Abstract

Good tools can bring mechanical verification to programs written in mainstream functional languages. We use hs-to-coq to translate significant portions of Haskell's containers library into Coq, and verify it against specifications that we derive from a variety of sources including type class laws, the library's test suite, and interfaces from Coq's standard library. Our work shows that it is feasible to verify mature, widely-used, highly optimized, and unmodified Haskell code. We also learn more about the theory of weight-balanced trees, extend hs-to-coq to handle partiality, and – since we found no bugs – attest to the superb quality of well-tested functional code.

Ready, `Set`, Verify!
Applying `hs-to-coq` to real-world Haskell code

Joachim Breitner
DFINITY Stiftung, Zug, Switzerland
(*e-mail:* `joachim@dfinity.org`)

Antal Spector-Zabusky
University of Pennsylvania, Philadelphia, USA
(*e-mail:* `antals@cis.upenn.edu`)

Yao Li
University of Pennsylvania, Philadelphia, USA
(*e-mail:* `liyao@cis.upenn.edu`)

Christine Rizkallah
University of New South Wales, Sydney, Australia
(*e-mail:* `c.rizkallah@unsw.edu.au`)

John Wiegley
DFINITY Stiftung, Zug, Switzerland
(*e-mail:* `john@dfinity.org`)

Joshua Cohen
University of Pennsylvania, Philadelphia, USA
(*e-mail:* `joscoh@sas.upenn.edu`)

Stephanie Weirich
University of Pennsylvania, Philadelphia, USA
(*e-mail:* `sweirich@cis.upenn.edu`)

1 Introduction

What would it take to tempt functional programmers to verify their code?

Certainly, better tools would help. We see that functional programmers who use dependently-typed languages or proof assistants, such as Coq (The Coq development team, 2016), Agda (Bove *et al.*, 2009), Idris (Brady, 2017), and Isabelle (Nipkow *et al.*, 2002), do verify their code, since their tools allow it. However, adopting these platforms means rewriting everything from scratch. What about the verification of *existing* code, such as libraries written in mature languages like Haskell?

Haskell programmers can reach for LiquidHaskell (Vazou *et al.*, 2014) which smoothly integrates the expressive power of refinement types with Haskell, using SMT solvers for fully automatic verification. But some verification endeavors require the full strength of a mature interactive proof assistant like Coq. The `hs-to-coq` tool, developed by Spector-Zabusky *et al.* (2018), translates Haskell types, functions and type classes into equivalent

Coq code – a form of shallow embedding – which can be verified just like normal Coq definitions.

But can this approach be used for more than the small, textbook-sized examples it has been applied to so far? Yes, it can! In this work, we use `hs-to-coq` to translate and verify the two set data structures from Haskell's `containers` package.[1] This codebase is not a toy. It is decades old, highly tuned for performance, type-class polymorphic, and implemented in terms of low-level features like bit manipulation operators and raw pointer equality. It is also an integral part of the Haskell ecosystem. We make the following contributions:

- We demonstrate that `hs-to-coq` is suitable for the verification of unmodified, real-world Haskell libraries. By "real-world", we mean code that is up-to-date, in common use, and optimized for performance. In Section 2 we describe the `containers` library in more detail and discuss why it fits this description.
- We present a case study not just of verifying a popular Haskell library, but also of developing a good *specification* of that library. This process is worth consideration because it is not at all obvious what we mean when we say that we have "verified" a library. Section 4 discusses the techniques that we have used to construct a rich, two-sided specification; one that draws from diverse, cross-validated sources and yet is suitable for verification.
- We extend `hs-to-coq` and its associated standard libraries to support our verification goal. In particular, in Section 5 we describe the challenges that arise when translating the `Data.Set` and `Data.IntSet` modules, and our solutions. Notably, we drop the restriction in previous work (Spector-Zabusky *et al.*, 2018) that the input of the translation must be intrinsically total. Instead, we show how to safely defer reasoning about incomplete pattern matching and potential nontermination to later stages of the verification process.
- We increase confidence in the translation done of `hs-to-coq`. In one direction, properties of the Haskell test suite turn into Coq theorems that we prove. In the other direction, the translated code, when extracted back to Haskell, passes the original test suite.
- We provide new implementation-agnostic insight into the verification of the weight-balanced tree data structure, as we describe in Section 6. In particular, we find the right precondition for the central balancing operations needed to verify the particular variant used in `Data.Set`.

Our work provides a rich specification for Haskell's finite set libraries that is directly and mechanically connected to the current implementation. As a result, Haskell programmers can be assured that these libraries behave as expected. Of course, there is a limit to the assurances that we can provide through this sort of effort. We discuss the verification gap and other limitations of our approach in Section 7.

We would like to have been able to claim the contribution of finding bugs in `containers`, but there simply were none. Still, our efforts resulted in improvements to the `containers`

[1] Specifically, we target version 0.5.11.0, which was released on January 22, 2018 and was the most recent release of this library at the time of publication; it is available at `https://github.com/haskell/containers/tree/v0.5.11.0`.

library. First, an insight during the verification process led to an optimization that makes the `Data.Set.union` function 4% faster. Second, we discovered an incompleteness in the specification of the validity checker used in the test suite.

The tangible artifacts of this work have been incorporated into the `hs-to-coq` distribution and are available as open source tools and libraries.[2]

2 The `containers` library

We selected the `containers` library for our verification efforts because it is a critical component of the Haskell ecosystem. With over 4000 publicly available Haskell packages using `containers`, it is the third-most depended-on package of the Haskell package repository Hackage, after `base` and `bytestring`.[3]

The `containers` library is both mature and highly optimized. It has existed for over a decade and has undergone many significant revisions in order to improve its performance. It contains seven container data structures, covering support for finite sets (`Data.Set` and `Data.IntSet`), finite maps (`Data.Map` and `Data.IntMap`), sequences (`Data.Sequence`), graphs (`Data.Graph`), and trees (`Data.Tree`). However most users of the `containers` library only use the map and set modules;[4] moreover, the map modules are essentially analogues of the set modules. Therefore, we focus on the verification of `Data.Set` and `Data.IntSet` in the majority of this work. Furthermore, we used our experience verifying `Data.Set` to also verify `Data.Map.Strict`, as we describe in Section 5.9.

2.1 Weight-balanced trees

The `Data.Set` module implements finite sets using weight-balanced binary search trees. The definition of the `Set` datatype in this module, along with its membership function, is given in Figure 1.[5]

These sets and operations are polymorphic over the element type and require only that this type is linearly ordered, as expressed by the `Ord` constraint on the `member` function. The `member` function descends the ordered search tree to determine whether it contains a particular element.

The `Size` component stored with the `Bin` constructor is used by the operations in the library to ensure that the tree stays balanced. The implementation maintains the balancing invariant

$$s_1 + s_2 \leq 1 \vee (s_1 \leq 3 s_2 \wedge s_2 \leq 3 s_1),$$

[2] The full repository for `hs-to-coq` is at `https://github.com/antalsz/hs-to-coq`; for the version connected to this work, see the `JFP-containers` tag, which can be accessed at `https://github.com/antalsz/hs-to-coq/tree/JFP-containers`. The examples/containers directory contains the verification of `containers`; consult examples/containers/README.md for a guide to the relevant portions of the code.

[3] `http://packdeps.haskellers.com/reverse`

[4] We calculated that 78% of the packages on Hackage that depend on `containers` use only sets and maps.

[5] All code listings in this paper are manually reformatted and may omit module names from fully qualified names.

```
------------------------------------------------------------------------
-- Sets are size balanced trees
------------------------------------------------------------------------
data Set a = Bin {-# UNPACK #-} !Size !a !(Set a) !(Set a)
           | Tip

type Size = Int

-- | O(log n). Is the element in the set?
member :: Ord a => a -> Set a -> Bool
member = go
  where go !_ Tip = False
        go x (Bin _ y l r) = case compare x y of
                                LT -> go x l
                                GT -> go x r
                                EQ -> True
```

Fig. 1. The `Set` data type and its membership function, from http://hackage.haskell.org/package/containers-0.5.11.0/docs/src/Data.Set.Internal.html#Set.

where s_1 and s_2 are the sizes of the left and right subtrees of a `Bin` constructor. This definition is based on the description by Adams (1992), who modified the original weight-balanced tree proposed by Nievergelt & Reingold (1972). Thanks to this balancing, operations such as insertion, membership testing, and deletion take time logarithmic in the size of the tree.

This type definition has been tweaked to improve the performance of the library. The ! annotations indicate that all components should be strictly evaluated; whenever a value constructed by `Bin` is evaluated, so are all of its strictly-evaluated fields. Because the `Size` field is strict, it can be unpacked; this is a size and speed optimization that stores the machine word directly with the constructor, rather than storing a pointer to a boxed heap representation thereof.

2.2 Big-endian Patricia trees

The `Data.IntSet` module also provides search trees, but these are specialized to values of type `Int` to provide more efficient operations, especially `union`. This implementation is based on big-endian Patricia trees, as proposed in Morrison's work on PATRICIA (1968) and described in a pure functional setting by Okasaki & Gill (1998).

The definition of this data structure is shown in Figure 2. The core idea is to use the bits of the stored values to decide in which subtree of a node they should be placed. In a node `Bin p m s1 s2`, the mask `m` has exactly one bit set. All bits higher than the mask bit are equal in all elements of that node; they form the prefix `p`. The mask bit is the highest bit that is not shared by all elements. In particular, all elements in `s1` have this bit cleared, while all elements in `s2` have it set. When looking up a value `x`, the mask bit of `x` tells us into which branch to descend.

```
data IntSet = Bin {-# UNPACK #-} !Prefix {-# UNPACK #-} !Mask !IntSet !IntSet
--Invariant: Nil is never found as a child of Bin.
--Invariant: The Mask is a power of 2. It is the largest bit position at
--           which two elements of the set differ.
--Invariant: Prefix is the common high-order bits that all elements share to
--           the left of the Mask bit.
--Invariant: In Bin prefix mask left right, left consists of the elements
--           that don't have the mask bit set; right is all the elements
--           that do.
            | Tip {-# UNPACK #-} !Prefix {-# UNPACK #-} !BitMap
--Invariant: The Prefix is zero for the last 5 (on 32 bit arches) or 6 bits
--           (on 64 bit arches). The values of the set represented by a tip
--           are the prefix plus the indices of the set bits in the bit map.
            | Nil

--A number stored in a set is stored as
--* Prefix (all but last 5-6 bits) and
--* BitMap (last 5-6 bits stored as a bitmask)
--  Last 5-6 bits are called a Suffix.
type Prefix = Int
type Mask   = Int
type BitMap = Word
type Key    = Int

--| O(min(n,W)). Is the value a member of the set?
member :: Key -> IntSet -> Bool
member !x = go
  where go (Bin p m l r) | nomatch x p m = False
                         | zero x m      = go l
                         | otherwise     = go r
        go (Tip y bm) = prefixOf x == y && bitmapOf x .&. bm /= 0
        go Nil = False
```

Fig. 2. The `IntSet` data type and its membership function, from http://hackage.haskell.org/package/containers-0.5.11.0/docs/src/Data.IntSet.Internal.html#IntSet.

Instead of storing a single value at the leaf of the tree, the implementation in `containers` improves time and space performance by storing the membership information of consecutive numbers as the bits of a machine-word-sized bitmap in the `Tip` constructor.[6]

The `Nil` constructor is the only way to represent an empty tree, and will never occur as the child of a `Bin` constructor. Every well-formed `IntSet` is either made of `Bin`s and `Tip`s, or a single `Nil`.

2.3 A history of performance tuning

The history of the `Data.Set` module can be traced back to 2004, when a number of competing search tree implementations were debated in the “Tree Wars” thread on the Haskell libraries mailing list. Benchmarks showed that Daan Leijen’s implementation had the best performance, and it was added to `containers` in 2005 as `Data.Set`.[7]

[6] Although this feature was contributed in 2011 by the first author, he certainly did this without having an eventual formal verification in mind.

[7] https://github.com/haskell/containers/commit/bbbba97c

In 2010, Milan Straka thoroughly evaluated the performance of the `containers` library and implemented a number of performance tweaks (Straka, 2010). For example:

> *When balancing a node, the function `balance` checked the balancing condition and called one of the four rotating functions, which rebuilt the tree using smart constructors. This resulted in a repeated pattern matching, which was unnecessary. We rewrote the `balance` function to contain all the logic and to use as few pattern matches as possible. That resulted in significant performance improvements in all `Set` methods that modify a given set.*

This change[8] replaced a fairly readable `balance` and several small and descriptive helper functions with a single dense block of code. A later change[9] by Straka created two copies of this dense, complicated `balance` function, each specialized and optimized for different preconditions.

Adams (1992) describes two algorithms for `union`, `intersection`, and `difference`: "hedge union" and "divide and conquer". Originally `containers` used the former, but in 2016 its maintainers switched to the latter,[10] again based on performance measurement.

The module `Data.IntSet` (and `Data.IntMap`) has been around even longer. Okasaki & Gill mention in their 1998 paper (Okasaki & Gill, 1998) that GHC had already made use of `IntSet` and `IntMap` for several years. In 2011, the `Data.IntSet` module was rewritten to use machine-words as bit maps in the leaves of the tree, as discussed at the end of Section 2.2.[11] This moved the `containers` library further away from the literature on Patricia trees and introduced a fair amount of low-level bit twiddling operations (e.g., `highestBitMask`, `lowestBitMask`, and `revNat`).

