code with the output formatted using the algorithm of the `go format` command [5] – with the help of the `go/format` package

One of the important parts of the core transformation logic implemented as the last step of the analysis phase was handling Go language features beyond function calls and definitions – we will discuss the details in the next section.

3 HANDLING GO LANGUAGE FEATURES

Despite its similarity to C, Go language features extend beyond simple C-like statements and function calls, and include first-class functions, interfaces, import statements, etc. A code transformation tool that aims to work with real-life programs should at least attempt to handle these features or risk increasing manual effort required to implement a given transformation. As we will see in the following parts of this section, modifying function definitions to alter their signatures may have a profound effect on these other Go language constructs.

3.1 First-class Functions

Functions are first-class constructs in Go and a given function can be called directly or, just like in the example below, using a function-type parameter (please assume that `baz` is a leaf library function):

```
func foo(p bool) bool {
  return bar(p, qux)
}

func bar(p bool, f func() bool) bool {
  return f()
}

func qux(p bool) bool {
  return lib.baz(p)
}
```

In order to correctly transform the code fragment above, we not only have to add a context parameter to all relevant function definitions and call sites (as described in Section 2.1), but also modify the parameter type used to pass function `qux` to `bar` in `foo`:

```
func foo(ctx context.Context, p bool) bool {
  return bar(ctx, p, qux)
}

func bar(ctx context.Context, p bool,
         f func(context.Context) bool) bool {
  return f(ctx)
}

func qux(ctx context.Context, p bool) bool {
  return lib.ctxBaz(ctx, p)
}
```

In order to support this transformation, the tool must inspect definitions of all functions[5] to discover and modify function-type parameters, in addition to traversing the call graph to identify all call paths for context propagation. A similar, though more complicated strategy has to be applied to correctly handle Go interfaces.

3.2 Interfaces

Interfaces in Go serve a similar role as in other popular programming languages (e.g., Java) – they declare a set of method signatures defining the behavior of a certain entity abstracted by the interface. A method in Go is similar to a Go function, but its signature in addition to the name and parameters, also specifies this method's receiver type, which describes a concrete entity whose behavior this particular method defines. In the following code fragment, we define method `qux` with the receiver type `Entity`[6] and we say that `qux` is in the *method set* of `Entity`[7] (again, please assume that `baz` is a leaf library function):

```
type Entity struct {}

func (Entity) qux(p bool) bool {
  return lib.baz(p)
}
```

An interface in Go specifies its own method set by declaring method signatures. In the following code fragment we define interface `AbstractEntity` declaring a single method `qux`:

```
type AbstractEntity interface {
  qux(p bool) bool
}
```

In Go, the "implements" relationship between receiver types and interfaces is implicit - a receiver type implements an interface if its method set is a super-set of the method set declared by the interface. Consequently, in the example above, type `Entity` implements interface `AbstractEntity`.

Since methods and interfaces may form the "implements" relationship, when changing the signature of the former, the tool may have to change the definition of the latter - in the example above, if a call to `lib.baz` is changed to call this function's context-aware equivalent, the tool not only has to add context parameter to method `qux`, but also modify declaration of `qux` inside interface `AbstractEntity`. Otherwise type `Entity` will no longer implement `AbstractEntity`. The resulting transformed code fragment looks as follows (definition of type `Entity` remains unchanged):

```
func (Entity) qux(ctx context.Context,
                  p bool) bool {
  return lib.ctxBaz(ctx, p)
}

type AbstractEntity interface {
  qux(ctx context.Context, p bool) bool
}
```

[5]Iteration over all functions of a given service is supported by the `golang.org/x/tools/go/callgraph` package

[6]We define type `Entity` as an empty structure for brevity.

[7]There may be more methods with the same receiver type.

After this transformation is completed, however, there may be other methods that implement the modified interface but that do not use, and thus do not require, an additional context argument. Still, their signatures must be updated to "match" definitions of the modified interfaces.

