Cheerios

Keith Simmons

Abstract

Computer scientists use formal verification to attempt to provide guarentees to the users of software about the run time characteristics of a given program. Verification techniques today often require a TCB or Trusted Code Base which due to time or effort constraints, the authors were not able to prove correct. When bugs occure in these pieces, they break down the strong guarentees of formal verification and devalue the extra time spent to complete it. In many cases, the verification of such a component could be reused in other projects, but not many mechanisms are in place to do so, especially when it comes to sharing libraries in coq beyond the standard library.

We present Cheerios, a formally verified library for serializing coq data types and writing serializers for custom types to be stored or sent over the wire. Cheerios uses modern software development practices to ensure it is easily readable and understandable. It also has a clear path for extending its standard library of serializable types with proof tactics which make proving the correctness of the new serializers relatively simple and provides the needed lemmas for users of Cheerios to immediately get up and running quickly with their project.

The verdi project has been critisized for not having a verified serializer and deserializer. [6, 5] To demonstrate Cheerios' utility we integrated it with the verdi project replacing the current usage of Ocaml's standard library marshalling functions with Cheerios and developed a vst for distributed systems in verdi which uses Cheerios to allow system creators to develop their own serializers instead of relying on the Ocaml marshalling library.

1 Introduction

Serialization is used to send or store complex data in the form of binary. Often, modern standard libraries will have serialization functionality built in [4, 3, 2], but do not have strict guarentees about the characteristics of the serialization. The OCaml serialization library, Marshal, currently has open bugs that could effect the runtime characteristics of formally verified software. [1] By reducing the size of the TCB for the verdi project, we improve the chances that verdi is bug free and reduce the surface area within which problems are able to occure. Other projects will also be able to drop Cheerios into their source and use it with relative ease.

Currently Cheerios does not serialize all of the base types in the coq standard library. This means that some basic serializers may be needed if a project uses the more obscure types. Cheerios does provide simple tools for writing those serializers though, so the burden should not be too high on the developer. Cheerios also only serializes to linked lists of booleans at this point. There are plans to allow all Cheerios serializers to be parameterized over their output types, but this is left for future work. Some forms of recursive data structures can be difficult to prove the serialize_deserialize_id property for. We would like to explore this particular case further so that we can provide better tactics or guidelines in the future. Cheerios also has no guarentees about how much memory or how large of a stream it will deserialize is. In the future we would like to provide layers on top of the existing type classes which do enforce these requirements so that one can be confident that Cheerios will not crash due to a stack overflow or similar issue.

The Cheerios project's TCB includes the Coq proof assistent and standard library. All other dependencies are verified in Coq.

Cheerios provides a clean interface for producing and using serializers in Coq. It describes a Serializer type class with serialize and deserialize functions, provides combinators for serializing complex data types, provides a deserialization monad which makes writing the deserializers simple and clean, and provides tactics for proving the serializer and deserializer functions correct for a given Serializer.

2 Overview

All Cheerios serializers are built with 4 components: A type to serialize and deserialize, a serialize function which takes an object of the above type and returns a list of booleans, a deserialize function which takes a list of booleans and returns an option of the above type and list

```
∀ a bin,
  deserialize ((serialize a) ++ bin)
    = Some (a, bin)
```

Figure 1: serialize_deserialize_id property

```
@serialize nat nat_Serializer 42
vs
serialize 42
```

Figure 2: By using type classes, the user does not have to specify which Serializer instance they are using. The compiler automatically locates the necessary parameters. This becomes especially useful when Serializers are parameterized over a base type as with combinators. The combinators can be chained without increasing the complexity of the calling code.

of booleans, and a proof that serializing is reversible.

2.1 Serializer Property

The reversibility property called serialize_deserialize_id_nil says that any serialized object, when deserialized yields the original object. This is a simplified version of the full serialize spec, shown in figure 1, which forces any binary added to the serialized object's list to be preserved after deserialization. By preserving binary not needed in the deserialization of an object, Cheerios allows users to send streams of data to the deserializer and just take off the bits needed in the deserialization of the first type.

One might wonder why we chose serialize reversibility instead of deserialize reversibility for our correctness property. When using a serializer, one requires that the object returned from disk or the network when deserialized is equal to the object serialized in the first place. If we had required binary be preserved, a user could create a serializer which serializes all data types to a single boolean which would satisfy the requirement. This way, no requirement is placed on the binary which is produced. You could concieve of a serializer being written which deserializes the same binary in two different ways depending on the type it is paramaterized over. This is completely allowed in the current version of Cheerios.