2.4 The test suite of `containers`

The first tests were added to `containers` in 2007,[12] in the form of a few regression tests for observed bugs. Three years later, Don Stewart added a comprehensive test suite using QuickCheck (Claessen & Hughes, 2000) with 91% code coverage, and reported that "[n]o bugs were found".[13] This test suite helped to maintain a consistently high quality and very few bugs crept into released versions of the library. In fact, the only serious bug mentioned in the library's changelog – a completely broken implementation of `Data.IntMap.restrictKeys` – only occurred because the tests for `restrictKeys` were accidentally not run as part of the test suite.[14]

[8] https://github.com/haskell/containers/commit/3535fcbe
[9] https://github.com/haskell/containers/commit/d17d7182
[10] https://github.com/haskell/containers/commit/c3083cfc
[11] https://github.com/haskell/containers/pull/3
[12] https://github.com/haskell/containers/commit/9d6c49b5
[13] https://github.com/haskell/containers/commit/38743e39
[14] https://github.com/haskell/containers/issues/392

3 Overview of our verification approach

In order to verify Set, IntSet, and Map, we use hs-to-coq to translate the unmodified Haskell modules to Gallina and then use Coq to verify the translated code. For example, consider the excerpt of the implementation of Set in Figure 1. The hs-to-coq tool translates this input to the following Coq definitions.[15] The type name Set is renamed to Set_ to avoid clashing with the Coq keyword.

```
Definition Size := GHC.Num.Int%type.

Inductive Set_ a : Type
  := Bin : Size -> a -> (Set_ a) -> (Set_ a) -> Set_ a
  |  Tip : Set_ a.

Definition member {a} `{GHC.Base.Ord a} : a -> Set_ a -> bool :=
  let fix go arg_0__ arg_1__
    := match arg_0__, arg_1__ with
         | _, Tip => false
         | x, Bin _ y l r => match GHC.Base.compare x y with
                               | Lt => go x l
                               | Gt => go x r
                               | Eq => true
                             end
       end
  in go.
```

The unpacking and strictness annotations were ignored, as they do not make sense in Coq: unpacking only controls memory layout, and not the semantics of the type; and strictness annotations control evaluation order, which is immaterial in a total language. While in Haskell the strictness annotations further rule out the use of infinite data structures, hs-to-coq already assumes that all data types are inductive (Spector-Zabusky *et al.*, 2018), so they add nothing.

These definitions also depend on hs-to-coq's pre-existing translated version of GHC's standard library base. Here, we use the existing translation of Haskell's Int type, the Ord type class, and Ord's compare method.

We carry out this translation for the Set, IntSet, and Map along with their attendant functions, and then verify the resulting Gallina code. In Section 4 we discuss the properties that we prove about these data structures, focusing on the details of the two set data structures. Naturally, this translation process means that these proofs comes with an attendant formalization gap, which we discuss in Section 7.1.

To further test the translation from Haskell to Coq, we also used Coq's extraction mechanism to translate the generated Gallina code, like that seen above, *back* to Haskell. This process converts the implicitly-passed type class dictionaries to ordinary explicitly-passed function arguments, but otherwise preserves the structure of the code. For example, the member function above is extracted back to Haskell as the following code:

```
member :: (Base.Eq_ a1) -> (Base.Ord a1) -> a1 -> (Set_ a1) -> Prelude.Bool
```

[15] In the file examples/containers/lib/Data/Set/Internal.v.

```
member h h0 arg_0__ arg_1__ =
  case arg_1__ of {
    Bin _ y l r ->
      case Base.compare h h0 arg_0__ y of {
        Prelude.EQ -> Prelude.True;
        Prelude.LT -> member h h0 arg_0__ l;
        Prelude.GT -> member h h0 arg_0__ r};
    Tip -> Prelude.False}
```

By providing an interface that restores the type-class based types, we can run the original `containers` test suite against this code. This process helps us check that `hs-to-coq` preserves the semantics of the original Haskell program during the translation process. Our handwritten shim module imports the extracted definitions above and then reexports these definitions with their original types by using Haskell's type class resolution to provide the explicit dictionaries.

```
--Create dictionary from an Eq type class constraint
eq_a :: Eq a => Base.Eq_ a
eq_a _ f = f (Base.Eq___Dict_Build (==) (/=))

--Create dictionary from an Ord type class constraint
ord_a :: Prelude.Ord a => Base.Ord a
ord_a _ = Base.ord_default Prelude.compare eq_a

--Restore the original type of the extracted member function
--by providing explicit dictionaries
member :: Prelude.Ord a => a -> Set a -> Bool
member = S2.member eq_a ord_a
```

Figure 3 provides an overview of the Haskell code that we target, the Gallina code that we translate it into, and the Coq proofs that we wrote. For ease of comparison (`Map` is quite different and much larger), the bar chart focuses on the two different set modules, but our numbers include `Map`. The relevant modules of the `containers` library contain 325 functions and 41 type class instances, written in 4096 lines of code (excluding comments and blank lines). Out of these, 36 functions and 20 type class instances (509 loc) were deemed "out of scope" and not translated. (We discuss untranslated definitions in more detail in Sections 5.3 and 7.2.) Our translation produces 4784 lines of Gallina code.

The `Set`, `IntSet`, and `Map` data structures come with extensive APIs. We specify and verify a representative subset of commonly used functions (listed in Figure 4), covering 68% of the `Set` API, 49% of the `IntSet` API, and 43% of the `Map` API. This verified API is complete enough to instantiate Coq's specification of finite sets and maps, along with many other specifications at varying levels of abstraction; for more detail, see Section 4.

As Coq is not an automated theorem prover, verification of these complex data structures requires significant effort. In total, we verified 2291 lines of Haskell; the verification of `Set`, `IntSet`, and `Map` required 9.0 lines of proof per lines of code. This factor is noticeably higher for `IntSet` (9.9×) than for `Set` (7.4×), as the latter is conceptually simpler to reason about and allowed us to achieve a higher degree of automation using Coq tactics.

Our proofs also require the formalization of several background theories (not counted in the proof-to-code ratio above), including: integer arithmetic and bits (1265 loc): lists and

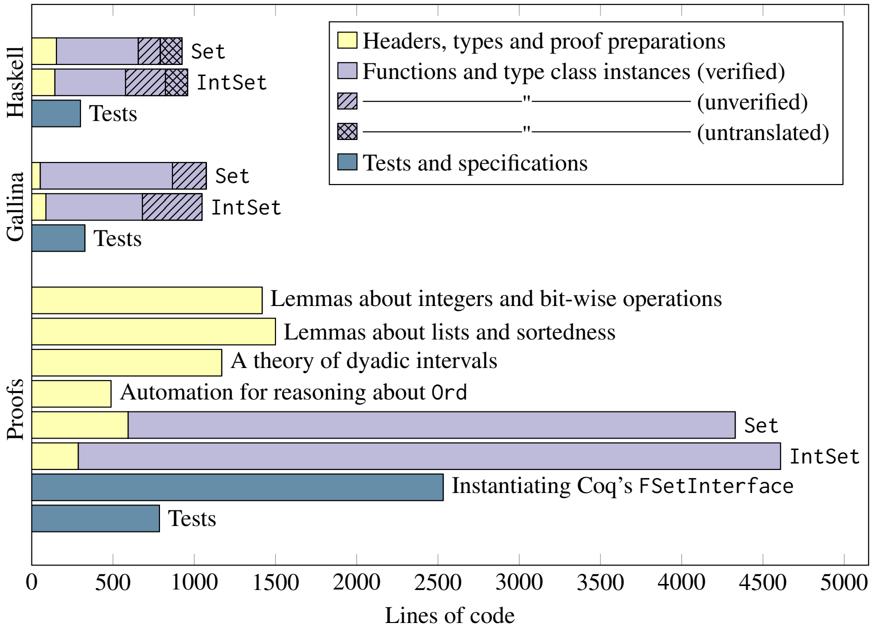

Fig. 3. A quantitative overview of the Haskell code, its translation into Coq and our proofs

sortedness (1500 loc); dyadic intervals, which are used for verifying `IntSet` (1169 loc); and support for working with lawful `Ord` instances (488 loc), including a complete decision procedure,[16] a port of Coq's `Coq.Structures.OrdersTac` to a setting using type classes instead of modules.

4 Specifying `Set` and `IntSet`

The phrase "we have verified this piece of software" on its own is meaningless: the particular specification that a piece of software is verified against matters. Good specifications are *rich*, *two-sided*, *formal*, and *live* (Appel *et al.*, 2017). A specification is *rich* if it "describ[es] complex behaviors in detail". It is *two-sided* if it is "connected to both implementations and clients". It is *formal* if it is "written in a mathematical notation with clear semantics". And it is *live* if it is "connected via machine-checkable proofs to the implementation".

All specifications of `Set` and `IntSet` that we use are formal and live by definition. They are formal because we express the desired properties using Gallina, the language of the Coq proof assistant; and they are live because we use `hs-to-coq` to automatically convert `containers` to Coq where we develop and check our proofs.[17] But how can we ensure that our specifications of `Set` and `IntSet` are two-sided and rich? How do we know that the

[16] In the file examples/containers/theories/OrdTactic.v.

[17] "Liveness" here solely refers to the fact that the specification and the Haskell source are automatically connected; it does not guarantee that the semantics of the two are identical. We discuss the formalization gap further in Section 7.1.

Set: API: delete, deleteMax, deleteMin, difference, disjoint, drop, elems, empty, filter, foldl, foldl', foldr, foldr', fromAscList, fromDescList, fromDistinctAscList, fromDistinctDescList, fromList, insert, intersection, isSubsetOf, lookupMax, lookupMin, map, mapMonotonic, maxView, member, minView, notMember, null, partition, singleton, size, split, splitAt, splitMember, take, toAscList, toDescList, toList, union, unions
Instances: Eq, Eq1, Monoid, Ord, Ord1, Semigroup
Internal functions: balanceL, balanceR, combineEq, glue, insertMax, insertMin, insertR, link, maxViewSure, merge, minViewSure, valid

IntSet: API: delete, difference, disjoint, elems, empty, filter, foldl, foldr, fromList, insert, intersection, isProperSubsetOf, isSubsetOf, map, member, notMember, null, partition, singleton, size, split, splitMember, toAscList, toDescList, toList, union, unions
Instances: Eq, Monoid, Ord, Semigroup
Internal functions: branchMask, equal, highestBitMask, mask, nequal, nomatch, revNat, shorter, valid, zero

Map: API: adjust, alter, assocs, balanceL, balanceR, delete, deleteMax, deleteMin, difference, drop, elems, empty, filter, findIndex, findWithDefault, foldl, foldl', foldlWithKey', foldr, foldr', foldrWithKey, fromAscList, fromDescList, fromDistinctAscList, fromDistinctDescList, fromList, fromSet, glue, insert, insertLookupWithKey, insertMax, insertWith, insertWithKey, intersection, keys, keysSet, link, link2, lookup, lookupLE, lookupLT, lookupMax, lookupMin, mapEither, mapMaybe, mapMaybeWithKey, mapWithKey, maxView, member, minView, notMember, null, partition, restrictKeys, singleton, size, split, splitAt, splitLookup, splitRoot, take, toAscList, toDescList, toList, union, unionWithKey, unions, unionsWith, update, updateLookupWithKey, updateWithKey, withoutKeys
Instances: Eq, Eq1, Monoid, Semigroup
Internal functions: insertMin, insertR, insertWithKeyR, insertWithR, isProperSubmapOfBy, maxViewSure, minViewSure, splitMember

Fig. 4. The verified subset of functions and type classes in Data.Set, Data.IntSet, and Data.Map.Strict

specifications are not just facts that happen to be true, but are useful for the verification of larger systems? What complex behaviors of the data structures should we specify?

To ensure that our specifications are two-sided, we use specifications that we did not invent ourselves. Instead, we draw our specifications from a variety of diverse sources: from several parts of the containers codebase (Sections 4.1 to 4.4), from Haskell type class laws (Section 4.5), from pre-existing Coq theories (Section 4.6), and from a mathematical description of sets (Section 4.7). This way, we also ensure that our specifications are rich because they describe the complex Set and IntSet data structures at varying levels of abstraction. Finally, by verifying the code against these disparate specifications, we not only increase the assurance that we captured all the important behaviors of Set and IntSet, but we also cross-validate the specifications against each other.

While we focus here on Set and IntSet, these techniques are all more general – in Section 5.9, we discuss how we were able to take the techniques presented here and extend them directly to verify the related Map data structure.

```
Inductive Bounded : Set_ e -> option e -> option e -> Prop :=
 | BoundedTip : forall lb ub,
   Bounded Tip lb ub
 | BoundedBin : forall lb ub s1 s2 x sz,
   Bounded s1 lb (Some x) ->
   Bounded s2 (Some x) ub ->
   isLB lb x = true -> (* If lb is defined, it is less than x *)
   isUB ub x = true -> (* If ub is defined, it is greater than x *)
   sz = (1 + size s1 + size s2) ->
   balance_prop (size s1) (size s2) -> (* weights of tree are balanced *)
   Bounded (Bin sz x s1 s2) lb ub.

(** Any set that has bounds is well-formed *)
Definition WF (s : Set_ e) : Prop := Bounded s None None.
```

Fig. 5. Well-formed weight-balanced sets

4.1 Specifying implementation invariants

`Set` and `IntSet` are two examples of abstract types whose correctness depend on invariants. Therefore, we define well-formedness predicates `WF: Set_ e -> Prop` and `WF : IntSet -> Prop` and show that the operations preserve these properties. The definition of well-formedness differs between the two types, but specifications of both are already present within `containers`.

Well-formed weight-balanced trees Our definition of `WF` for `Set` is derived from the `valid` function defined in the `containers` library. This function checks whether the input (1) is a balanced tree, (2) is an ordered tree, and (3) has the correct values in its size fields. It is not part of the normal, user-facing API of `containers` (since all exported functions preserve well-formedness), but is used internally by the developers for debugging and testing. For us, it is valuable as an executable specification, with less room for ambiguity and interpretation than comments and documentation.

However, rather than using `valid` directly, we define well-formedness as an inductive predicate, because we find it more useful from a proof engineering perspective. In particular, our definition of `WF`, as shown in Figure 5, relies on the `Bounded` inductive family. Its indices express lower and upper bounds of the elements stored in the tree; `None` means unbounded. At the same time, the property also checks that the sizes of the two subtrees are balanced in the `Bin` case.[18] Nevertheless, we can relate the `WF` predicate to the `valid` function found in `containers`:

```
Lemma Bounded_iff_valid : forall s, WF s <-> valid s = true.
```

Well-formed Patricia trees Our well-formedness definition for `IntSet` is derived from the comments in the `IntSet` data type, shown in Figure 2, where the type declaration almost disappears beneath a large swath of comments describing its invariants.