3.2.1 Matching Modified Interfaces. Unconditionally changing signatures of all methods whose receiver type implements a given interface would certainly solve the problem, but it is likely unnecessary as we only need to modify methods that are actually called using this interface (some of the "implements" relationships, due to their implicit nature, may be unintentional). Continuing with the previous example, let us introduce another `qux` method with a different receiver type and a polymorphic call site:

```
type OtherEntity struct {}

func (OtherEntity) qux(p bool) bool {
  return p
}

func main() {
  v := OtherEntity{}
  v.qux(true)
}
```

In this extended example, type `OtherEntity` also implements interface `AbstractEntity`, necessitating modification of new `qux`'s signature. However, since the new `qux` method does not actually use context, the tool special-cases transformation of its signature and of its call site in the `main` function:

- it inserts an additional parameter to `qux` with the *blank identifier* (i.e., _) instead of a specific name to indicate that the parameter value will never be used
- it inserts "invalid" (or, in other words, "artificial") context as an argument at the call site – its value is specified in the JSON config file in the `CtxParamInvalid` field that must contain a valid Go expression (e.g., a function call)

The resulting transformed code fragment looks as follows ("invalid" context is specified as a call to the `Background` function of the `context` package, and the definition of type `OtherEntity` remains unchanged):

```
func (OtherEntity) qux(_ context.Context,
                       p bool) bool {
  return p
}

func main() {
  v := OtherEntity{}
  v.qux(context.Background(), true)
}
```

Below we describe the algorithm used to find and update methods that need to be modified.

3.2.2 Algorithm. Technically, we could analyze the entirety of the service's source code, find all polymorphic call sites where the tool injects an additional context argument, discover definitions of methods that can be called at these call sites, and finally update

signatures of all the discovered methods. This type of algorithm would likely be costly and possibly unnecessary considering the typical use of polymorphic calls, at least in the source code of the services existing at our company. We observe that an interface is typically used as a parameter to a function which in turn makes a polymorphic call using this interface. A typical scenario here is for an interface to be implemented by the code running in production and by the "mock" code created (often auto-generated) for testing purposes. As a result, we find a simpler algorithm, requiring analysis of only a small portion of the source code, to be quite effective – instead of finding interfaces to match by looking for all polymorphic call sites, find the relevant interfaces by looking for interface-typed function parameters.

Here is a more detailed description of the algorithm:

(1) During the call graph traversal record interfaces that will be eventually modified as a result of injecting a context parameter to the method's signature.[8] In addition to remembering the set of interfaces, also remember which methods will be modified in each interface.
(2) After the graph traversal is finished, iterate over all functions in the program and record all functions with a parameter whose type is in the modified interfaces set that has been recorded in step 1. In addition to remembering the function itself, remember which parameter matches one of the modified interface types.
(3) For all functions found in step 2, find their call sites and record the value of an argument at the call site that matches the position of the parameter recorded in step 2 – this argument value can be used to identify and record a receiver type passed as an argument (matching a given interface-type parameter).
(4) For all receiver types recorded in step 3:
    (a) Modify the signature of methods in the methods set of a given receiver type (matching those modified in the interface this receiver type implements, as recorded in step 1) to take a context parameter
    (b) At all call sites for methods with modified signatures pass "invalid" context as the first argument.

We acknowledge that this algorithm is tailored to work well with a specific interface usage pattern. However, this pattern is not only popular across multiple services developed by different developers at our company, but more broadly it is also documented [3] as a well-known solution used in some of the most popular open-source Go mocking frameworks, such as gomock [10]. This makes us believe in this pattern's generality but alternatives, such as the more expensive one described earlier in this section, can clearly be considered as well if need be.

3.3 Import Statements

If the tool changes a signature of a function in a given file to include an additional context parameter, in order for this file to successfully compile afterwards it needs to contain an import statement for the package where the context type is defined. If such *context package import* statement is not present, the tool will inject it automatically.

[8]Checking if a given receiver type implements a given interface is supported by `go/types` "standard" package

In particular, considering the example in Section 2.1, the tool would inject a context package import statement for package `context` (the `CtxPkgPath` field in the config file determines this package's path as described in Section 2.2). The tool is also smart enough to recognize a situation when an optional package alias is specified as part of the existing context package import, and will use this alias instead of the package name specified in the config file (`CtxPkgName` field). An existing import may, however, also refer to a *different* package with the same name as the one specified in the config file. In this case, a developer has an opportunity to define a globally unique alias to avoid the name clash - it is specified using the `CtxPkgAlias` field in the config file.

