Verifying Interfaces Between Container-Based Components

Or... A Type System By Any Other Name

Christopher S. Meiklejohn
Université catholique de Louvain
Instituto Superior Técnico

Zeeshan Lakhani
Northeastern University

Peter Alvaro
UC Santa Cruz

Heather Miller
Northeastern University
École Polytechnique Fédérale de Lausanne

Abstract

Container-based programming has become the *lingua franca* for building distributed applications. More than ever before, application developers not only have to reason about the behavior of containerized components, but also the composition of these components that are increasingly presented to the developer as prepackaged, "black box" solutions. As a result, there is an increasing need for ways to ensure that composition preserves individual component invariants. In this work, we present a methodology for ensuring correctness of compositions for containers based on existing programming language research, namely research in the field of type systems. We motivate our solution examining two use cases, both leveraging existing public cloud and edge infrastructure solutions and industry use cases.

1 Introduction

Modern distributed applications position the container as the fundamental unit of computation, giving application developers the ability to focus on higher-level design patterns instead of the low level details in code. This is evidenced by the growing popularity of microservice architectures, i.e., architectures in which applications are constructed through the composition of containerized software components [3]. Furthermore, commercial cloud providers incentivize application developers to deploy and compose "black box" container-based components on their platform by providing registry marketplaces like Docker Hub, container deployment systems like Google's Container Engine, and "serverless" infrastructures like Amazon's Lambda. Not only are these techniques pervasive in cloud computing, but state-of-the-art infrastructures for edge computing, such as Amazon Lambda@*Edge*, also promote this design philosophy by deploying the same or equivalent containerized components to the edge. Therefore, the programmers of today progressively have to reason about the correct composition of existing off-the-shelf solutions.

Containers are no longer being treated simply as a deployment vehicle, but instead as objects in a distributed and orchestrated object-oriented programming environment [3]. Container-driven development is successful today because it allows isolated teams of developers to independently iterate on components of a larger application as long as interactions at the boundary between components is preserved. Even with invariants playing a crucial role, containerized systems are increasingly made up of heterogeneous and diverse communication protocols, programming languages, application frameworks, and subsystems working in concert. To tackle this disparity, developers working on containers typically rely on providing and consuming application programming interfaces (APIs) across containers, supplying different functionality as a guide on how to properly compose them. However well documented these interfaces are, they are usually solely specified via implementation and not mechanically checked. Therefore, the onus of correctly understanding these interfaces, and ensuring that their composition is correct, is placed on the application developer.

To address these problems, we take a cue from type theory and propose the use of universal polymorphism in interface specification. We present two use cases that motivate the two types of universal polymorphism identified by Cardelli and Wegner [4], subtype (inclusion) and parametric polymorphism, respectively.

2 Motivating Example: Apache Kafka

Our first example examines a data loss bug, discovered by Kingsbury [8] and formalized by Alvaro et al. [1] where the Apache Kafka fault-tolerant, replicated queue system, interacting with the Apache Zookeeper distributed configuration service, fails to provide fault-tolerance because of the composition of the system

through a single, underspecified interface that does not account for network failures.

2.1 Background

The Apache Zookeeper system is a fault-tolerant, distributed key-value store that is commonly used as a distributed configuration service. Zookeeper exposes this key-value store through a filesystem-like API. When used for cluster membership, each node in the cluster connects to Zookeeper and writes what is referred to an ephemeral node — an object that will be present as long as the node remains connected to Zookeeper. Thus, the set of ephemeral nodes designates a view of the current cluster membership — nodes that are connected to Zookeeper.

Apache Kafka is a fault-tolerant, replicated queue system, typically configured by Apache Zookeeper. Kafka uses membership information provided by Zookeeper for establishing the replica set used for writes against the queue. Kafka selects one of the nodes from the membership set as the leader and uses primary-backup replication by writing to the leader, waiting for the followers to acknowledge; before acknowledging the write to the user. Composition appears trivial: as soon as membership changes, Kafka's replication system is notified of the updated membership and incorporates the latest membership and leader for data storage of items in the queue.

For example, consider three Kafka nodes $\{A,B,C\}$ and a single Zookeeper node Z. In this example, a single network partition that isolates nodes $\{A,Z\}$ from $\{B,C\}$ causes the Zookeeper system to notify A that the cluster membership and leader are itself as it is no longer receiving heartbeats from nodes B and C. When node A accepts a write, a response is returned to the user after only storing the value at A, ensuring that a crash failure of A will result in data loss. In this example, it is important to note that both systems guaranteed local invariants: Zookeeper provides a view of all non-crashed nodes; Kafka does not acknowledge writes until writes are acknowledged by the entire replica set.