In this case, the documentation-derived well-formedness predicate is stronger than the corresponding `valid` function from the implementation – the Haskell function was missing

[18] In the file examples/containers/theories/SetProofs.v.

some necessary conditions. We reported this to the library authors,[19] who have since fixed valid.

This fix to valid is an example of how verification allows us to cross-validate specifications and ensure that the invariants written in the comments are adequately reflected by the code. On the other hand, sometimes we discover that it is the comments in the code that are incomplete. For example, the comment describing the precondition for balanceL in the Data.Set module was misleading and too vague; for more detail, see Section 6.1.

4.2 Property-based testing

At the next level of verification, we would like to show that the implementations of Set and IntSet are correct according to the implementors of the module. We specify correctness by deriving a definition directly from the test suite that is distributed with the containers library.

Thanks to the popularity of property-based testing within the Haskell community, this test suite contains a wealth of precisely specified general properties expressed using QuickCheck (Claessen & Hughes, 2000). For example, one such property states that the union operation is associative, in terms of the Haskell-level equality function ==:

```
prop_UnionAssoc :: IntSet -> IntSet -> IntSet -> Bool
prop_UnionAssoc t1 t2 t3 = union t1 (union t2 t3) == union (union t1 t2) t3
```

which we can interpret as a theorem about union:[20]

```
Theorem thm_UnionAssoc:
  forall t1, WF t1 -> forall t2, WF t2 -> forall t3, WF t3 ->
  union t1 (union t2 t3) == union (union t1 t2) t3 = true.
```

We do not have to write these theorems by hand: as we describe in Section 5.8, we use hs-to-coq in a nonstandard way to automatically turn these executable tests into Gallina propositions (i.e., types). We have translated IntSet's test suite in this manner and have used our other specifications to prove that all QuickCheck properties about verified IntSet functions are theorems (with one exception due to our choice of integer representation – see Section 5.5).

4.3 Numeric overflow in Set

There is one way in which we have diverged from the specification of correctness given by the comments of the containers library. The Data.Set module states:[21]

> *Warning: The size of the set must not exceed maxBound::Int. Violation of this condition is not detected and if the size limit is exceeded, its behavior is undefined.*

[19] https://github.com/haskell/containers/issues/522
[20] In the file examples/containers/theories/IntSetPropertyProofs.v.
[21] http://hackage.haskell.org/package/containers-0.5.11.0/docs/Data-Set.html

In practice, it makes no difference whether `Int` is bounded or not, as a valid set with $(2^{63}-1)$ elements would require at least 368 exabytes of storage. What does this imply for our specification of `Set`? Should we use fixed-width integers to represent `Int`? This choice would closely match the implementation, but we would have to carefully add preconditions to all our lemmas to avoid integer overflow, greatly complicating the proofs, with little verification insight to be gained. Furthermore, such a specification would be difficult to use by clients, who themselves would need to prove that they satisfy such preconditions.

Instead, we translate Haskell's `Int` type to Coq's type of unbounded integers (called `Z`). This mapping avoids the problem of integer overflow altogether and is arguably consistent with the comment, as this choice replaces undefined behavior with concrete behavior. (The situation is slightly different for `IntSet`; see Section 5.5.)

4.4 Rewrite rules

So far, we have only been concerned with specifying the correctness of the data structures using the definition of correctness that is present in the original source code; the comments, the `valid` functions and the QuickCheck properties. However, to be able to claim that our specifications are two-sided, we need to show that the properties that we prove are useful to clients of the module.

One source of such properties are the *rewrite rules* (Peyton Jones *et al.*, 2001) that enable library-specific compiler optimization. The `containers` library includes a small number of such rules. These annotations instruct the compiler to replace any occurrence of the pattern on the left-hand side in the rule by the expression on the right hand side. The standard example is

```
{-# RULES "map/map" forall f g xs. map f (map g xs) = map (f . g) xs #-}
```

which fuses two adjacent calls to `map` into one, eliminating the intermediate list.

The program transformation *list fusion* (Peyton Jones *et al.*, 2001) is implemented completely in terms of rewrite rules, and the rules in the `containers` library set up its functions for fusion; an example is

```
{-# RULES "Set.toAscList"
          forall s. toAscList s = build (\c n -> foldrFB c n s) #-}
```

which transforms the `toAscList` function into an equivalent representation in terms of `build`.

We can view these rewrite rules as a direct specification of properties that the compiler assumes are true during compilation. Rewrite rules are used by GHC during optimization; if any of these properties are actually false, GHC will silently produce incorrect code. Therefore, any proof about the correctness of GHC's compilation of these files depends on a proof of these properties. We have manually translated all the rules into Coq – there are only few of them, so manual translation is viable – and have proved that the translated operations satisfy this specification.[22]

[22] In the files examples/containers/theories/SetProofs.v and examples/containers/theories/IntSetProofs.v.

```
Class OrdLaws (t : Type) {HEq : Eq_ t} {HOrd : Ord t} {HEqLaw : EqLaws t} := {
  (* The axioms *)
  Ord_antisym    : forall a b,   a <= b = true -> b <= a = true -> a == b = true;
  Ord_trans_le   : forall a b c, a <= b = true -> b <= c = true -> a <= c = true;
  Ord_total      : forall a b,   a <= b = true \/ b <= a = true;
  (* The other operations, in terms of <= or == *)
  Ord_compare_Lt : forall a b,   compare a b = Lt <-> b <= a = false;
  Ord_compare_Eq : forall a b,   compare a b = Eq <-> a == b = true;
  Ord_compare_Gt : forall a b,   compare a b = Gt <-> a <= b = false;
  Ord_lt_le      : forall a b,   a  <  b = negb (b <= a);
  Ord_ge_le      : forall a b,   a >= b =      (b <= a);
  Ord_gt_le      : forall a b,   a  >  b = negb (a <= b)}.
```

Fig. 6. Our codification of the `Ord` type class laws

4.5 Type classes with laws

Many Haskell type classes come with *laws* that all instances of the type class should satisfy, which provides another source of external specification that we can use. For example, an instance of `Eq` is expected to implement an equivalence relation, an instance of `Ord` should describe a linear order, and an instance of `Monoid` should be, well, a monoid.

We reflect these laws using type classes whose members are the required properties. For example, we have defined `EqLaws`, `OrdLaws` (shown in Figure 6), and `MonoidLaws`. These classes can only be instantiated if the corresponding instance is lawful.

Even though we have defined these laws ourselves, using our understanding of what they mean for the Haskell standard library, we argue that they form a two-sided specification. In particular, we have been clients to the `Ord` laws in our verification of the `Set` data structure. Almost all theorems about `Set` must constrain the element type to one that is an instance of `OrdLaws`. Therefore, we know our specification of these laws is sufficiently strong to verify this library.

At the same time, we have also shown that multiple type class instances satisfy these laws including both set modules[23] and other types such as `Z`, `unit`, tuples, `option`, and `list`.[24] Because we successfully instantiated these type classes for many types, we also know that they are not too strong.

That said, nailing down the precise form of the type class laws can be tricky. Consider the case of a `Monoid` instance for a type `T`. The associativity law can be stated as "for all elements `x`, `y` and `z` of `T`, we have that `x <> (y <> z)` is equal to `(x <> y) <> z`," where `<>` is the monoid operation (i.e. `mappend` in Haskell). But in order to write this down as part of `MonoidLaws` in Coq, we need to make two decisions:

1. What do we mean by "equal"? The first option is to use Coq's propositional equality and require that `x <> (y <> z) = (x <> y) <> z`. This would, however, prevent us from making `Set_`, with `(<>) = union`, a member of `MonoidLaws`: two extensionally-equal sets may be represented by differently structured trees. Therefore, we instead require that the two expressions are equal according to their `Eq` instance:`(x <> (y <> z) == (x <> y) <> z) = true`. The tradeoff with this approach

[23] In the files examples/containers/theories/SetProofs.v and examples/containers/theories/IntSetProofs.v.

[24] In the file base-thy/GHC/Base.v.

is that it precludes instances like `MonoidLaws b -> MonoidLaws (a -> b)`, since functions have no instance of `Eq` (and indeed, in general cannot have decidable equality).

For many types, however, this distinction is moot, since Haskell's equality coincides with structural equality; for example, this is the case for `Bool`, for `Integer`, and for `IntSet` itself (although not for `Set`, as mentioned above). To facilitate reasoning about such types, we provide the type class `EqExact`, which states that

```
forall {a} `{EqExact a}, x == y = true <-> x = y
```

2. What do we mean by "For all elements of `T`"? The obvious choice is universal quantification over all elements of `T`:

```
forall (x y z : T), x <> (y <> z) == (x <> y) <> z = true
```

But again, this collides with common practice in Haskell. Once again, consider `Set`: `union` only works correctly on well-formed sets. Therefore, our approach is to define an instance of this and other classes not for the type `Set_ e`, but for the type of well-formed sets, `{s : Set_ e | WF s}`, where type class laws hold universally. This instance reflects the "external view" of the data structure – clients should only have access to well-formed sets.

An alternative could be to instead constrain `MonoidLaws`'s theorems to hold only on members of `T` that are well-formed in some general way (e.g., according to an `IsWF` type class that could be instantiated at different types). In this way, we could instantiate `MonoidLaws` directly with types that require well-formedness, without the need for subset types.

4.6 Specifications from the Coq standard library

Because we are working in Coq, we have access to a standard library of specifications for finite sets, which we know are two-sided because they have already appeared in larger Coq developments. The `Coq.FSets.FSetInterface` module[25] provides module types that cover many common operations and their properties. The module types come in two varieties: one that specifies sets of elements that merely have decidable equality (`WSfun`, `WS`), and one that specifies sets of elements that can be linearly ordered (`Sfun`, `S`). The `WSfun` and `Sfun` modules are presented as module functors that take an `OrderedType` module, containing the linearly-ordered element type, as an input; the `WS` and `S` modules are the same, but they inline this information.

For example, the parameterized module type `WSfun` provides one specification of a finite set type, called `t` in the excerpt from this interface below. The element type of this set, `E.t`, is required to have decidable equality.

```
Module Type WSfun (E : DecidableType).
  Definition elt := E.t.
  Parameter t : Type.            (* Set type *)
```

[25] https://coq.inria.fr/library/Coq.FSets.FSetInterface.html

```
  Parameter In : elt -> t -> Prop.  (* Characteristic function
                                       for a Set *)

  Parameter mem : elt -> t -> bool. (* Membership function *)

  (* Specification of mem *)
  Parameter mem_1 : forall x s, In x s -> mem x s = true.
  Parameter mem_2 : forall x s, mem x s = true -> In x s.

  ...
End WSFun.
```

Every operation in this interface, such as `mem` above, is accompanied by a small number of properties that specify the behavior of the operation.

We instantiate all four interfaces for `Set` and `IntSet`.[26] For example, the instance for `Set` starts out:

```
Module SetFSet (E : OrderedType) <: WSfun(E) <: WS <: Sfun(E) <: S.

  Definition t := {s : Set_ elt | WF s}.

  Program Definition In (x :elt) (s : t) : Prop := ....

  Program Definition mem : elt -> t -> bool := member.

  Lemma mem_1 : forall (s : t) (x : elt), In x s -> mem x s = true.
  Proof. ... (* Proof may assume that s is well-formed *) ... Qed.

  ...
End SetFSet.
```

Instantiating these interfaces runs into two small hiccups. The first is that they talk about *all* sets, not simply all *well-formed* sets. Therefore, as in the previous section, we instantiate these interfaces with the subset type `{s : Set_ e | WF s}`. The second is that Coq's module system does not interact with type classes, and `Set_` is defined such that its element type must be an instance of the `Ord` type class. This impedance mismatch requires us to write a module which can generate an `Ord` instance from a Coq `OrderedType` module.

By successfully instantiating this module interface, we obtain two benefits. First, we must prove theorems that cover many of the main functions provided by `containers`; these theorems are particularly valuable, as the interface itself is heavily used by Coq users. Second, by instantiating this interface, we connect our injected Haskell code to the Coq ecosystem, enabling Coq users to easily use the `containers`-derived data structures in their developments, should they so desire.

4.7 Abstract models as specifications

Tests, type classes and the other sources of specifications do not fully describe the intended behavior of all functions. We therefore have to also come up with specifications on our own. We do this by relating a concrete search tree to the abstract set that it represents; that is, we provide a denotational semantics. We denote a set with elements of type `e` as its indicator

[26] In the files examples/containers/theories/SetProofs.v and examples/containers/theories/IntSetProofs.v.

function of type e -> bool; for Set e, we provide a denotation function sem : forall {e} `{Eq_ e}, Set_ e -> (e -> bool), and for IntSet, we provide a denotation relation Sem : IntSet -> (N -> bool) -> Prop.

This approach allows us to abstractly describe the meaning of operations like insert. For Set_, we do this by providing a theorem like

```
Theorem insert_sem:
   forall {a} `{OrdLaws a} (s : Set_ a) (x : a), WF s ->
   forall (i : a), sem (insert x s) i = (i == x) || sem s i.
```

(For IntSet, some technical details differ.) However, there is more we need to know about insert than just its denotation. We also need to know that it preserves well-formedness and bounds, and – to reason about balancing – its size. To avoid having to prove these properties independently, we define a relation Desc that completely describes a set, by asserting that it is well-formed and relating it to its bounds, its size and its denotation:

```
Definition Desc (s : Set _e) (lb ub : option e) (sz : Z) (f : e -> bool) :=
   Bounded s lb ub /\ size s = sz /\ (forall i, sem s i = f i).
```

This allows us to state a single theorem about insert, namely

```
Lemma insert_Desc: forall x s lb ub,
   Bounded s lb ub ->
   isLB lb x = true -> (* If lb is defined, it is less than x *)
   isUB ub x = true -> (* If ub is defined, it is greater than x *)
   Desc (insert x s) lb ub
         (if sem s y then size s else (1 + size s))
         (fun i => (i == x) || sem s i).
```

and prove everything we need to know about insert in one single inductive proof. This theorem describes insert completely: it describes its bounds, its size, its semantics, as well as its well-formedness. [27]

The theorem also introduces a layer of abstraction that we can build upon. In fact, we specify *all* functions this way, and use these specifications, rather than the concrete implementation, to prove the other specifications. (We have an analogous Desc relation for IntSet that describes the properties of Patricia trees.)