Before describing the second type of injected import statements we need to introduce the notion of context expressions.

3.3.1 Context Expressions. In the original example (Section 2.1) we have a "raw" context value passed to function `lib.ctxBaz`, but in some cases it may be necessary to pass a *context expression* instead, that is an expression that somehow utilizes the context (e.g., as an argument to a function call). For example consider the following extension to the example from Section 2.1:

```
func bam(p bool) bool {
  return lib.bazExp(p)
}
```

Assume that we want to transform this code fragment to replace a call to `lib.bazExp` with a call to its context-aware equivalent `lib.ctxBazExp` taking as an argument the following context expression: `other_lib.setTimeout(ctx, 1000)` (the actual semantics of the expression is irrelevant for the example, beyond the fact that the function call it contains is defined in package `other_lib`). The resulting code would look as follows:

```
func bam(ctx context.Context, p bool) bool {
  return lib.ctxBazExp(
   other_lib.setTimeout(ctx, 1000), p)
}
```

In order to be successfully compiled, the transformed code fragment needs to import the `other_lib` package. Both a context expression and optional *context expression import* statements can be defined in the config file as part of the leaf function specification in the `LibFns` array (see Section 2.2 for description). Similarly to the context package import statement injection described above, a context expression import statement can be optionally aliased to avoid name clashes. The complete leaf function specification for the Section 2.1 example extended to accommodate function `bam` looks as follows, where `some_dir` is `other_lib`'s package path (`Name` field) and `other_lib` is the optional alias (`Alias` field) [9]:

```
"LibFns" :
 [{ "Name": "baz", "NewName": "ctxBaz" },
  {
    "Name": "bazExp",
    "NewName": "ctxBazExp",
    "CtxImports": [{"Import": "some_dir",
                    "Alias": "other_lib"}]
  }]
```

[9]The rest of the config file remains unchanged.

To summarize, considering the extended example, the actual import statements injected into the file where none previously existed would be as follows:

```
import "context"
import other_lib "some_dir"
```

One could consider injecting import statements using another simpler tool. Please note, however, that an import cannot be injected "blindly" – one has to consider per-file aliasing implications described above and the injection must happen *only* if the import statement is needed by the code in the modified file, otherwise the Go compiler will complain about the import statement being superfluous.

4 LIMITATIONS

In this section, we describe issues that the tool, being "best-effort", cannot currently handle, and which require manual intervention on the developer's side. We also discuss some heuristics that the tool uses when encountering some more problematic situations and how these are communicated to the developer. Please note that fixes for a lot of issues requiring manual correction are relatively easy to perform as they only require inspection of a single function.

4.1 Manual Corrections

In this section we describe situations when, after the automated code transformation is completed, selected manual modifications to the service's code are required to have it built and run correctly.

4.1.1 Spurious Context Injections. No static call graph construction algorithm is 100% accurate (e.g., due to reflective calls that cannot be discovered statically). In particular, the resulting call graph may contain an edge from a function definition to a call site, particularly a polymorphic call site, even if this function is never called at this call site. In such cases, the tool would inject a context parameter at this call site and the resulting code would fail compilation. This has to be rectified by the developer – the context argument must be removed.

4.1.2 Variadic Signatures. The tool can place a context argument (or a context expression argument – see Section 2.2 for explanation) for a given leaf call at an arbitrary argument position – if undefined in the leaf function specification portion of the config file (`ArgPos` optional field), it defaults to the first argument. This poses a certain problem for variadic signatures (i.e., signatures of functions that take a variable number of arguments) – they are handled correctly if arguments are specified one-by-one at the call site, but not if a variadic argument is passed as an array (both argument passing methods are valid in Go [15]). Technically the tool could handle this, at least partially (e.g., automatically handle argument injection at the first or last position), but the manual fix is typically very straightforward and results in nicer code than what would be auto-generated in this case – we often observe an argument array being constructed right before the call, which is a natural place to inject an additional argument, but it is difficult to handle such injection automatically in the general case.