The data loss problem occurs because the composition of Zookeeper and Kafka does not encode a Kafka application invariant: for the system to remain fault-tolerant to f faults, the write should not be accepted unless the membership contains $f+1$ members under the primary-backup replication scheme. Therefore, Kafka should not accept membership updates where membership is composed of a single member if the system is to remain fault-tolerant to a single fault.

2.2 Contemporary Solutions

The example composition problem presented between Zookeeper and Kafka is a problem of underspecification on the side of the Kakfa API. Zookeeper provides a general interface, one that is used by Kafka for getting a list of non-failed nodes. However, Kafka requires that to remain tolerant to f failures, the membership set that reads from Zookeeper must contain $f+1$ members — an invariant that is implicit.

Today, one typical way that this problem might be addressed is via documentation. The Kafka documentation could provide information about precisely what requirements its membership interface needs to remain fault-tolerant. Then, the application developer could write a dynamic layer to interpose between Zookeeper and Kafka, thus ensuring that either (a.) the membership provided to Kafka is at least $f+1$ members, or (b.) in the case there is only a single replica in the membership set, coerce the set into returning the empty set, causing Kafka to refuse writes until $f+1$ members were available.

However, this solution is both ad hoc and error prone. For instance, if Kafka was to change its internal replication strategy, from primary-backup, where $f+1$ members are required for fault-tolerance, to state-machine replication, where $2f+1$ members are required for fault-tolerance, then the system would left again in a similar problem where acknowledged durable writes may be lost under a number of failures less than f. As a strategy for application composition, documentation is insufficient because it is not machine checked or generally specified, and consequently places the burden of ensuring that compositions are correct through either manual or semi-formal testing [1].

2.3 Solution: Subtype Polymorphism

In the field of programming languages, subtype (a form of inclusion) polymorphism is a technique to relate data types to other data types through a notion of substitutability. Substitutability is a property where code written to operate on the supertype can safely be substituted for any of the subtypes in the subtyping relationship. [10]

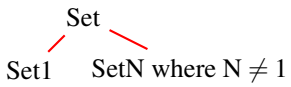

Figure 1: Hasse diagram depicting interface required by Zookeeper / Kafka interaction where fault-tolerance of 1 crash failure is required. In this case, the system should prohibit the use of Set1 statically.

To simplify our example, let us assume a desired fault-tolerance level of f and that membership is presented

by Zookeeper as a set containing a list of node identifiers. Using this, we can type the membership interface of Zookeeper as Set, representing the set of any possible cardinality. Kafka, is looking for Set0 or SetX where f is the desired fault tolerance level and $X = (f + 1)$. Kafka is written assuming a subtype of the membership, where Zookeeper provides only the supertype: this is the converse of substitutability, and does not hold. Therefore, using the membership provided by Zookeeper, without filtering out membership sets that would violate the invariant, can lead to data loss bugs. We depict the subtyping relation by the Hasse diagram in Figure 1.

3 Example: Sensor Readings

Amazon Greengrass is a service that allows programmers writing containerized Lambda functions to extend their processing to the edge, specifically by configuring and connecting devices designed for the Internet of Things (IoT) and allowing them to communicate with Lambda instances and databases running at Amazon Points of Presence or data centers.

To motivate the case for leveraging parameteric polymorphism, we extend our previous example with the addition of temperature sensors running at the edge that periodically report temperature of a room within the Greengrass infrastructure. Using Lambda, developers are able to write both a producer-driven application that places aggregates from individual sensor readings into a Kafka queue as well as a separate consumer-driven Lambda function to read samples from Kafka and trigger an alarm if a sensor reading is above or below a particular threshold. This example application is deceptively simple, and the Lambda function is typed as Float $\rightarrow$ Bool, the output being whether or not an alarm should be triggered.

When tested within a simulated sensor environment inside the data center, our application code may execute correctly; however, when operating at the edge with real devices, readings may arrive malformed due to an incorrectly operating or provisioned hardware sensor. In reality, the device might return an incompatible *NaN* (not a number), an integer if some sensor were configured to round, *ceil* or *floor* values, or the expected and specified floating point number. Furthermore, these *NaNs* may only appear at containers deployed out at the edge and never at the ones located within the data center. A programmer has been given "black box" objects for leveraging performance gains and lowering latency spikes due to proximity, but they have not been given an easy-to-use methodology to debug and handle issues around interoperability and correctness. How can they determine if something is *maybe* one kind of value or another?

In our trivial example, the type of sensor readings are overspecified: invalid sensor readings consumed by Kafka for processing with Lambda may lead to triggered decisions based on those sensor readings to fail because the consumer of the queue always expects to read values of type Float. Imagine this simple failure being expanded to more complex application chains of Lambda functions, ones involving sidecar proxies and load balancers, stream processors and data stores, or other queue-driven systems, e.g. Amazon Kinesis, all of which may occur downstream and rely upon the possibly malformed data Kafka is receiving. An interconnected system, made up of a myriad of container-based components, could become infected, causing false triggers and misinformation. *Chaos Engineering*, distributing tracing, and distributed replay debuggers have all attempted to solve this problem, but only do so after the fact — after the wrong information has already propagated throughout components.