An alternative abstract model for finite sets is the sorted list of their elements, i.e. the result of toAscList. The meaning of certain operations, like foldr, take or size, can naturally be expressed in terms of toAscList, but would be very convoluted to state in terms of the indicator function, and we use this denotation – or both – where appropriate.

5 Producing verifiable code with hs-to-coq

Identifying what to prove about the code is only half of the challenge – we also need to get the Haskell code into Coq. Ideally, the translation of Haskell code into Gallina using hs-to-coq would be completely automatic and produce code that can be verified as easily

[27] In the file examples/containers/theories/SetProofs.v.

as code written directly in Coq – and for textbook-level examples, that is the case (Spector-Zabusky *et al.*, 2018). However, working with real-world code requires adjustments to the translation process to make sure that the output is both accepted by Coq and amenable to verification.

A core principle of our approach is that the Haskell source code *does not need to be modified* in order to be verified. This principle ensures that we verify "the" containers library (not a "verified fork") and that the verification can be ported to a newer version of the library.

The crucial feature of hs-to-coq that enables this approach is the support for *edits*: instructions to treat some code differently during translation. Edits are specified in plain text files, which also serve as a concise summary of our interventions. The hs-to-coq tool already supported many forms of edits; for example, specifying when names need to be changed, when parts of the module should be ignored or replaced by some other term, when we want to map Haskell types to existing Coq types, or when a recursive function definition needs an explicit termination proof. In the course of this work, we added new features to hs-to-coq – such as the ability to apply rewrite rules, to handle partiality and to defer termination proofs to the verification stage – and extended the provided base library.

In this section we demonstrate some of the challenges posed by translating real-world code, and show how hs-to-coq's flexibility allowed us to not only to overcome them, but also to facilitate subsequently proving the input correct.

5.1 Unsafe pointer equality

An example of a Haskell feature that we cannot expect to translate without intervention is *unsafe pointer equality*. GHC's runtime provides the scarily named function reallyUnsafePtrEqualty#, which the containers library wraps as ptrEq :: a -> a -> Bool. If this function returns True, then both arguments are represented in memory by the same pointer. If this function returns False, we know nothing – this function is underspecified and may return False even if the two pointers are equal. In fact, if containers is compiled with a non-GHC compiler, it will define ptrEq to always return False.

The ptrEq operation is used, for example, in Set.insert x s when x is already a member of s: If ptrEq indicates that the subtree is unchanged, the function skips the redundant re-balancing step – which enhances performance – and returns the original set rather than constructing a semantically equivalent copy – which increases sharing. Because ptrEq is only ever used for optimization such as this, its weak specification is safe – the worst that could happen if it returns False overzealously is that some extra work is done.

Coq does not provide any way of reasoning about memory, so when we use hs-to-coq, we must replace ptrEq with something else. But what?

One option is to replace the definition of ptrEq with a definition that does not do any computation and simply always returns False. This can be done using the following edit:

```
replace Definition ptrEq : forall {a}, a -> a -> bool := fun _ _ _ => false.
```

This allow us to proceed with translation and verification. However, the code in the `True` branch of an unsafe pointer equality test would be dead code in Coq, and our verification would miss bugs possibly lurking there.

Consequently, we choose a different encoding that captures the semantics of `ptrEq` more precisely. We make the definition of `ptrEq` *opaque* and partially specify its behavior.[28]

```
Definition ptrEq_spec :
  { ptrEq : forall a, a -> a -> bool
  | forall a (x y : a), ptrEq _ x y = true -> x = y }.
Proof. apply (exist _ (fun _ _ _ => false)). intros; congruence. Qed.

Definition ptrEq : forall {a}, a -> a -> bool := proj1_sig ptrEq_spec.
Lemma ptrEq_eq : forall {a} (x:a)(y:a), ptrEq x y = true -> x = y.
Proof. exact (proj2_sig ptrEq_spec). Qed.
```

Here, we define the `ptrEq` function together with a specification as `ptrEq_spec`. Although the function in `ptrEq_spec` also always returns `false`, the **`Qed`** at the end of its definition completely hides this implementation of `ptrEq`. While the specification `ptrEq_eq` is vacuously true, making this definition opaque forces verification to proceed down both paths. While could achieve the same using **`Axiom`**, our variant protects us from accidentally introducing inconsistencies to Coq.

There is no way we could define `ptrEq` any more precisely: any function defined in Coq respects structural equality, but a `ptrEq` that could return `true` would have to distinguish between structurally-equal terms. The only way `ptrEq` can respect structural equality *and* return `true` on values at different locations in memory is by never returning `true` at all. And because `ptrEq` is underspecified, this is consistent; in fact, as mentioned above, the constantly-`False` function is the exact definition used by `containers` if the underlying GHC primitive is not available.

We do have to trust that GHC's definition of pointer equality has this specification, but given this assumption, we can soundly verify that pointer equality is used correctly.

5.2 Evaluation order

A shallow embedding of Haskell into Coq makes the difference between strict and lazy code vanish, because Gallina is a total language and does not care about evaluation order.

Haskell has "magic" functions like `seq` that allow the programmer to explicitly control strictness, and the `containers` library uses it to improve performance. Its effect is irrelevant in Coq, and we instruct `hs-to-coq` to use this simple, magic-free implementation for it:

```
Definition seq {a} {b} (x : a) (y : b) := y.
```

Set: API: deleteAt, deleteFindMax, deleteFindMin, elemAt, findIndex, findMax, findMin
Instances: Data, IsList, NFData, Read, Show, Show1
Internal functions: showTree

IntSet: API: deleteFindMax, deleteFindMin, findMax, findMin, fromAscList, fromDistinctAscList
Instances: Data, IsList, NFData, Read, Show
Internal functions: showTree

Map: API: alterF, atKeyImpl, deleteFindMax, deleteFindMin, differenceWith, differenceWithKey, elemAt, findMax, findMin
Instances: Data, IsList, NFData, Ord1, Read, Read1, Show, Show1, Show2
Internal functions: , alterFCutoff, alterFFallback, alterFYoneda, bogus, deleteAlong, filterGt, filterLt, find, insertAlong, lookupTrace, replaceAlong

Fig. 7. Untranslated functions and type classes in Data.Set, Data.IntSet, and Data.Map.Strict

5.3 Eliminating unwanted parts of the code

Figure 7 lists the untranslated portions of the Set, IntSet, and Map modules. This makes up 15% of the code, as can also be seen in Figure 3, and is smaller than the parts that were verified (see Figure 4).

Many of these operations are functions that we choose to ignore for the sake of verification – for example, the function showTree in Data.Set prints the internal structure of such a set as an ASCII-art tree. This function is not used elsewhere in the module. In the interest of a tidier and smaller output, we skip this function using an edit:

```
skip showTree
```

Similarly, we skip functionality related to serialization (the Show and Show1 type classes), deserialization (the Read type class), generic programming (the Data type class), and overloaded list notation (the IsList type class).

Furthermore, we skip some operations whose public API is partial. For example, evaluating findMax empty will call error and throw an exception, as the empty set has no maximum element. We cannot model this exception in Coq (see Section 5.4 for how we handle calls to error within total functions), so we skip findMax and similar functions (findMin, deleteFindMax, deleteFindMin, findIndex, elemAt and deleteAt). This elision is not significant because the containers API provides total equivalents for many of these functions (e.g., lookupIndex, which returns Nothing when the index is out of bounds).

Finally, we skip the two functions that use mutual recursion as this feature is not yet supported by hs-to-coq (fromAscList and fromDistinctAscList). While basic structural mutual recursion is supported by Coq, the existing facilities for nonstructural recursion (such as **Program Fixpoint**; see Section 5.7) do not support mutual recursion.

[28] In the file examples/containers/lib/Utils/Containers/Internal/PtrEquality.v.

5.4 Partiality in total functions

In contrast to the skipped functions above, some functions use partiality in their implementation in ways that cannot be triggered by a user of the public API. In particular, they may use calls to Haskell's `error` function when an invariant is violated.

For example, the central balancing functions for `Set`s, `balanceL` and `balanceR`, may call `error` when passed an ill-formed `Set`. Because our proofs only reason about well-formed sets, this code is actually dead. It does not matter how we translate `error` – any term that is accepted by Coq is good enough. However, `error` in Haskell has the type

```
error :: String -> a
```

which means that a call to error can inhabit *any* type. We cannot define such a function in Coq, and adding it as an axiom would be glaringly unsound.

Therefore, we extended `hs-to-coq` to use the following definition for `error`:

```
Class Default (a :Type) := { default : a }.
Definition error {a} `{Default a} : String -> a.
Proof. exact (fun _ => default). Qed.
```

The type class enforces that we use `error` only at non-empty types, ensuring logical consistency. Yet we will notice that something is wrong when we have to prove something about it. Just as with `ptrEq` (Section 5.1), by making the definition of `error` opaque using **`Qed`**, we are prevented from accidentally or intentionally using the concrete `default` value of a given type in a proof about `error`. Furthermore, when we extract the Coq code back to Haskell for testing, we translate this definition back to Haskell's `error` function, preserving the original semantics.

This encoding is inspired by Isabelle, where all types are inhabited and there is a polymorphic term `undefined :: a` that denotes an unspecified element of any type.

5.5 Translating the `Int` in `IntSet`

As discussed in Section 4.1, we map Haskell's finite-width integer type `Int` to Coq's unbounded integer type `Z` in the translation of `Data.Set` in order to match the specification that integer overflow is outside the scope of the specified behavior.

For `IntSet`, however, this choice would cause problems. Big-endian Patricia trees require that two different elements have a highest differing bit. This is not the case for `Z`, where negative numbers have an infinite number of bits set to `1`; for instance, `-1` is effectively an infinite sequence of set bits. Fortunately, `hs-to-coq` is flexible enough to allow us to make a different choice when translating `IntSet`; we can pick any suitable type where all elements have a finite number of bits set, such as the natural numbers (`N`) or a fixed width integer type.

Given that Coq's standard library provides a fairly comprehensive library of lemmas about `N` and decision procedures (`omega` and `lia`) that work with it, we chose to use `N` for now, with the intention to eventually switch to a 64-bit integer type. This is the appropriate generalization of `IntSet` to an infinite domain. In Haskell, the domain is 64-bit words, which happen to be interpretable as negative numbers. When we generalize to an infinite domain, we generalize to bit strings of unbounded but finite length, which we can most simply interpret as nonnegative.

The `IntSet` code uses bit-level operations, like `complement` and `negate`, that do not exist for `N`. To deal with this we extended `hs-to-coq` with support for *rewrite edits* like

```
rewrite forall x y, (x .&. complement y) = (xor x (x .&. y))
```

which instruct it to replace any expression that matches the left-hand side by the right hand side. For signed or bounded integer types, both sides are equivalent. For unbounded unsigned types, like Coq's type `N`, the left hand side is undefined (values in `N` have no complement in `N`), while the right hand side is perfectly fine. When we switch to bounded integers in the `IntSet` code, we can remove these edits.

5.6 Low-level bit twiddling

The `containers` library uses highly tuned bit-twiddling algorithms to operate on `IntSet`s. For example, the function `revNat` reverses the order of the bits in a 64-bit number:

```
revNat :: Nat -> Nat
revNat x1 = case ((x1 `shiftRL` 1) .&. 5555555555555555) .|.
                 ((x1 .&. 5555555555555555) `shiftLL` 1) of x2 ->
            case ((x2 `shiftRL` 2) .&. 3333333333333333) .|.
                 ((x2 .&. 3333333333333333) `shiftLL` 2) of x3 ->
            case ((x3 `shiftRL` 4) .&. 0F0F0F0F0F0F0F0F) .|.
                 ((x3 .&. 0F0F0F0F0F0F0F0F) `shiftLL` 4) of x4 ->
            case ((x4 `shiftRL` 8) .&. 00FF00FF00FF00FF) .|.
                 ((x4 .&. 00FF00FF00FF00FF) `shiftLL` 8) of x5 ->
            case ((x5 `shiftRL` 16) .&. 0000FFFF0000FFFF) .|.
                 ((x5 .&. 0000FFFF0000FFFF) `shiftLL` 16) of x6 ->
            (x6 `shiftRL` 32) .|. (x6 `shiftLL` 32)
```

Though complicated, this code is within the scope of what `hs-to-coq` can translate, and we can verify its correctness.[29]

However, we can't keep up with *all* their tricks. For example, `indexOfTheOnlyBit`, which was contributed by Edward Kmett,[30] takes a number with exactly one bit set and calculates the index of said bit. It does so by unboxing[31] the input, multiplying it by a magic constant, and using the upper 6 bits of the product as an index into a table stored in an unboxed array literal. This manifests as the following scary-looking code:

```
indexOfTheOnlyBit :: Nat -> Int
indexOfTheOnlyBit bitmask = I# (lsbArray `indexInt8OffAddr#` unboxInt
        (intFromNat ((bitmask * magic) `shiftRL` offset)))
```

[29] In the file examples/containers/theories/RevNatSlowProofs.v.