4.1.3 Return Values. At this point, the tool does not modify a return type even if what is returned is a function and this function's signature has been modified to take an additional context parameter. It is non-trivial to auto-generate code for this situation in the general case (e.g., not all return statements necessarily return a function with modified signature), but is conceivable to support it in the future if the real need arises, possibly in a limited fashion.

4.1.4 Reflection. Similarly to many other code transformation tools relying on static analysis, our tool does not handle reflection. For example, functions in Go can be looked up using their names stored in arrays, which in the general case makes it difficult to discover which function is being used at a reflective call site.

4.2 Heuristics

In this section we describe some heuristics that the tool uses when encountering some more problematic situations. The decisions made by the tool are subject to the developer's verification – the tool can generate an appropriate description including a file name and a line number in the JSON format to be post-processed for integration with the development workflow (in particular, we post this information as inline comments in our code review system).

4.2.1 Third-party Code. The code of any given service often uses external libraries whose code can be accessed for analysis purposes but cannot be easily modified. This has consequences on what kinds of modifications are allowed in the actual service's code. In particular, since functions in Go are first-class and can be used as parameters, they may be passed as arguments to functions defined by the third-party code. Similarly, the receiver type of a method in the service's code may implement an interface defined by the third-party code. In both cases, if the function/method is identified by the tool as one requiring an additional context parameter, its signature cannot be modified since, in the general case, the third-party code cannot be modified either. The tool handles this by injecting a local "invalid" context (see Section 3.2.1 for description) variable at the beginning of the function/method instead of injecting a context parameter into the signature.

4.2.2 Alternative Entry Points. The main entry point to the service's code is a request handler accepting requests from upstream services. Typically, it is the handler that provides context to be propagated throughout the service's code as part of the data coming from the upstream. The tool finds the call paths requiring context by starting with the leaf functions and discovering the transitive callers of these functions until the request handler is reached. However, the request handler does not have to be the only entry point to the service. In particular, an `init` function used for package initialization [13] or test functions executed by the test harness can constitute alternative entry points. In these types of situations, context is typically not available and, similarly to Section 4.2.1 above, the tool inserts "invalid" context variable to be used for subsequent function calls.

4.2.3 Function Containers. Functions in Go are first-class constructs and can be stored in containers – in particular, they can be array elements and can also be used as both keys and values in maps. Changing signatures of such functions to accommodate an additional context parameter would cause a compilation error as

the container types and the function types would no longer match. There are two solutions to this issue:

(1) Avoid changing signatures of such functions by injecting a local variable defining "invalid" context to be used for subsequent calls
(2) Inject a context parameter to all functions that can be used by any container and change type of the containers

If a service stores a lot of functions in containers, tracking all such functions to implement either solution manually could be tedious. As a convenience, the tool supports solution 1 above as part of the automated transformation, and injects an "invalid" context variable to all functions whose signatures match the types used by any of the containers. This is similar to the tool's handling of methods implementing already modified interfaces (see Section 3.2.1) but is considered a heuristic as there are at least two valid methods of solving this issue.

5 EVALUATION

The main goal of the experimental evaluation was to determine the effectiveness of the tool using some measure of how much of the manual effort otherwise required to implement the context propagation transformation could be eliminated when using the tool. Additionally, we wanted to gain an insight into how important it is to support various Go language features we described in Section 3.

5.1 Setup

Performing a user case study was beyond the scope of this paper, instead in our evaluation we use rather simple metrics that we believe nevertheless give at least an idea of how effective the tool is compared to doing the code transformation fully manually. The main metric is a ratio of the number of LOCs (lines of code) that the developer had to modify manually after the automated code transformation was completed (if any – otherwise zero lines had to be manually modified) to the total number of LOCs that had to be changed (including both the code automatically modified by the tool and optional manual developer changes) for the service to compile and pass all tests – see Section 5.2.1 for the description of this main result.