3.1 Solution: Parametric Polymorphism

The *right* type of data arriving from the queue in our example is Option[Number], a parameterized type with variants that can be matched on Some[Float] or Some[Integer], which can be coerced into a Float, representing a valid and applicable temperature reading from a sensor, or the None type. The latter represents an invalid temperature reading that was either corrupted in transit (i.e. incorrectly deserialized) or the result of malfunctioning or wrongly provisioned sensors (i.e. returning a value of the wrong type, like String). Accordingly, our Lambda function should be typed Option[Number] $\rightarrow$ Option[Bool], from sensor input to the receiving triggered computation.

This propagation of the Option[A] type is pervasive in the chaining of Lambda functions where a Lambda is invoked in response to a particular event, e.g. queue insertion or object creation or modification, by passing the object into the Lambda invocation as a formal parameter. In Lambda chaining, users write a function $(A \rightarrow B) \rightarrow ()$ where an object of type A is transformed to an object of type B before being inserted into a queue or database for the next stage of processing, all which finally returns the type Unit. While this is what the programmer explicitly wants to encode, the execution model of Lambda makes no guarantees that the object passed as a formal parameter matches type A, therefore relying on the program to provide their own ad hoc implementation of Option types.

Parametric polymorphism allows developers to parameterize types based on a generic type and write application code that operates on these generic types. This enables the programmer to write code like Number $\rightarrow$ Bool and trivially extend this code to the generic Option type, yielding a function Option[Number] $\rightarrow$ Option[Bool].

This type has now been specialized to account for values being read from the queue that either were (a.) corrupted in transit or (b.) the result of an invalid sensor reading.

4 Towards a Type System

Solutions for standardizing a specification across a chain of containers would involve preserving invariants across sub-components while speaking a variety of APIs, each of which with its own semantics. What has been proposed and attempted thus far has mostly been ad hoc and error prone: interfaces are adapted to fit any way possible and mostly specified by implementation only.

To solve the problem we need a technique for specifying the types of the interfaces that are at the boundary between different interacting containers, and a technique for specifying the dependency graph between the interfaces describing their interactions with one another that allows us to analyze whether or not the interactions are well typed.

4.1 Interface Description Languages

A discussion of object-oriented distributed programming would be remiss if it failed to mention OMG's Common Object Request Broker Architecture (CORBA) system [14]. CORBA set out to enable transparent distributed programming across different architectures, different platforms, and different languages. As CORBA supported remove method invocations on distributed objects from different languages, objects would need to define their interfaces in an external format, referred to as an Interface Definition Language (IDL). This definition would assist in type conversion and mapping the objects between the source and destination languages.

CORBA, while successful, demonstrated that transparency was problematic because of the fundamental problems of distributed computing: latency and partial failure [7]. Techniques such as Java's Remote Method Invocation (RMI) proved more successful by specifically identifying only a subset of objects that would be accessed remotely, forcing the developer to think about distribution at these particular interaction points. Modern container-based distributed programming avoids these same pitfalls by also forcing the developer to explicitly provide interfaces that will be the boundary points between different containers. However, many approaches to date do not specify these interfaces precisely: most APIs today speak JSON over REST, a dictionary format that is defined dynamically at runtime and does not lend itself well to being verified statically.

Therefore, approaches such as the OpenAPI specification have attempted to formalize this by specifying typed APIs and implementations such as Swagger, then synthesize multi-language interfaces based on this specification. However, these interfaces remain dynamic at runtime. We believe there is no inherent challenge to mass adoption of a typed specification for APIs, as approaches like Swagger, Google's Protocol Buffers and Apache's Thrift have demonstrated. However, many of these systems remain primitive and do not support advanced type system feature, such as the polymorphic types that are discussed in this paper.

4.2 Soundness Across Boundaries

Inspired by Felleisen et al., [6] our approach is to specify the interfaces of container-based components and the boundaries between these components. These specifications are then checked via standard type checking procedure [13, 5, 12], or if necessary, using some sort of static solver. This "type-checker" for interfaces between container-based components either rejects problematic configurations, or just typechecks. For a program that typechecks, one of the following three situations are possible: (1) the configuration of container-based components is valid and requires no adaptation, (2) we generate a container that performs implicit conversion between types, or (3) we generate a container which performs contract enforcement.

It is necessary, but not always possible, to perform implicit conversions between component outputs and inputs in the case where outputs and inputs do not exactly match up. Its necessary to provide contract enforcement when we want to filter particular values of a type from being provided as an output for another container's input.