[30] `https://github.com/haskell/containers/commit/e076b33f`

[31] Unboxing is analogous to the `{-# UNPACK #-}` pragma mentioned in Section 2.2; unpacking a field stores an unboxed value there. Unboxed values are always fully evaluated, bypassing the uniform, lazy representation common to all other Haskell values. For example, unboxed `Int`s are machine words and may be stored in registers.

```
--Less obvious structural recursion
link :: a -> Set a -> Set a -> Set a
link x Tip r = insertMin x r
link x l Tip = insertMax x l
link x l@(Bin sizeL y ly ry) r@(Bin sizeR z lz rz)
   | delta*sizeL < sizeR = balanceL z (link x l lz) rz
   | delta*sizeR < sizeL = balanceR y ly (link x ry r)
   | otherwise           = bin x l r
--Well-founded recursion
foldlBits :: Int -> (a -> Int -> a) -> a -> Nat -> a
foldlBits prefix f z bitmap = go bitmap z
   where go 0  acc = acc
         go bm acc = go (xor bm bitmask) ((f acc) $! (prefix+bi))
            where !bitmask = lowestBitMask bm
                  !bi = indexOfTheOnlyBit bitmask
--Deferred recursion
fromDistinctAscList :: [a] -> Set a
fromDistinctAscList [] = Tip
fromDistinctAscList (x0 : xs0) = go (1::Int) (Bin 1 x0 Tip Tip) xs0
   where go !_ t []      = t
         go s l (x : xs) = case create s xs of (r :*: ys) ->
                           let !t' = link x l r in go (s `shiftL` 1) t' ys

         create !_ [] = (Tip :*: [])
         create s xs@(x : xs')
            | s == 1      = (Bin 1 x Tip Tip :*: xs')
            | otherwise = case create (s `shiftR` 1) xs of
                            res@(_ :*: []) -> res
                            (l :*: (y:ys)) -> case create (s `shiftR` 1) ys of
                               (r :*: zs) -> (link y l r :*: zs)
```

Fig. 8. Recursion styles

```
  where
       unboxInt (I# i) = i
       magic           = 0x07EDD5E59A4E28C2
       offset          = 58
       !lsbArray       = "\63\0\58\1\5...15\8\23\7\6\5"#
```

We currently cannot translate this code because `hs-to-coq` does not yet support unboxed arrays or unboxed integers. Even if we could (such as by treating the unboxed values as ordinary boxed values), the algorithm it uses depends crucially on the finite width of machine words, relying on both a finite lookup table (`lsbArray`) as well as on unsigned integer overflow in the multiplication. We therefore replace it with a simpler but more general definition based on a integer logarithm function provided by Coq's standard library:

```
redefine Definition indexOfTheOnlyBit := fun x => N.log2 x.
```

Similarly, we provide simpler definitions for the four low-level bit-twiddling functions `branchMask`, `mask`, `zero` and `suffixBitMask`.

5.7 Non-trivial recursion

In order to prove the correctness of `Set` and `IntSet`, we must deal with termination. There are two reasons for this. First, we intrinsically want to prove that none of the functions provided by `containers` go into an infinite loop. Second, Coq requires that all defined functions are terminating, as unrestricted recursion would lead to logical inconsistencies. Depending on how involved the termination argument for a given function is, we use one of the following four approaches.

5.7.1 Obvious structural recursion

By default, `hs-to-coq` implements recursive functions directly using Coq's **`fix`** keyword. This works smoothly for primitive structural recursion; indeed, a majority of the recursive functions that we encountered, such as `member` in Figure 1, were of this form and required no further attention.

5.7.2 Almost-structural recursion

Another common recursion pattern can be found in binary operations such as `link` in Figure 8. Here, every recursive call shrinks *either or both* of its arguments to immediate subterms of the originals, leaving the others unchanged. This almost-structural recursion is already beyond the capabilities of Coq's termination checker. We therefore instruct `hs-to-coq` to use Coq's **`Program Fixpoint`** command to translate these functions in terms of well-founded recursion by adding the edits

```
termination link {measure (size_nat arg_0__ + size_nat arg_1__)}
obligations link termination_by_omega
```

This specifies both:

1. The termination measure, which is the sum of the sizes of the arguments (we defined the function `size_nat : IntSet -> nat`).
2. The termination proof that the measure decreases on every call. This is represented as the Coq tactic `termination_by_omega`, which is a thin wrapper we defined around `omega`, a Coq tactic to decide linear integer arithmetic.

We can use these edits (with `size_nat` and `termination_by_omega`) to get Coq accepts such recursive definitions without the need for any further proofs or manual intervention.

5.7.3 Well-founded recursion

A small number of functions recurse in a non-structural way, such as `foldlBits` in Figure 8, which recurses on the input after clearing the least-significant set bit (``bm `xor` (lowestBitMask bm)``). We can handle this sort of logic using the same machinery as before, but now we have to write a *specialized* termination tactic and declare it in the `obligations` hint. To do so, we need to prove necessary lemmas *before* translating the Haskell module in question. In particular, our lemmas cannot mention functions that are defined in the translated Haskell module. We can do that by exploiting the fact that Coq's value definitions are not generative: since the function `lowestBitMask` is defined elsewhere to be `lowestBitMask (bm : N) = 2 ^ N_ctz bm`, we can still use lemmas such as `foldlBits_proof` to reason about `lowestBitMask`:

```
Lemma foldlBits_proof: forall a,
  N.eqb a 0 = false -> N.to_nat (N.lxor a (2 ^ N_ctz a)) < N.to_nat a.
```

Coq's **`Program Fixpoint`** only supports top-level functions, but we frequently encounter local recursive functions – the `go` idiom, as seen here. To support this, we extended `hs-to-coq` to offer some of the convenience provided by **`Program Fixpoint`** by translating local recursive functions using the same well-founded-recursion-based fixed-point combinator as **`Program Fixpoint`**.

5.7.4 Deferred recursion

Finally, we encounter some functions that require elaborate termination arguments, such as `fromDistinctAscList` in Figure 8. It has two local recursive functions, `go` and `create`, and to convince ourselves that `fromDistinctAscList` is indeed terminating, we have to reason as follows:

> *The function `create` bitshifts its first argument to the right upon each recursive call, until the argument is 1. Ergo, it is terminating, but only for positive input – it clearly loops if `x` is `0`. The function `go` recurses on smaller lists as its third argument, but to see that, we first have to convince ourselves that the list in the tuple returned by `create` is never larger than the list passed to `create`. Also, `go` calls `create`, so we need to ensure that the `s` passed to it is positive. The `go` function bitshifts `s` to the left at every call, so if `go` is called with a positive `s`, then `s` will remain positive in recursive calls. Finally, we see that `fromDistinctAscList` calls `go` with `s` equal to 1, which is positive, so we can conclude that `fromDistinctAscList` terminates.*

If we wanted to convince Coq of this termination pattern, we would have to turn `create` and `go` into top-level definitions, change their types to rule out invalid (non-positive) inputs, define `create` using **`Program Fixpoint`**, and provide an explicit termination argument by well-founded recursion. Then we could prove that `create` preserves the list lengths, which we need to define `go`, again using **`Program Fixpoint`**. This is certainly possible, but it is not simple, especially in an automatic translation.

For hard cases like these we resort to *deferred termination checking*, a feature that we added to `hs-to-coq`. We can instruct it to use the following axiom as a very permissive fixed-point combinator and translate the code of `fromDistinctAscList` essentially unchanged:

```
Axiom deferredFix :
  forall {a r} `{Default r}, ((a -> r) -> (a -> r)) -> a -> r.
```

On its own, `deferredFix` does not do anything; it merely sits in the translated code applied to the original function body. It is consistent, since its type could be implemented by a function that always returns `default` (see Section 5.4). And it does not prevent the user from running extracted code – we can extract this axiom to the target language's unrestricted fixpoint operator (e.g., `Data.Function.fix` in Haskell), although this costs us the guarantee that the extracted code is terminating.

When it comes time to verifying something about a function that is defined using `deferredFix`, we need to give `deferredFix` meaning. We do so using a second axiom, `deferredFix_eq_on`, which states that for any well-founded relation `R` (`well_founded R`), if the recursive calls in `f` are always at values that are strictly `R`-smaller than the input (`recurses_on R`), then we may unroll the fixpoint of `f`:

```
Definition recurses_on {a b}
   (P : a -> Prop) (R : a -> a -> Prop) (f : (a -> b) -> (a -> b))
   := forall g h x, P x -> (forall y, P y -> R y x -> g y = h y) -> f g x = f h x.
```

```
Axiom deferredFix_eq_on : forall {a b} `{Default b}
   (f : (a -> b) -> (a -> b)) (P : a -> Prop) (R : a -> a -> Prop),
   well_founded R -> recurses_on P R f ->
   forall x, P x -> deferredFix f x = f (deferredFix f) x.
```

The predicate `P : a -> Prop` allows us to restrict the domain to inputs for which the function is actually terminating – crucial for go and `create`.

The predicate `recurses_on P R f` characterizes the recursion pattern of `f`, but does so in a very extensional way and only considers recursive calls that can actually affect the result of the function. For instance, it would consider a recursive function defined by `f n = if f n then true else true` to be terminating.

We can use this non-trivial axiom without losing too much sleep, because we we have an implementation of `deferredFix` and `deferredFix_eq_on` in terms of classical logic and the axiom of choice (as provided by the Coq module `Coq.Logic.Epsilon`).[32] This means that both axioms are consistent with plain Coq. We do not know if `deferredFix_eq_on` is strictly weaker than classical choice, so users of `hs-to-coq` who want to combine the output of `hs-to-coq` with developments known to be inconsistent with classical choice (e.g., homotopy type theory) should be cautious.

Pragmatically, working with `deferredFix` is quite convenient, as we can prove termination together with the other specifications about these functions. The actual termination proofs themselves are not fundamentally different from the proof obligations that **`Program Fixpoint`** would generate for us and – although not needed in this example – can be carried out even for nested recursion through higher-order functions like `map`.

This approach to defining recursive functions via extensionality was inspired by Isabelle's function package (Krauss, 2006). It is also an instance of the recursion schemes described Charguéraud (2010a).

5.8 Translating Haskell tests to Coq types

When we translate code, we usually want to preserve the semantics of the code as much as possible. Things are very different when we translate the QuickCheck tests defined in the `containers` test suite, as we discussed in Section 4.2: whereas the semantics of the test suite in Haskell is a program that creates random input to use as input to test executable

[32] In the file base/GHC/DeferredFixImpl.v.

properties, we want to re-interpret these properties as logical propositions in Coq. Put differently, we are turning executable code into *types*.

Recall the example QuickCheck property that we saw Section 4.2, which specifies that set union is associative in terms of the Haskell-level equality function ==:

```
prop_UnionAssoc :: IntSet -> IntSet -> IntSet -> Bool
prop_UnionAssoc t1 t2 t3 = union t1 (union t2 t3) == union (union t1 t2) t3
```

How does QuickCheck work with this?

QuickCheck's API provides types and type classes for writing property-based tests. In particular, it defines an opaque type `Property` that describes properties that can be checked using randomized testing, a type class `Testable` that converts various testable types into a `Property`, and a type constructor `Gen` that describes how to generate a random value. These are combined, for instance, in the QuickCheck combinator

```
forAll :: (Show a, Testable prop) => Gen a -> (a -> prop) -> Property
```

which tests the result of a function on inputs generated by the given generator. The `Testable` instance for functions, for example, uses `forAll` to generate an input and then tests the output; testing `prop_UnionAssoc` thus involves, essentially, evaluating

```
forAll (arbitrary :: Gen IntSet) $ \t1 ->
  forAll (arbitrary :: Gen IntSet) $ \t2 ->
    forAll (arbitrary :: Gen IntSet) $ \t3 ->
      toProp $ union t1 (union t2 t3) == union (union t1 t2) t3
```

Since we want to *prove*, not *test*, these properties, we do not convert the QuickCheck implementation to Coq. Instead, we write a small Coq module that provides the necessary pieces of the interface of `Test.QuickCheck`, but interprets these types and functions in terms of Coq propositions. In particular:

- We use Coq's non-computational type of propositions, **`Prop`**, instead of using QuickCheck's computational type of results `Property`;
- `Gen a` is simply a wrapper around a logical predicate on `a`; and
- `forAll` quantifies (using Coq's **`forall`**) over the type `a`, and ensures that the given function – now a predicate – holds for all members of `a` that satisfy the given "generator".

Concretely, this leads to the following adapted Coq code:

```
Record Gen a := MkGen { unGen : a -> Prop }.
Class Testable (a : Type) := { toProp : a -> Prop }.
Definition forAll {a prop} `{Testable prop}
                  (g : Gen a) (p : a -> prop) : Prop :=
  forall (x : a), unGen g x -> toProp (p x).
```

We provide similar translations for QuickCheck's operators `===`, `==>`, `.&&.` and `.||.`, and we replace generators such as `choose :: Random a => (a, a) -> Gen a` with their corresponding predicates.

With this module in place, `hs-to-coq` translates the test suite into a "proof suite". As we saw in Section 4.2, a test like `prop_UnionAssoc` is now a definition of a Coq proposition, that is to say a type, and can be used as the type of a theorem:

API: adjust, alter, assocs, balanceL, balanceR, delete, deleteMax, deleteMin, difference, drop, elems, empty, filter, findIndex, findWithDefault, foldl, foldl', foldlWithKey', foldr, foldr', foldrWithKey, fromAscList, fromDescList, fromDistinctAscList, fromDistinctDescList, fromList, fromSet, glue, insert, insertLookupWithKey, insertMax, insertWith, insertWithKey, intersection, keys, keysSet, link, link2, lookup, lookupLE, lookupLT, lookupMax, lookupMin, mapEither, mapMaybe, mapMaybeWithKey, mapWithKey, maxView, member, minView, notMember, null, partition, restrictKeys, singleton, size, split, splitAt, splitLookup, splitRoot, take, toAscList, toDescList, toList, union, unionWithKey, unions, unionsWith, update, updateLookupWithKey, updateWithKey, withoutKeys
Instances: Eq, Eq1, Monoid, Semigroup
Internal functions: insertMin, insertR, insertWithKeyR, insertWithR, isProperSubmapOfBy, maxViewSure, minViewSure, splitMember

Fig. 9. Verified functions from Data.Map.Strict

```
Theorem thm_UnionAssoc : toProp prop_UnionAssoc.
```

If we evaluate toProp, we get that this theorem equivalent to the following expanded form (which we saw back in Section 4.2):

```
Theorem thm_UnionAssoc:
  forall t1, WF t1 -> forall t2, WF t2 -> forall t3, WF t3 ->
  union t1 (union t2 t3) == union (union t1 t2) t3 = true.
```

Here, the WF constraints correspond to the arbitrary generators in Haskell, which were constrained to only generate well-formed IntSets.