The API we are making context-aware in our evaluation is the logging API defined in the popular open-source `zap` package [28]. An internal package developed in our company provides a function that can be used to form a context expression passed to the logging calls of the `zap` package. The internal package is not yet open-sourced but neither its implementation nor the content of the context expression have any bearing on this evaluation – the only relevant part is the description of which calls to `zap` package's logging methods are being transformed and what is the position where the context expression is injected:

- receiver type Logger: DPanic, Debug, Fatal, Info, Panic, Warn (second position), Error (last position)
- receiver type SugaredLogger: DPanic, DPanicf, DPanicw, Debug, Debugf, Debugw, Error, Errorf, Errorw, Fatal, Fatalf, Fatalw, Info, Infof, Infow, Panic, Panicf, Panicw, Warn, Warnf, Warnw (last position)

Table 1: Services statistics and code transformation effort results

	LOC	LEAF	AUT add	AUT rm	MAN add	MAN rm
s1	643339	733	895	747	0	0
s2	568572	100	211	188	2	2
s3	478560	133	1160	995	8	8
s4	460850	135	182	163	3	3
s5	360307	65	132	124	1	1
s6	287504	325	725	630	10	12
s7	194139	10	52	35	0	0
s8	101345	79	381	279	12	15
s9	67794	12	24	19	0	0
s10	46324	52	140	114	8	8
s11	14883	148	244	219	0	0
s12	13570	27	77	74	0	0
s13	11894	13	124	95	1	1
s14	11315	77	208	200	104	57
s15	10407	51	66	51	0	0
s16	8108	35	153	130	19	16
s17	4176	23	25	23	0	0

We ran the tool on a standard development machine (MacBook Pro, 6 2.6GHz Intel i7 cores, 32GB RAM, macOS Mojave 10.14.6).

5.2 Results

We have evaluated our approach on the set of 17 production services. We use artificial identifiers (**s1**, **s2**, etc.) to differentiate between them as these are proprietary services used within our company. The code of these services varies in size and in the number of the leaf API calls that are used as starting points for context propagation, as described in the first 2 columns in Table 1, which is sorted in the order of the number of lines of code (LOCs). The numbers in the **LOC** column of the table have been obtained using the `cloc` command [10] and reflect actual code implementing the logic of the service (blank lines and comments are excluded). As we can see, some of the services are quite large, revolving around half a million lines of code. The number of leaf calls is presented in the **LEAF** column in Table 1.

The code transformation times did not play a significant role in this experiment as the whole automated transformation process takes less than 18s on average (averaged over 10 runs) for the largest service we transformed (**s1**).

5.2.1 Transformation Effort. The main result of the evaluation is presented in Figure 2, where we compare the number of manual changes to the number of total changes required to transform a service to the point when it builds and passes all its original tests. We present two ratios computed by comparing diff files representing the automated changes and the following optional manual changes – the left bar for each service is for the ratio of lines added and the right bar is for the ratio of lines removed (as computed using the *diffstat* command [11]).