In the Kafka example, to remain tolerant to 1 crash failure, we must prevent Kafka from being notified of a replica set containing one member. With implicit conversion, we can convert the singleton set to an empty set, which prevents the system from violating safety by refusing writes when a single replica is available. With contract enforcement, we can prohibit Kafka from being notified of singleton set values for the replica set, causing Kafka to attempt to write to unavailable members of the cluster, also preserving safety, but performing additional, useless work. Choosing the correct solution is up to the programmer of the application and the components being used.

4.3 Further Challenges

Many difficulties still exist in the realization of the idea proposed above, including (a.) the provision of a general type system for container-based components and (b.) that is suitable for all open-source compositions available to developers today. We talk more specifically about these

issues below.

Typing. Ensuring that a unified type system is adopted by several different open source projects and consortia, maintained by a group of different developers is a big challenge to the adoption of this approach. Given that there is already movement towards standardizing interfaces between different projects in the open source ecosystem, we imagine that the challenge of encoding those interfaces formally as types is not insurmountable.

More recently, Burns and Oppenheimer [3] have argued for a technique inspired by the Simple Network Management Protocol (SNMP), where Abstract Syntax Notation One (ASN.1) is leveraged for specification of data structures and their serialization and transmission over the network. While this encoding of polymorphic types with ASN.1 has been previously explored in academia, it remains unclear if this is a viable or desirable approach in practice. [3, 9]

Subtyping relation. Ensuring that a subtyping relation is general enough to support all possible consumers of the interface seems as if it may be more challenging in systems that provide richer APIs, such as a system like Kafka which provides enqueue and dequeue APIs. That said, this approach can prove incredibly valuable in systems like Zookeeper, which are core infrastructure used by a large number of other systems.

To express more complex boundary interactions, the solution may be to adopt *dependent type theory* [11], which can describe a greater cardinality of enumerations and identify a correct subset of possible subtypes based on the value(s) associated with a type. A standard example in dependent type literature is expressing a function which returns the *first* element of a list [2], which would have the type of $\text{first} : \Pi \text{n} : \text{Nat}.\text{Vector}(\text{n}+1) \rightarrow data$. Described this way, the *first* function would cause a compilation failure whenever a developer attempted to apply it to an empty vector.

5 Conclusion

In building distributed applications today, containers are the *lingua franca*. However, the challenges in ensuring that applications built from the composition of containers remains largely a burden on the application developer. This is only further exacerbated by the fact that containerized applications are written using different programming languages and have underspecified requirements, but present interfaces that, to the developer, seem to be easily compatible with one another. In this work, we have proposed a technique inspired by the programming language community that we believe will ease the burden of building large-scale container based applications by providing a basis for ensuring compositions are correct.

References

[1] ALVARO, P., ROSEN, J., AND HELLERSTEIN, J. M. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 331–346.

[2] ASPINALL, D., AND HOFMANN, M. Dependent types. In *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[3] BURNS, B., AND OPPENHEIMER, D. Design patterns for container-based distributed systems. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (Denver, CO, 2016), USENIX Association.

[4] CARDELLI, L., AND WEGNER, P. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv. 17*, 4 (Dec. 1985), 471–523.

[5] CREMET, V., GARILLOT, F., LENGLET, S., AND ODERSKY, M. A core calculus for scala type checking. In *International Symposium on Mathematical Foundations of Computer Science* (2006), Springer, pp. 1–23.

[6] FELLEISEN, M., FINDLER, R. B., FLATT, M., KRISHNAMURTHI, S., BARZILAY, E., MCCARTHY, J., AND TOBIN-HOCHSTADT, S. A programmable programming language. *Commun. ACM 61*, 3 (Feb. 2018), 62–71.

[7] KENDALL, S. C., WALDO, J., WOLLRATH, A., AND WYANT, G. A note on distributed computing. Tech. rep., Mountain View, CA, USA, 1994.

[8] KYLE KINGSBURY. Call me maybe: Kafka. `https://aphyr.com/posts/293-call-me-maybe-kafka`. Accessed: 2018-03-19.

[9] LAVENDER, R. G., KAFURA, D. G., AND MULLINS, R. Programming with asn. 1 using polymorphic types and type specialization. In *ULPAA* (1994), pp. 151–166.

[10] LISKOV, B. H., AND WING, J. M. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16*, 6 (Nov. 1994), 1811–1841.

[11] MARTIN-LÖF, P. *Intuitionistic type theory*, vol. 1 of *Studies in Proof Theory. Lecture Notes*. Bibliopolis, Naples, 1984. Notes by Giovanni Sambin.

[12] MILNER, R. A theory of type polymorphism in programming. *Journal of computer and system sciences 17*, 3 (1978), 348–375.

[13] PIERCE, B. C., AND TURNER, D. N. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS) 22*, 1 (2000), 1–44.

[14] VINOSKI, S. Corba: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications magazine 35*, 2 (1997), 46–55.