5.9 Verifying Data.Map.Strict

We also used hs-to-coq to verify large parts of Data.Map.Strict, a finite map implementation from the containers library.[33] Since the underlying data structure – a weight-balanced tree – is the same as that of Data.Set, many of the specifications and proofs are broadly similar. As before, the specifications draw on type class laws, Coq's FMapInterface module, and an abstract definition similar to that of Section 4.7.

We used the function sem: **forall** {e a} `{Eq_ e}, Map e a -> e -> option a as the denotation for maps. This type introduces added complexity into the specification and proofs relative to the indicator function for sets, as we use an option a instead of simply a boolean value. When working with a Set, it is sufficient to prove that a given element exists somewhere in the set; but when working with a Map, we must show that a given key not only appears, but also is mapped to the correct value. For example, in fromList, the description specifies that if a key appears multiple times in the pairs of an input list, the last value in the list is the one that is used in the map.

Nevertheless, only slight modifications of the specifications and proofs developed for Data.Set (such as the Desc relation described in Section 4.7) were required to be able to

[33] The library comes in two varieties: Data.Map.Strict, which forces the evaluation of the values in the map; and Data.Map.Lazy, which does not. As we do not preserve evaluation order (as discussed in Section 5.2, the difference is not relevant to our work.

handle this additional complexity. Furthermore, this machinery is also applicable to functions in the Data.Map.Strict API that have no analogue in Data.Set. These extensions provide further evidence for the utility and flexibility of the original specifications.

Our verification of Data.Map.Strict had one source of specification not available to Data.Set. In other work, we have experimented with the use of hs-to-coq in order to verify optimization passes in GHC's intermediate language, Core.[34] These optimizations use finite maps to record variable sets and environments. Since we were verifying these before we had verified Data.Map, during the verification process, we recorded the properties of the maps that were required as a list of over thirty axioms. For example, one axiom states that if a key-value pair is inserted in a map, then looking up the given key should return the given value. This property was expressed as the following:

```
Lemma lookup_insert: forall key value (m: Map e a),
 WF m ->
 lookup key (insert key value m) = Some value.
```

In this work, we were able to prove all of these lemmas using our abstract Desc specifications in a straightforward way.

Lastly, the vast majority of the Data.Map.Strict proofs were completed by the sixth author, an undergraduate with limited Haskell and Coq experience. This both demonstrates the utility of the hs-to-coq tool and suggests that producing verified code is a realistic task even without significant expertise.

6 Contributions to containers: Theory and Practice

We chose the Set, IntSet, and Map modules of the containers library as our target because they nicely represent the kind of Haskell code that we want to see verified in practice. Nevertheless, the deep understanding required to verify these modules led to new insights into the algorithms themselves and to improvements to their Haskell implementation.

6.1 New insight into the verification of weight-balanced trees

The data structure underlying Set and Map was originally presented by Nievergelt & Reingold (1972). It is a binary search tree with the invariant that if s_1 is the size of the left subtree and s_2 the size of the right subtree of a node, then

$$\mathrm{bal}_{\mathrm{NR}}(s_1,s_2) := (s_1+1) \leq \delta \cdot (s_2+1) \wedge (s_2+1) \leq \delta \cdot (s_1+1)$$

holds for a balancing tuning parameter δ. In 1992, Adams suggested a variant of the balancing condition, namely

$$\mathrm{bal}(s_1,s_2) := s_1+s_2 \leq 1 \vee (s_1 \leq \delta \cdot s_2 \wedge s_2 \leq \delta \cdot s_1).$$

The conditions are very similar, but not equivalent: the former allows, for example, $\delta = 3$, $s_1 = 2$ and $s_2 = 0$, which the latter rejects.

[34] See the examples/ghc directory.

Initially, the containers library used Adams's balancing condition with the parameters $\delta = 4$ (for sets) or $\delta = 5$ (for maps). Campbell (2010) found that these parameters are actually invalid and exhibited a sequence of insertions and deletions that produced an unbalanced tree. Subsequently, the containers library switched to $\delta = 3$ in both modules. Inspired by this bug report, Hirai & Yamamoto (2011) thoroughly analyzed this data structure with the help of a Coq formalization, and identified the valid range for the balancing parameter δ, albeit only for Nievergelt & Reingold's variant – our proof seems to be the first mechanical verification of Adam's variant.

Given such thorough analysis of the algorithms, we did not expect to learn anything new about this data structure, and for the most part, this was true. Our proofs are free of any manual calculations about tree sizes and the balancing condition. We just state the proper preconditions for each lemma, and Coq's automation for linear integer arithmetic, lia (Besson, 2006), takes care of the rest.

One exception was the crucial function balanceL which is used, according to the documentation "when the left subtree might have been inserted to or when the right subtree might have been deleted from". This suggests the precondition

$$(\mathrm{bal}(s_1 - 1, s_2) \land 0 < s_1) \lor \mathrm{bal}(s_1, s_2) \lor \mathrm{bal}(s_1, s_2 + 1)$$

corresponding to the three cases: left tree inserted to, no change, and right tree deleted from. This is also what Hirai and Yamamoto used in their formalization of Nievergelt & Reingold's variant. And indeed, this precondition is strong enough to verify that the output of balanceL is balanced – but we found the precondition is *too* strong. The link operation, shown in Figure 8, is supposed to balance two arbitrary trees using balanceL. In the verification of link we were unable to satisfy this precondition.

We found a precondition for balanceL that is both strong enough for the verification of balanceL and weak enough to allow the verification of link, namely

$$(\mathrm{bal}^*(s_1 - 1, s_2) \land 0 < s_1) \lor \mathrm{bal}(s_1, s_2) \lor \mathrm{bal}(s_1, s_2 + 1)$$

where we used the following modified balancing condition for the left tree (the one we inserted into)

$$\mathrm{bal}^*(s_1, s_2) := \delta \cdot s_1 \le \delta^2 \cdot s_2 + \delta \cdot s_2 + s_2 \land s_2 \le s_1.$$

No creativity was involved in coming up with this formula: In the proof of link, in the case where we call balanceL with trees with sizes s_1 and s_2, the interactive proof system conveniently gives us all available facts about the sizes of the tress. We put these into a single formula, namely

$$\exists s\, s_l\, s_r,\, s_1 = 1 + s + s_l \land s_2 = s_r \land \mathrm{bal}(s_l, s_r) \land \delta \cdot s < 1 + s_l + s_r \land 1 \le s \land 0 \le s_l \land 0 \le s_r,$$

and then (manually) eliminated the existential quantifiers to obtain $\mathrm{bal}^*(s_1 - 1, s_2)$.

6.2 Verification of big-endian Patricia trees

There is surprisingly little literature on the verification of big-endian Patricia trees. They are sorted search trees, but as they exploit the additional structure of their keys being bit strings, additional theory is required to verify them.

$$\frac{\begin{array}{c} \mathtt{s}_1 :: r_1 :: D_1 \qquad \mathtt{s}_2 :: r_2 :: D_2 \\ r_1 \subseteq h_1(d(p,b)) \qquad r_2 \subseteq h_2(d(p,b)) \qquad b > 0 \qquad \mathtt{p'} = p \cdot 2^b \qquad \mathtt{m} = 2^{b-1} \end{array}}{\mathtt{Bin\ p'\ m\ s_1\ s_2} :: d(p,b) :: D_1 \cup D_2}$$

$$\frac{\mathtt{p'} = p \cdot w \qquad 0 < \mathtt{bm} < 2^w}{\mathtt{Tip\ p'\ bm} :: d(p, \log_2 w) :: \{p \cdot w + j \mid \text{bit } j \text{ set in } \mathtt{bm}\}}$$

Fig. 10. The denotation of `IntSet`

In particular, where the verification of regular search trees uses open intervals to describe the range of possible values of a tree, Patricia trees build on *dyadic intervals*. These intervals are of the form $[p \cdot 2^b, \ldots, (p+1) \cdot 2^b - 1] =: d(p,b)$ for some prefix $p \in \mathbb{N}$ and bit-width $b \in \mathbb{N}$, and have an interesting algebraic structure that plays a crucial role in the verification. In particular:[35]

- A dyadic interval is non-empty.
- Either two dyadic intervals are disjoint, or one is contained in the other.
- A dyadic interval $d(p,b)$ with $b > 0$ is the disjoint union of its two halves, $h_1(d(p,b)) := d(p,b-1)$ and $h_2(d(p,b)) := d(p+1,b-1)$.
- In particular, if another dyadic interval r overlaps with with both halves of $d(p,b)$, then $d(p,b) \subseteq r$.
- The dyadic intervals, ordered by inclusion, form a join semi-lattice: for any two dyadic intervals r_1, r_2 there exists a least dyadic interval $r_1 \sqcup r_2$ that contains the two.

The verification of the `IntSet` library[36] revolves around the relation $\mathtt{s} :: r :: D$, given in Figure 10. This relation expresses that a non-empty tree `s` of type `IntSet` describes the set D, which is contained in the dyadic interval r:

Lemma 1
If $\mathtt{s} :: r :: D$, then

- $\emptyset \subset D \subseteq r$ and
- `member x s == true` iff $x \in D$.

We can now specify the operations on Patricia trees in terms of this relation, e.g. the specification for the union operator would be: “if $\mathtt{s}_1 :: r_1 :: D_1$ and $\mathtt{s}_2 :: r_2 :: D_2$, then $\mathtt{union\ s_1\ s_2} :: r_1 \sqcup r_2 :: D_1 \cup D_2$”. By including the specification on the associated dyadic interval in the statement of the lemma, a proof by induction has all necessary information about the subterms to conclude.

This theory applies to natural numbers (bounded or unbounded). Negative numbers require additional thought, as the nice algebraic structure of dyadic intervals breaks down: A dyadic interval is either completely negative or completely non-negative; the two dyadic intervals $r_1 = d(-1,1) = \{-2,-1\}$ and $r_2 = d(0,1) = \{0,1\}$ do not have a least upper bound $r_1 \sqcup r_2$. So in order to apply this theory to signed integers, we have to somehow map them to unsigned integers:

[35] The Coq formalization of this theory is in the file examples/containers/theories/DyadicIntervals.v

[36] In the file examples/containers/theories/IntSetProofs.v.

- For signed integers of a bounded bit-width w – as encountered in IntSet – the usual encoding of negative numbers *two's complement*, is suitable. It represents a negative number i by negating all bits of its absolute value $|i|$ and then adding 1 to the result. This effectively maps the negative numbers $\{-2^{w-1},\dots,-1\}$ onto the range $\{2^{w-1},\dots 2^{w}-1\}$. Most operations only use bit-level operations and work just fine. Operations related to order (e.g. toList, findMin) need to treat a Bin constructor that branches on the highest bit specially, as in this case, the right subtree contains the smaller numbers.
- For signed unbounded types, the two's complement representation is unsuitable: It represents -1 as a number with infinitely many bits set, which we cannot just interpret as a unsigned number. In order to build a IntSet-like data structure for signed unbounded types, one would either have to chose a different encoding of natural numbers where all numbers are represented with finitely many bit set (such as *signed magnitude representation*), or simply use a pair of sets of absolute values, one for the negative and one for the positive numbers.

6.3 Improvements to containers

Although our verification did not find bugs in the code of the library, we were able to improve containers: during the verification of Data.Set.union, we noticed that it was using a nested pattern match to check if an argument is a singleton set, when it could be testing if the size was 1 directly. This change turned out to provide a 4% speedup to union, as measured by the benchmark suite i container, and has been released with containers 0.6.[37]

Additionally, as we mention in Section 4.1, our well-formedness property for IntSet uncovered a weakness in the valid function for IntSet and IntMap, which was used in the test suite. The valid function failed to ensure that some of the invariants hold recursively in the tree structure, which was necessary for the proof. We notified the containers maintainers,[38] who then made the valid function complete.

We also provide a package, containers-verified,[39] which re-exports the types and definitions we have verified from the precise version of containers we are working with. This way, a developer who wants to use only the verified portion of the implementation can replace their dependency on containers with a dependency on containers-verified.

7 Assumptions and Limitations

We have shown a way to make mechanically checked, formal statements about existing Haskell code, and have applied this technique to verify parts of the containers library. But are the theorems that we prove actually true? And if they are, how useful is this method?

[37] https://github.com/haskell/containers/commit/b1a05c3a2

[38] https://github.com/haskell/containers/issues/522

[39] https://hackage.haskell.org/package/containers-verified

7.1 The formalization gap

As always, when a theorem is stated about a object that does not purely exist within mathematics, its validity depends on a number of assumptions.

- First and foremost, we have to assume that Coq behaves as documented and does not allow us to prove false theorems. This is of particular relevance because we rely on fine details of Coq's machinery (e.g. the behavior of **Qed**, Section 5.4) and optionally add consistent axioms (see Section 5.7.4). We also use the axiom of proof irrelevance, which is consistent with Coq and a consequence of classical logic in the Calculus of Inductive Constructions (Coquand, 1989).
- The biggest assumption is that the semantics of a Gallina program, as defined by the Calculus of Constructions (Coquand & Huet, 1988), models the behavior of a running Haskell program in a meaningful way. At this time, we cannot even attempt to close this gap, as there are neither formal semantics nor verified compilers for Haskell. While Haskell and Gallina are very similar (both are pure functional languages with algebraic data types), the significant differences between them (such as Haskell's partiality and laziness) mean that the gap is a real presence.
  We can gain additional confidence by *testing* this connection: we extracted our Gallina versions of `Set` and `IntSet` back to Haskell and successfully ran the original test suite of `containers` against it. Every successful test is further evidence that the corresponding theorem (see Section 4.2) is indeed a theorem about the Haskell program.
- Furthermore, we rely on `hs-to-coq` translating Haskell code into the correct Gallina code. The translator itself is a sizable piece of code, unverified (and such verification is currently out of our reach, not least because there is no formal semantics of Haskell) and therefore surely not free of bugs. We get some confidence into the tool from manually inspecting its output and observing that it is indeed what we would consider the "right" translation from Haskell into Gallina, and additional confidence from the fact that we were actually able to prove the specifications, which would not be possible if the translated code behaved differently than intended. Moreover, extracting the translated code back to Haskell and running the test suite also stress-tests the translation.
- The translation was not completely automatic and required manual edits. With each edit, we add another assumption to the formalization gap: Does our underspecification of pointer equality encompass the actual behavior of GHC's `reallyUnsafePtrEqualty#`? Given our choice of using unbounded integers, does the size field in a `Set` really never overflow in practice? Are our manually written versions of low-level bit twiddling functions correct? We list and justify our manual interventions in Section 5.