[10] https://linux.die.net/man/1/cloc
[11] https://linux.die.net/man/1/diffstat

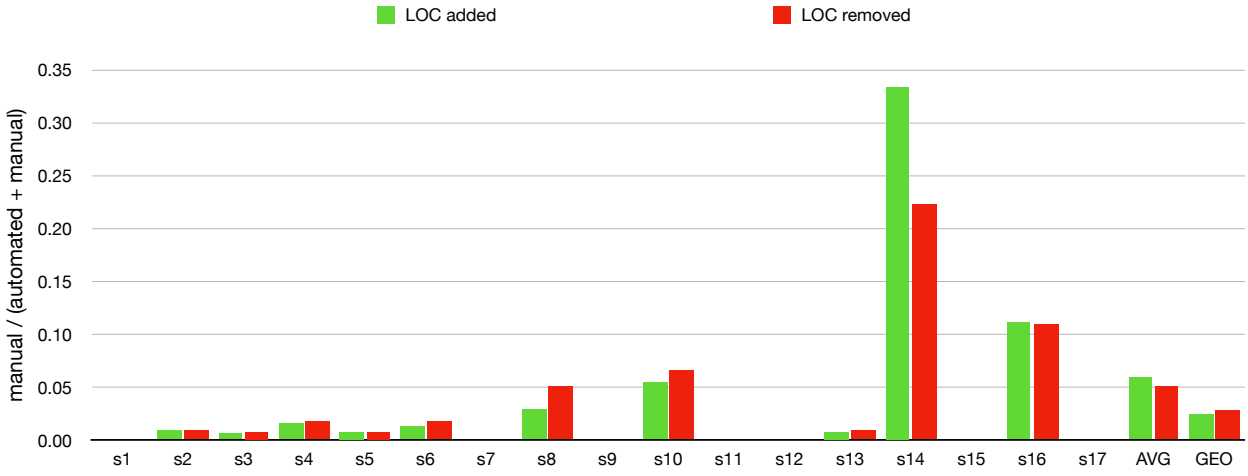

Figure 2: Ratio of manual to all (automated + manual) changes.

Table 2: Code transformation results (language features)

	SIG mod	CALL mod	PRM mod	INT mod	MET mod	IMP add
s1	3	744	0	0	0	95
s2	13	135	0	0	0	56
s3	157	715	0	9	23	135
s4	1	136	0	0	0	45
s5	10	84	0	2	4	22
s6	45	495	1	8	8	135
s7	7	28	0	0	0	12
s8	65	189	0	5	9	58
s9	2	14	0	0	0	8
s10	17	88	0	0	0	24
s11	20	179	0	2	2	24
s12	3	71	0	0	0	2
s13	19	77	0	0	0	9
s14	14	113	0	1	1	21
s15	0	51	0	0	0	11
s16	25	90	0	2	2	20
s17	0	23	0	0	0	2

In terms of the arithmetic mean (AVG bars in Figure 2), for both lines added and removed the ratio of the number of manual changes to the number of total changes is at most 0.6. Consequently, according to this metric, the average amount of manual effort required is at most 6% of the total effort that would be required to transform the service. The result is even better when considering the geometric mean (GEO bars in Figure 2) – the ratios for both lines added and removed equal 0.3. Please also note that over 1/3rd of all services, including some really large ones (e.g., **s1** in Table 1), did not require any additional changes after being transformed by the tool. When computing both means, we have excluded services which did not require any manual changes to avoid skewing the result.

We fully understand that the metric we use is simple, and we do not claim these numbers as a "hard" result, but we do believe that even this metric gives a strong indication that the tool is capable of significantly reducing the amount of effort required to transform production service code.

The "raw" numbers indicating the effectiveness of the tool are presented in the last 4 columns in Table 1: the number of lines added and removed according to the diffs representing auto-generated changes (**AUT add** and **AUT rm** columns) and the number of lines added and removed according to the diffs representing changes implemented manually by the developer (**MAN add** and **MAN rm** columns). Another way of looking at these numbers is that columns **AUT add** and **MAN add** represent the number of service code line *modifications* (every line removal in a diff file is matched with an addition, but not vice versa), and the number of service code line *additions* is represented by subtracting numbers in the "rm" columns from number in the "add" columns. Comparing the total number of automatic modifications with the total number of manual modifications yields an additional effectiveness metric of the tool – 4 manual code line modifications were needed per 100 automatically modified lines of code.

As we can see, it is not uncommon for a service to require hundreds of lines of code to change, and making that many changes by-hand could be really tedious. Admittedly, some relatively large services require changes only in the order of tens of lines of code (e.g. **s7**), but then all the services in this case study that required less than 100 lines of code to change were transformed by the tool fully automatically, without requiring any manual intervention.

5.2.2 Manual Changes. Among all services used for the experimental evaluation, we needed to manually correct (see Section 4.1 for details) 77 cases of spurious context injections, 18 variadic signatures, and 4 return values of a function type (no issues related to reflection that required a manual intervention were discovered). A significant number of manual corrections, combined with an overall small total number of modifications, created two main outliers in Figure 2: **s14** (mainly due to manual variadic signature corrections) and **s16** (mainly due to manually correcting return values of a function type). All required manual corrections were signaled by the compiler (as failed builds) and were therefore relatively easy to perform – in terms of subjective difficulty levels (details omitted

for brevity), the most trivial issues to address were related to variadic signature, followed by spurious context injections, followed by return values of a function type.