2.2 Type Class

Cheerios Serializers are instances of the Serializer type class. By using the type class functionality in Coq, a user does not need to specify the exact version of the serializer needed for serializing a given object. The type system will locate the correct instance if it has been defined or send an error message if it does not yet exist. This works great in practice because one can define serializers for all of the data types for a project in one place and then just call serialize and deserialize to utilize the serialization code. This does cause issues on occasion when insufficient type information is provided for the serializer. In such a case, one needs to directly anotate what they expect the deserializer to return or else it will fall back on defaults based on serializer instance ordering which is probably not desired.

```
Definition option_deserialize :
    deserializer (option A) :=
  b <- deserialize ;;
  match b with
   true => Some <$> deserialize
   false => ret None
  end.
```

Figure 3: Deserializer monad syntax removes unnecessary match statements created by the need for deserializers to fail.

Another benifit of using Coq type classes for serializers is that we get parameterization over types for free. The definition for serializing option types depends on the existance of a serializer for the base type, but is identical for any base type. Once a combinator is written, it will work for any serializer already defined. This reduces the need to write specialized serializers for a project if they use common data structures to represent their data.

2.3 Deserializer Monad

In order for serializers to be effective, there must be the ability for deserializing an arbitrary binary string to fail if formatted incorrectly. In such a case, instead of causing the program to crash we use an option return type. When building these combinators over many serialization types, it is not uncommon to have many layers of match statements in order to peel away the nested option objects. To ease the burden on the user, we developed a deserialization monad which removes the need for nested match statements. The arrow syntax indicates binding the value of deserializing to whatever is on the left or returns None if the deserialization fails. We provide fmap and sequence syntaxes to simplify applying functions to deserialized as shown in Figure 3.

2.4 serialize_deserialize_id_crush

We have also simplified the effort required to prove that a given serializer is reversible by providing the serialize_deserialize_id_crush tactic shown in Figure 4 which

```
Ltac deserializer_unfold :=
  unfold
    sequence,
    fmap,
    fail,
    put,
    get,
    bind,
    ret
  in *.

Ltac serialize_deserialize_id_crush :=
  intros; deserializer_unfold;
  repeat rewrite
    ?app_assoc_reverse,
    ?serialize_deserialize_id;
    auto
```

Figure 4: The proof tactic for automating the serialize_deserialize_id proof.

unfolds any deserializer monad functions present, and solves most proofs automatically.

We found that many serializers and combinators can be simplified by using more basic serializers to encode pieces of the new type. Lists are serialized by first encoding the length of the list and then the individual elements. Instead of encoding our own version of a number serializer, we were able to reuse the serializer for nats. Since the serialize_deserialize_id_crush tactic rewrites using the serialize_deserialize_id lemma, the proof goes through without any added complexity.

In some cases especially when serializers are recursive as we find in the list serializer, some care needs to be taken in the order you destruct and call serialize_deserialize_id_crush. Because of the brute force method of unfolding monad functions, it is possible to unfold too much and get stuck. Looking into improving this experience is future work.

3 Use Case

We developed a Verified System Transformer using Verdi's toolkit to allow systems written in Verdi to serialize the message and data types to list bools. After the future work of improving the extraction functionality for Cheerios is complete we will apply the VST to the existing verdi systems in order to remove the dependence on Marshal. During the development of the VST the vast majority of the engineering effort was expended working with Verdi's code. Little to no work was needed to integrate Cheerios beyond what all VSTs require. We were able to parameterize the VST over the Serializer types for the message and input/output types so that applying it to a given system will be as simple as writing a basic serializer for each of the types and completing the equality proof.

4 Conclusion

This paper presented Cheerios, a formally verified serialization library. Cheerios uses Coq's type system and type classes to make using it as simple as possible while still allowing extension of the supported types efficient. Cheerios provides a solution to using unverified software for serializing objects and can be dropped in to projects to reduce their TCB and prevent common serialization bugs.

Cheerios can serve as an example of how to take a small piece of a common trusted component in verified systems and break it into a reusable library that eases the difficulty of building more complex systems. As more libraries are created to solve similar problems, the gap between the time it takes to produce verified software and the time it takes to produce software with traditional methods will decrease allowing more people to benifit from it.

References

[1] Bugs in ocaml marshal. http://caml.inria.fr/mantis/view.php?id=7238. Accessed: 2016-6-10.

[2] Java objectinputstream. http://docs.oracle.com/javase/7/docs/api/java/io/ObjectInputStream.html. Accessed: 2016-6-10.

[3] .net xmlserializer. https://msdn.microsoft.com/en-us/library/system.xml.serialization.xmlserializer.aspx. Accessed: 2016-6-10.

[4] Ocaml marshaling. http://caml.inria.fr/pub/docs/manual-ocaml/libref/Marshal.html. Accessed: 2016-6-10.

[5] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.

[6] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi:

A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 357–368. ACM, 2015.