The formalization gap of our work is relatively large compared to, say, the gap for the verification of programs written in Gallina in the first place. But for the purpose of ensuring the correctness of the Haskell code, that is less critical. Even if one of our assumptions is flawed, it is much more likely that the flaw will get in the way of concluding the proofs, rather than allowing us to conclude the proofs without noticing a bug. Incomplete proofs can uncover bugs, too.

7.2 *Limitations of our approach*

We proved multiple specifications about a large part of the code, but there are limits to what theorems we can prove, and what we can prove them about.

Since we work with a shallow embedding of Haskell in Coq, we cannot make statements about the performance of the Haskell code – which is a pity, given that the `containers` library contains highly optimized code and provides clear specifications of the algorithmic complexity of their operations. Similarly, we cannot verify that the operations are as strict or lazy as documented.

We also cannot translate and verify all code in the `containers` library, because some functions use language features not yet supported by `hs-to-coq`, such as mutual recursion and unboxed arrays. When this affects a crucial utility function we can provide a manual translation. This widens the formalization gap, but enables verification of code that depends on it. When it affects less central code, e.g. the `Data` instances, we can simply skip the translation (Section 5.3).

Partiality is a particularly interesting limit. Coq only allows total functions, but practical Haskell code uses partiality, often in a benign way: Calls to `error` in code paths that are unreachable as long as invariants are maintained, or recursive functions that terminate on the actual arguments they are called with, but may diverge on other inputs. Whereas Spector-Zabusky *et al.* (2018) considered such code out of scope, we have found ways to deal with "internal partiality" (see Section 5.4 and Section 5.7.4).

Taking a step back, it might seem that our approach may see limited adoption in the Haskell community because it requires expertise in Coq. But though tied to a Haskell artifact, verification is isolated from the codebase. Haskell programmers can focus on their domain, trying to get the best performance out of the code and without having to know about verification, while proof engineers can work solely within Coq and do not have to be fluent in Haskell.

In this paper we verify a specific version of the Haskell code, and do not discuss ongoing maintenance of such a verification. It remains to be seen how resilient the proofs are against changes in the Haskell code. Changes of syntactic nature, or changes to a function's strictness, might be swallowed by `hs-to-coq`. Other changes might affect the translated code, but still allow the proofs to go through, if our proof tactics are flexible enough. In general, though, we expect that changes in the code require changes in the proofs. Since Coq is an interactive theorem prover, it will at least clearly point out which parts of our development need to be updated.

8 Related work

8.1 *Verification of purely functional data structures*

Purely functional data structures, such as those found in Okasaki's book (1999) are frequent targets of mechanical verification. That said, we believe that we are the first to verify the Patricia tree algorithms that underlie `Data.IntSet`.

Verifying weight-balanced trees in Haskell Similar to `hs-to-coq`, LiquidHaskell (Vazou *et al.*, 2014) can be used to verify existing Haskell code. Users of this tool annotate their

Haskell source files with refinement types and other annotations. LiquidHaskell then uses an SMT solver to automatically discharge proof obligations described by the refinements. This means that LiquidHaskell provides more automation than `hs-to-coq`; however, the language of Coq is higher-order and more expressive than the language of SMT solvers. Vazou *et al.* (2017) compared the experiences of using LiquidHaskell vs. plain Coq, and found that both have advantages.

Vazou *et al.* (2013) also described the use of LiquidHaskell to verify the `Data.Map` module of finite maps from `containers`. Although not the same as `Data.Set`, this code shares the same underlying data structure (weight-balanced trees) and the implementations of the two are similar. Their verification has similarities with our work; they also use unbounded integers as the number representation and leave functions like `showTree` unspecified. However, we develop a richer specification, which includes a semantic description of each operation we verified, constraints about the tree balance, and the ordering of the elements in the tree. In contrast, Vazou *et al.* limit their specifications to ordering only. For example, in addition to showing that the insertion operation preserves the order of elements of the tree, our work also shows that: (1) insertion preserves the balancing conditions of the weight-balanced tree, (2) the `size` field at each node in the tree is maintained correctly (i.e., the `size` is equal to the number of its descendants), and (3) the tree returned by this operation contains the inserted element, and all elements in the original tree, but nothing more. Although it might be possible to replicate our specifications by using theories of finite sets and maps in SMT (Kröning *et al.*, 2009) to encode these properties using refinement types in LiquidHaskell, this approach has not been explored.

Furthermore, our specification also includes type class laws, and we are able to verify that `Set` and `IntSet` have lawful instances of the `Eq`, `Ord`, `Semigroup`, and `Monoid` type classes. When Vazou *et al.*'s original work was developed in 2013, LiquidHaskell did not have the capability to state and prove these properties. Since then, there have been new developments in LiquidHaskell, particularly refinement reflection (Vazou *et al.*, 2018), which could make it possible to specify and prove type class laws.

Both LiquidHaskell and `hs-to-coq` check for termination of Haskell functions. In LiquidHaskell, the termination check is an option that can be deactivated, allowing the sound verification of nonstrict, non-terminating functions (Vazou *et al.*, 2014). In contrast, a proof of termination is a requirement for verifying functions using `hs-to-coq` (Spector-Zabusky *et al.*, 2018). However, `hs-to-coq` is able to take advantage of many options available in Coq for proving termination of non-trivial recursion, including structural recursion, **`Program Fixpoint`** and our own approach based on `deferredFix`. This latter approach allowed us to reason about `fromDistinctAscList` (see Section 5.7.4) and prove that it is indeed terminating; on the other hand, Vazou *et al.* deactivate the termination check for this function.

Verifying weight-balanced trees in other languages Hirai & Yamamoto (2011) implemented a weight-balanced tree similar to Haskell's `Data.Set` library (albeit using the balancing condition of Nievergelt & Reingold (1972)) and mechanically verified its balancing properties in Coq. More recently, Nipkow & Dirix (2018) extended this work to formalize similar weight-balanced trees in Isabelle and further verified the functional correctness of insertions and deletions.

We verify more functions in the `Data.Set` library than prior work. Operations that are unique to our development include `foldl`, `isSubsetOf`, and `fromDistinctAscList`. The code verified both by Hirai & Yamamoto and by Nipkow & Dirix is also different from the latest `containers` library; for example, it does not use pointer equality (Section 5.1). Another difference is that Hirai & Yamamoto defined the `union`, `intersection`, and `difference` functions based on the "hedge union" algorithm, but `containers` has since changed to use the "divide and conquer" algorithm.

Hirai & Yamamoto specify only the balancing constraints, whereas we develop a richer specification that also includes a semantic description of each operation we verified and the ordering of the elements in the tree. We also gained new insights about the balancing conditions of the weight-balanced tree through our verification effort (see Section 6.1). Nipkow & Dirix's specification contains the same properties as ours, but they only specify the behavior of `insert` and `delete`; we have verified a significantly larger set of functions (see Figure 4).

Other verifications of balanced trees There are many existing works on mechanically verifying purely functional balanced trees. We briefly mention a few here. Filliâtre & Letouzey (2004) implemented AVL trees in Coq, and verified their functional correctness as well as their balancing conditions. Appel (2011) did the same thing for red-black trees. These implementations have now become parts of Coq standard library. Charguéraud (2010b) translated OCaml implementations of Okasaki (1999)'s functional data structures to characteristic formulae expressed as Coq axioms. Licata (2012) lectured on verifying red-black trees in Agda at the Oregon Programming Languages Summer School. McBride (2014) showed how to represent the ordering relationships in Agda for general data structures, not just binary search trees. Nipkow (2016) showed how to automatically verify the ordering properties of eight different binary search tree structures, by specifying each in terms of the sorted list of their elements, a method he used again in his verification of weight-balanced trees (Nipkow & Dirix, 2018). Ralston (2009) verified AVL trees in ACL2.

8.2 Verification tools for Haskell

Previous work using `hs-to-coq` has only applied it to small examples. Spector-Zabusky *et al.* (2018) describe three case studies, two of which require less than 20 lines of Haskell. The longest example (the `Bag` module taken directly from the GHC compiler) is 247 lines of code. Furthermore, none of the reasoning required for these examples is particularly deep. Our work provides experience with more complex, externally-sourced, industrial-strength examples.

The `coq-haskell` library (Wiegley, 2017) is a handwritten Coq library designed to make it easier for Haskell programmers to work in Coq. In addition to enabling easier Coq programming, it also provides support for extracting Coq programs to Haskell.

The prototype contract checker `halo` (Vytiniotis *et al.*, 2013) takes a Haskell program, uses GHC to desugar it into the intermediate language Core, and translates the Core program into a first-order logic formula. It then invokes an SMT solver to prove this formula; a successful proof implies that the original program is crash-free.

Haskabelle was hs-to-coq's counterpart in Isabelle. It translated total Haskell code into equivalent Isabelle function definitions. Similar to hs-to-coq, it parsed Haskell, desugared syntactic constructs, and configurably adapted basic types and functions to their counterparts in Isabelle's standard library. It used to be bundled with the Isabelle release, but it has not been updated recently and was dropped from the distribution.

Haskell has been used as a prototyping language for mechanically verified systems in the past. The *seL4 verified microkernel* started with a Haskell prototype that was semi-automatically translated to Isabelle/HOL (Klein *et al.*, 2009; Derrin *et al.*, 2006). The authors found that the availability of the Haskell prototype provided a machine-checkable formal executable specification of the system. They used this prototype to refine their designs via testing, allowing them to make corrections before full verification.

The *Programmatica project* (Hallgren *et al.*, 2004) included a tool that translates Haskell code into the Alfa proof editor. Their tool only produces valid proofs for total functions over finite data structures. The logic of the Alfa proof assistant is based on dependent type theory, but without as many features as Coq. In particular, the Programmatica tool compiles away type classes and nested pattern matching; both of these features are retained by hs-to-coq.

Dybjer *et al.* developed a tool for automatically translating Haskell programs to the *Agda/Alfa proof assistant* (2004). They explicitly note the interplay between testing and theorem proving and show how to verify a tautology checker. Abel *et al.* (2005) translate Haskell expressions into the logic of the Agda 2 proof assistant. Their tool works later in the GHC pipeline than hs-to-coq and translates Core expressions.

8.3 Translating other higher-order functional languages

There are many large and successful verification projects that demonstrate that functional languages are well suited for verification. In contrast to our work, these projects require either re-implementing the code in a new functional language, as is the case for Cogent (O'Connor *et al.*, 2016; Amani *et al.*, 2016) and F* (Swamy *et al.*, 2016); re-implementing the code in a proof assistant, such as HOL4 in the case of CakeML (Myreen & Owens, 2014; Kumar *et al.*, 2014); or taking an SMT solver-based approach, as found in Leon (Blanc *et al.*, 2013). The CakeML and Cogent projects have a different focus than ours, and they provide a higher assurance in their verified code. CakeML (Kumar *et al.*, 2014) has a verified compiler and Cogent has a certifying compiler (O'Connor *et al.*, 2016; Rizkallah *et al.*, 2016). Both tools provide mechanically checked proofs that their shallow embeddings correspond to the functional code being verified.

Cogent is a restricted higher-order functional language (O'Connor *et al.*, 2016) that was used to verify filesystems (Amani *et al.*, 2016). Its compiler produces C code, a high-level specification in Isabelle/HOL, and an Isabelle/HOL refinement proof linking the two (O'Connor *et al.*, 2016; Rizkallah *et al.*, 2016). Chen *et al.* (2017) integrated property-based testing into the Cogent framework. The authors claim that property-based testing enables an incremental approach to a fully verified system, as it allows for the replacement of tests of properties stated in the specification by formal proofs. Our work substantiates this claim, as indeed one of the ways in which we obtain specifications is through the QuickCheck properties provided by Haskell as discussed in Section 4.2.

CakeML is a large subset of ML with a verified compiler and runtime system (Kumar *et al.*, 2014). Users of CakeML can write pure functional programs in the HOL4 proof assistant and verify them in HOL4. They can then extract an equivalent correct by construction CakeML program. This method has been used to verify several data structures including red black trees, crypto protocols, and a CakeML version of the HOL light theorem prover (Myreen & Owens, 2014).

*F** is a general-purpose functional language that allows for a mixture of proving and general-purpose programming (Swamy *et al.*, 2016). Programs can be specified using dependent and refinement types and automatically verified using one of F*'s backend SMT solvers. Its subset Low* (Protzenko *et al.*, 2017) has been used to verify high-assurance optimized cryptographic libraries.

Leon is a verification tool for a pure subset of Scala that checks pre- and postconditions for functions (Blanc *et al.*, 2013). These contracts are given in terms of the `requires` and `ensures` methods that are already present in the Scala standard library, so a sufficiently well-annotated existing Scala program would need no further specification. Leon uses Z3 to ensure that the contracts are satisfied, and is *complete* – if a counterexample exists, it will be found. It also supports extensions, allowing Leon to verify a larger subset of Scala through translation into a simpler form.