5.2.3 Language Features. The first two columns in Table 2 describe how many automatic modifications to function signatures and function calls were performed, and in the remaining columns we report on how many modifications were due to handling language features as described in Section 3:

- **SIG mod** - number of modified function signatures
- **CALL mod** - number of modified call sites
- **PRM mod** - number of modified function-type parameters
- **INT mod** and **MET mod** - number of modified interfaces and number of modified methods within these interfaces
- **IMP add** - number of added import statements

In terms of "obvious" modifications, as we can see, we have on average tens of function signatures modified, with the maximum number reaching over 150 modifications (**s3**). At the same time, the number of modified calls is almost an order of magnitude higher, reaching over 700 modifications (**s1** and **s3**). Interestingly, for some services (**s15** and **s17**), we see no function signatures modified and yet we see modifications to function calls. This is because the latter number includes calls to the leaf functions – these services simply already have context of appropriate type propagated on all required call paths and the only action that the tool had to take was to inject the context expression to the leaf calls. As you can see, for these services the number of leaf calls in Table 1 is the same as the number of modified calls in Table 2 (the remaining changes are due to auto-injecting import statements). While it is not inconceivable to manually modify hundreds of function signatures and calls for a single service, we believe that even with the help of modern IDEs this would not easily scale to a large number of services.

In terms of the Go language features that the tool supports, we can see that a certain non-trivial number of interfaces and interface methods had to be modified, at least for some services (**INT mod** and **MET mod** columns in Table 2). The most demanding service here is **s3** with 9 interfaces and 23 interface methods modified.

The tool also automatically injects import statements if and only if it is necessary (see Section 3.3 for the explanation) – it is often required to insert tens, and sometimes even hundreds of them (**IMP add** column in Table 2). On the other hand, at least for the services in our case study, we only very occasionally see a function-type parameter that needs to be modified (**PRM mod** column in Table 2). In summary, it seems like automatically handling interface modifications and import injections was quite important, but handling function-type parameters, while still useful, significantly less so.

6 RELATED WORK

In terms of comprehensive solutions for Go services context propagation problem, the only alternatives to using a code rewriting tool such as ours involve "mimicking" goroutine-local storage and using it to store context. There are at least a few projects [6, 22, 25] supporting this type of solution and they all work similarly. The main idea is to somehow (e.g., by using undocumented language features or stack manipulation and traversal) obtain a per-goroutine identifier and use it as a key to a map. These types of solutions, however, tend to be brittle and can have unforeseen performance consequences, plus assigning goroutines unique identifiers goes against recommended Go programming practices [12].

With respect to the code transformation and refactoring tools, the body of work covering different techniques for different programming languages is too large to cite, with separate books [4] and surveys [20] covering only portions of the work done in this area. However, as far as we can tell, no existing tool supports context propagation as a single end-to-end transformation. A partial support for selected analyses or code rewriting tasks that can help with context propagation does exist, but not necessarily for Go.

In particular, some IDEs supporting C# (but not Go) development (e.g., Rider[19]) provide the *Introduce Parameter* refactoring [18]. This refactoring supports "lifting" an arbitrary expression from the callee's body to its callers, where it is used as an additional argument passed to the callee. Similarly to the context propagation transformation, this involves automatically injecting a parameter at all the call sites and to the signature of the callee. The refactoring is also powerful enough to modify relevant interfaces if necessary, similarly to how the context propagation transformation does it (see Section 3.2). However, in C# the "implements" relationship is explicit in the type system which simplifies implementation of the refactoring. Additionally, this refactoring works on a single method definition at a time, which could result in hundreds of separate modifications. Finally, it requires the code to be correctly typed at each step, which does not hold for context propagation (a "lifted" context argument is undefined in the caller until the subsequent refactoring step is completed).