8.4 Extraction

The semantic proximity of Haskell and Coq, which we rely on, is also used in the other direction by Coq's support for code extraction to Haskell (Letouzey, 2002). Several projects use this feature to verify Haskell code (Chen *et al.*, 2015; Joseph, 2014). However, since extraction starts with Coq code and generates Haskell, it cannot be used to verify pre-existing Haskell. Although in a certain sense `hs-to-coq` and extraction are inverses, round-tripping does not produce syntactically equivalent output in either direction. In one direction, `hs-to-coq` extensively annotates the resulting Coq code; in the other, extraction ignores many Haskell features and inserts unsafe type coercions. In this work, we use testing to verify that round-tripping produces operationally equivalent output; this provides assurance about the correctness of both `hs-to-coq` and extraction.

CertiCoq (Anand *et al.*, 2017) and Œuf (Mullen *et al.*, 2018) are verified compilers from Gallina to assembly. CertiCoq can compile any Gallina program, while Œuf can only compile a limited subset of Gallina. However, Œuf provides stronger guarantees about the Gallina code that is in the limited subset it translates.

9 Conclusions and Future Work

We verified the two finite set modules that are part of the widely used and highly-optimized `containers` library. Our efforts provide the deepest specification and verification of this code to date, covering more of the API and proving stronger, more descriptive properties than prior work.

In future work, there is yet more to verify in `containers`. For example, we plan to add a version of `IntSet` that uses 64-bit ints as the element type in addition to our current version with unbounded natural numbers. That way users could choose the treatment of overflow that makes the most sense for their application. Furthermore, we hope to adapt our existing proofs of `IntSet` to the analogous `Data.IntMap` module.

The fact that we did not find bugs says a lot about the tools that are already available to Haskell programmers for producing correct code, such as a strong, expressive type system and a mature property-based testing infrastructure. However, few would dare to extrapolate from these results to say that all Haskell programs are bug free! Instead, we view verification as a valuable opportunity for functional programmers and an activity that we hope will become more commonplace.

Acknowledgments This material is based upon work supported by the National Science Foundation under Grant No. 1319880 and Grant No. 1521539.

References

Abel, Andreas, Benke, Marcin, Bove, Ana, Hughes, John, & Norell, Ulf. (2005). Verifying Haskell programs using constructive type theory. *Pages 62–73 of: Haskell Workshop*. ACM.

Adams, Stephen. (1992). *Implementing sets efficiently in a functional language,*. Research Report CSTR 92-10. University of Southampton.

Amani, Sidney, Hixon, Alex, Chen, Zilin, Rizkallah, Christine, Chubb, Peter, O'Connor, Liam, Beeren, Joel, Nagashima, Yutaka, Lim, Japheth, Sewell, Thomas, Tuong, Joseph, Keller, Gabriele, Murray, Toby, Klein, Gerwin, & Heiser, Gernot. 2016 (Apr.). Cogent: Verifying high-assurance file system implementations. *Pages 175–188 of: International Conference on Architectural Support for Programming Languages and Operating Systems*.

Anand, Abhishek, Appel, Andrew, Morrisett, Greg, Paraskevopoulou, Zoe, Pollack, Randy, Savary Belanger, Olivier, Sozeau, Matthieu, & Weaver, Matthew. (2017). CertiCoq: A verified compiler for Coq. *CoqPL Workshop*. CoqPL '17.

Appel, Andrew W. (2011). *Efficient verified red-black trees*.

Appel, Andrew W., Beringer, Lennart, Chlipala, Adam, Pierce, Benjamin C., Shao, Zhong, Weirich, Stephanie, & Zdancewic, Steve. (2017). Position paper: the science of deep specification. *Philosophical transactions of the royal society of london a: Mathematical, physical and engineering sciences*, **375**(2104).

Besson, Frédéric. (2006). Fast reflexive arithmetic tactics: the linear case and beyond. *Pages 48–62 of: TYPES*. Lecture Notes in Computer Science, vol. 4502. Springer.

Blanc, Régis, Kuncak, Viktor, Kneuss, Etienne, & Suter, Philippe. (2013). An overview of the leon verification system: verification by translation to recursive functions. *Pages 1:1–1:10 of: Proceedings of the 4th Workshop on Scala, SCALA@ECOOP 2013, Montpellier, France, July 2, 2013*. ACM.

Bove, Ana, Dybjer, Peter, & Norell, Ulf. (2009). A brief overview of Agda – A functional language with dependent types. *Pages 73–78 of:* Berghofer, Stefan, Nipkow, Tobias, Urban, Christian, & Wenzel, Makarius (eds), *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. Lecture Notes in Computer Science, vol. 5674. Springer.

Brady, Edwin. (2017). *Type-driven development with Idris*. Manning.

Campbell, Taylor. (2010). *Bug in Data.Map*. e-mail to the Haskell libraries mailing list.

Charguéraud, Arthur. (2010a). The optimal fixed point combinator. *Pages 195–210 of: Proceedings of the First International Conference on Interactive Theorem Proving*. ITP'10. Berlin, Heidelberg: Springer-Verlag.

Charguéraud, Arthur. (2010b). Program verification through characteristic formulae. *Pages 321–332 of: ICFP*. ACM.

Chen, Haogang, Ziegler, Daniel, Chajed, Tej, Chlipala, Adam, Kaashoek, M. Frans, & Zeldovich, Nickolai. (2015). Using Crash Hoare logic for certifying the FSCQ file system. *Pages 18–37 of: SOSP*. ACM.

Chen, Zilin, O'Connor, Liam, Keller, Gabriele, Klein, Gerwin, & Heiser, Gernot. (2017). The Cogent case for property-based testing. *Pages 1–7 of: Workshop on Programming Languages and Operating Systems (PLOS)*. Shanghai, China: ACM.

Claessen, Koen, & Hughes, John. (2000). QuickCheck: a lightweight tool for random testing of Haskell programs. *Pages 268–279 of: ICFP*. ACM.

Coquand, T. 1989 (Sept.). *Metamathematical investigations of a calculus of constructions*. Tech. rept. RR-1088. INRIA.

Coquand, Thierry, & Huet, Gérard P. (1988). The calculus of constructions. *Information and computation*, **76**(2/3), 95–120.

Derrin, Philip, Elphinstone, Kevin, Klein, Gerwin, Cock, David, & Chakravarty, Manuel M. T. (2006). Running the manual: An approach to high-assurance microkernel development. *Pages 60–71 of: Haskell Symposium*. ACM.

Dybjer, Peter, Haiyan, Qiao, & Takeyama, Makoto. (2004). Verifying Haskell programs by combining testing, model checking and interactive theorem proving. *Information & software technology*, **46**(15), 1011–1025.

Filliâtre, Jean-Christophe, & Letouzey, Pierre. (2004). Functors for proofs and programs. *Pages 370–384 of:* Schmidt, David (ed), *Programming Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Hallgren, Thomas, Hook, James, Jones, Mark P., & Kieburtz, Richard B. (2004). An overview of the Programatica toolset. *HCSS*.

Hirai, Yoichi, & Yamamoto, Kazuhiko. (2011). Balancing weight-balanced trees. *Journal of functional programming*, **21**(3), 287–307.

Joseph, Adam Megacz. 2014 (May). *Generalized arrows*. Ph.D. thesis, EECS Department, University of California, Berkeley.

Klein, Gerwin, Elphinstone, Kevin, Heiser, Gernot, Andronick, June, Cock, David, Derrin, Philip, Elkaduwe, Dhammika, Engelhardt, Kai, Kolanski, Rafal, Norrish, Michael, Sewell, Thomas, Tuch, Harvey, & Winwood, Simon. (2009). seL4: Formal verification of an OS kernel. *Pages 207–220 of: ACM Symposium on Operating Systems Principles*. Big Sky, MT, USA: ACM.

Krauss, Alexander. (2006). Partial recursive functions in higher-order logic. *Pages 589–603 of: IJCAR*. LNCS, vol. 4130. Springer.

Kröning, Daniel, Rümmer, Philipp, & Weissenbacher, Georg. (2009). A proposal for a theory of finite sets, lists, and maps for the SMT-Lib standard. *Informal proceedings, 7th International Workshop on Satisfiability Modulo Theories at CADE*, vol. 22.

Kumar, Ramana, Myreen, Magnus O., Norrish, Michael, & Owens, Scott. (2014). CakeML: A verified implementation of ML. *Pages 179–191 of: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. New York, NY, USA: ACM.

Letouzey, Pierre. (2002). A new extraction for Coq. *Pages 200–219 of: TYPES*. LNCS, vol. 2646. Springer.

Licata, Dan. (2012). *15-150 Lecture 21: Red-black trees*. Lecture at the Oregon Programming Language Summer School.

The Coq development team. (2016). *The Coq proof assistant reference manual*. LogiCal Project. Version 8.6.1.

McBride, Conor Thomas. (2014). How to keep your neighbours in order. *Pages 297–309 of: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. New York, NY, USA: ACM.

Morrison, Donald R. (1968). PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, **15**(4), 514–534.

Mullen, Eric, Pernsteiner, Stuart, Wilcox, James R., Tatlock, Zachary, & Grossman, Dan. (2018). Œuf: Minimizing the Coq extraction TCB. *Pages 172–185 of: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. New York, NY, USA: ACM.

Myreen, Magnus O., & Owens, Scott. (2014). Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of functional programming*, **24**(May), 284–315.

Nievergelt, Jürg, & Reingold, Edward M. (1972). Binary search trees of bounded balance. *Pages 137–142 of: STOC*. ACM.

Nipkow, Tobias. (2016). Automatic functional correctness proofs for functional search trees. *Pages 307–322 of:* Blanchette, J., & Merz, S. (eds), *Interactive Theorem Proving (ITP) 2016*, vol. 9807.

Nipkow, Tobias, & Dirix, Stefan. (2018). Weight-balanced trees. *Archive of formal proofs*, Mar. `http://isa-afp.org/entries/Weight_Balanced_Trees.html`, Formal proof development.

Nipkow, Tobias, Paulson, Lawrence C., & Wenzel, Markus. (2002). *Isabelle/HOL – A proof assistant for higher-order logic*. Lecture Notes in Computer Science, vol. 2283. Springer.

O'Connor, Liam, Chen, Zilin, Rizkallah, Christine, Amani, Sidney, Lim, Japheth, Murray, Toby, Nagashima, Yutaka, Sewell, Thomas, & Klein, Gerwin. 2016 (Sept.). Refinement through restraint: Bringing down the cost of verification. *International Conference on Functional Programming*.

Okasaki, Chris. (1999). *Purely functional data structures*. Cambridge University Press.

Okasaki, Chris, & Gill, Andrew. (1998). Fast mergeable integer maps. *Pages 77–86 of: Workshop on ML*.

Peyton Jones, Simon, Tolmach, Andrew, & Hoare, Tony. (2001). Playing by the rules: rewriting as a practical optimisation technique in GHC. *Haskell Workshop*.

Protzenko, Jonathan, Zinzindohoué, Jean-Karim, Rastogi, Aseem, Ramananandro, Tahina, Wang, Peng, Zanella-Béguelin, Santiago, Delignat-Lavaud, Antoine, Hriţcu, Cătălin, Bhargavan, Karthikeyan, Fournet, Cédric, & Swamy, Nikhil. (2017). Verified low-level programming embedded in F*. *Proc. ACM program. lang.*, **1**(ICFP), 17:1–17:29.

Ralston, Ryan. (2009). ACL2-certified AVL trees. *Pages 71–74 of: Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and Its Applications*. ACL2 '09. New York, NY, USA: ACM.

Rizkallah, Christine, Lim, Japheth, Nagashima, Yutaka, Sewell, Thomas, Chen, Zilin, O'Connor, Liam, Murray, Toby, Keller, Gabriele, & Klein, Gerwin. 2016 (Aug.). A framework for the automatic formal verification of refinement from Cogent to C. *International Conference on Interactive Theorem Proving*.

Spector-Zabusky, Antal, Breitner, Joachim, Rizkallah, Christine, & Weirich, Stephanie. (2018). Total Haskell is reasonable Coq. *Pages 14–27 of: CPP*. ACM.

Straka, Milan. (2010). The performance of the Haskell CONTAINERS package. *Pages 13–24 of: Proceedings of the Third ACM Haskell Symposium on Haskell*. Haskell '10. New York, NY, USA: ACM.

Swamy, Nikhil, Hriţcu, Cătălin, Keller, Chantal, Rastogi, Aseem, Delignat-Lavaud, Antoine, Forest, Simon, Bhargavan, Karthikeyan, Fournet, Cédric, Strub, Pierre-Yves, Kohlweiss, Markulf, Zinzindohoue, Jean-Karim, & Zanella-Béguelin, Santiago. (2016). Dependent types and multi-monadic effects in F*. *Pages 256–270 of: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. New York, NY, USA: ACM.

Vazou, Niki, Rondon, Patrick M., & Jhala, Ranjit. (2013). Abstract refinement types. *Pages 209–228 of: Proceedings of the 22nd European Conference on Programming Languages and Systems*. ESOP'13. Berlin, Heidelberg: Springer-Verlag.

Vazou, Niki, Seidel, Eric L., Jhala, Ranjit, Vytiniotis, Dimitrios, & Peyton-Jones, Simon. (2014). Refinement types for Haskell. *Pages 269–282 of: ICFP*. ACM.

Vazou, Niki, Lampropoulos, Leonidas, & Polakow, Jeff. (2017). A tale of two provers: Verifying monoidal string matching in Liquid Haskell and Coq. *Pages 63–74 of: Haskell Symposium*. ACM.

Vazou, Niki, Tondwalkar, Anish, Choudhury, Vikraman, Scott, Ryan G., Newton, Ryan R., Wadler, Philip, & Jhala, Ranjit. (2018). Refinement reflection: complete verification with SMT. *PACMPL*, **2**(POPL), 53:1–53:31.

Vytiniotis, Dimitrios, Peyton Jones, Simon, Claessen, Koen, & Rosén, Dan. (2013). HALO: Haskell to logic through denotational semantics. *Pages 431–442 of: POPL*. ACM.

Wiegley, John. (2017). *`coq-haskell`: A library for formalizing Haskell types and functions in Coq*. `https://github.com/jwiegley/coq-haskell`.