Generally speaking, Go is a relatively new language with a somewhat limited tooling support. In particular, existing code rewriting tools (e.g., eg [7], gofmt [8], gorename [9]) support general but simple code modifications largely based on pattern matching. Consequently, no existing tooling can assist a developer in implementing context propagation with the same or similar effectiveness to our code transformation tool. However, developers are not completely on their own when rewriting code to support context propagation as they can at least take advantage of code navigation and editing capabilities of modern IDEs. For example, GoLand [17], one of the most popular IDEs for Go, supports discovery of a given function's callers and of the "implements" relationship between receiver types and interfaces. This can partially simplify identification of the code fragments that need to be modified, but the manual effort involved, particularly for large change sets, can still be overwhelming.

7 CONCLUSIONS

In this paper we tried to provide an insight into the design and implementation of a practical static code transformation tool that is capable of significantly reducing the amount of manual effort required for implementing context propagation for real-life production services. We have presented results of the tool's evaluation supporting our claims about its effectiveness, discussed its limitations, and highlighted usefulness of automatically handling selected Go language features. The tool's code has been open-sourced and is available at https://github.com/uber-research/go-context-propagate. Last but not least, we would like to thank everyone that contributed to this work, with a special thanks to Lazaro Clapp.

REFERENCES

[1] Sameer Ajmani. 2014. Go Concurrency Patterns: Context. https://blog.golang.org/context

[2] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* (1991). https://doi.org/10.1145/115372.115320

[3] Che Dan. [n.d.]. A GoMock Quick Start Guide. https://betterprogramming.pub/a-gomock-quick-start-guide-71bee4b3a6f1

[4] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional.

[5] Andrew Gerrand. 2013. go fmt your code. https://blog.golang.org/go-fmt-your-code

[6] Massimiliano Ghilardi. 2018. gls. https://github.com/cosmos72/gls

[7] Go Team. [n.d.]. Command eg. https://godoc.org/golang.org/x/tools/cmd/eg

[8] Go Team. [n.d.]. Command gofmt. https://golang.org/cmd/gofmt

[9] Go Team. [n.d.]. Command gorename. https://godoc.org/golang.org/x/tools/cmd/gorename

[10] Go Team. [n.d.]. gomock. https://github.com/golang/mock

[11] Go Team. [n.d.]. Package context. https://golang.org/pkg/context/

[12] Go Team. 2018. Why is there no goroutine ID? https://golang.org/doc/faq#no_goroutine_id

[13] Gopher Guides. 2019. Understanding init in Go. https://www.digitalocean.com/community/tutorials/understanding-init-in-go

[14] Gopher Guides. 2019. Understanding the GOPATH. https://www.digitalocean.com/community/tutorials/understanding-the-gopath

[15] Uday Hiwarale. [n.d.]. Variadic functions in Go. https://medium.com/rungo/variadic-function-in-go-5d9b23f4c01a

[16] David Hubbell. [n.d.]. Top 4 Pros and Cons of Microservices Architecture. https://www.spkaa.com/blog/top-4-pros-cons-microservices-architecture/

[17] JetBrains. [n.d.]. Goland. https://www.jetbrains.com/go

[18] JetBrains. [n.d.]. Introduce Parameter. https://www.jetbrains.com/help/rider/Refactorings__Introduce_Parameter.html

[19] JetBrains. [n.d.]. Rider. https://www.jetbrains.com/rider

[20] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering* (2004). https://doi.org/10.1109/TSE.2004.1265817

[21] Anna Monus. 2018. What are microservices? The pros, cons, and how they work. https://raygun.com/blog/what-are-microservices

[22] JT Olio. 2018. gls. https://github.com/jtolds/gls

[23] Good Rebels. 2018. To go or not to go micro: the pros and cons of microservices. https://medium.com/@goodrebels/to-go-or-not-to-go-micro-the-pros-and-cons-of-microservices-7967418ff06

[24] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. https://research.google.com/archive/papers/dapper-2010-1.pdf

[25] Tyler Stillwater. 2015. gls. https://github.com/tylerb/gls

[26] Twitter. [n.d.]. Zipkin. https://blog.twitter.com/engineering/en_us/a/2012/distributed-systems-tracing-with-zipkin.html

[27] Uber. [n.d.]. TChannel. https://www.jetbrains.com/help/rider/Refactorings__Introduce_Parameter.html

[28] Uber. 2019. zap. https://github.com/uber-go/zap