Ready, Set, Verify! Applying hs-to-coq to Real-World Haskell Code (Experience Report)

JOACHIM BREITNER, University of Pennsylvania, USA
ANTAL SPECTOR-ZABUSKY, University of Pennsylvania, USA
YAO LI, University of Pennsylvania, USA
CHRISTINE RIZKALLAH, University of New South Wales, Australia
JOHN WIEGLEY, BAE Systems, USA
STEPHANIE WEIRICH, University of Pennsylvania, USA

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.

CCS Concepts: • **Software and its engineering** → **Software verification**;

Additional Key Words and Phrases: Coq, Haskell, verification

ACM Reference Format:
Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, `Set`, Verify! Applying `hs-to-coq` to Real-World Haskell Code (Experience Report). *Proc. ACM Program. Lang.* 2, ICFP, Article 89 (September 2018), 16 pages. https://doi.org/10.1145/3236784

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. Our `hs-to-coq` tool [Spector-Zabusky et al. 2018] translates Haskell types,

Authors' addresses: Joachim Breitner, joachim@cis.upenn.edu, University of Pennsylvania, USA; Antal Spector-Zabusky, antals@cis.upenn.edu, University of Pennsylvania, USA; Yao Li, liyao@cis.upenn.edu, University of Pennsylvania, USA; Christine Rizkallah, c.rizkallah@unsw.edu.au, University of New South Wales, Australia; John Wiegley, john.wiegley@baesystems.com, BAE Systems, USA; Stephanie Weirich, sweirich@cis.upenn.edu, University of Pennsylvania, USA.

2475-1421/2018/9-ART89
https://doi.org/10.1145/3236784

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 finite 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. This work makes 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.
- As part of this work, we needed to extend `hs-to-coq` and its associated standard libraries to support our verification goal. 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 use testing to increase confidence in the `hs-to-coq` translation. 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.

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 6.

We would like to claim that this effort found bugs in `containers`, but this mature code was already functionally correct. Still, our efforts resulted in improvements to the `containers` 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] Furthermore, an extended version of this report is available [Breitner et al. 2018].

[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.
[2] https://github.com/antalsz/hs-to-coq.

```
data Set a = Bin  {-# UNPACK #-}  !Size !a !(Set a) !(Set a)
           | Tip
type Size  = Int
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[5]

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 depending on containers, it is the third-most used package in Hackage, after base and bytestring.[3]

The containers library is mature and highly optimized. It has existed for over a decade and has undergone many significant revisions for improving 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 of the uses are only of the map and set modules.[4] Moreover, the map modules are essentially analogues of the set modules. Therefore, we focus on Data.Set and Data.IntSet.

2.1 Weight-Balanced Trees and Big-Endian Patricia 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. 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

$$s_1 + s_2 \leq 1 \lor (s_1 \leq 3s_2 \land s_2 \leq 3s_1),$$

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 and 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 on the Bin data constructor instruct the compiler to unpack the size component, removing a level of indirection. The ! annotations indicate that all components should be strictly evaluated.

The Data.IntSet module also provides search trees; specifically, ones which have been specialized hold Ints in order to provide more efficient operations, especially union. This implementation

[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]From http://hackage.haskell.org/package/containers-0.5.11.0/docs/src/Data.Set.Internal.html#Set.
All code listings in this paper are manually reformatted and may omit module names from fully qualified names.

is based on big-endian Patricia trees, as proposed by Morrison [1968] and adapted to a functional setting by Okasaki and Gill [1998].

This data structure uses the bits of a stored value for deciding in which subtree the value should be placed. Instead of storing a single value at a leaf of the tree, this implementation improves performance by storing the membership information of consecutive numbers as a word-size bitmap.

2.2 A History of Performance Tuning

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`.[6] The code was later refactored by Straka [2010], who thoroughly evaluated and tweaked the library for performance.[7] 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[8] based on performance measurements.

The module `Data.IntSet` (and `Data.IntMap`) has been around even longer. Okasaki and Gill mention in their 1998 paper that GHC had already made use of `IntSet` and `IntMap` for several years. In 2011, the `Data.IntSet` module was re-written to use machine-words as bitmaps in the leaves of the tree.[9] This moved the `containers` library further away from the literature on Patricia trees and introduced a fair number of low-level bit twiddling operations.

3 OVERVIEW OF OUR VERIFICATION APPROACH

In order to verify `Set` and `IntSet`, 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 from the implementation of `Set` in Figure 1. The `hs-to-coq` tool translates this input to the following Coq definitions.[10] The strictness and unpacking annotations are ignored, as they do not make sense in Coq, and 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.
```

[6] https://github.com/haskell/containers/commit/bbbba97c
[7] https://github.com/haskell/containers/commit/3535fcbe
[8] https://github.com/haskell/containers/commit/c3083cfc
[9] https://github.com/haskell/containers/pull/3
[10] In the file examples/containers/lib/Data/Set/Internal.v

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

In this way, we use hs-to-coq to translate Set and IntSet into Gallina code which we then verify. Section 4 details the properties that we prove about the two data structures.

To test the Haskell to Coq translation, we use 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. By providing an interface that restores the type-class based types, we can run the original containers test suite against this code. This process allows us to check that hs-to-coq preserves the semantics of the original Haskell program during translation.

The set modules of the containers library contain 149 functions and 22 type class instances, written in 2207 lines of code (excluding comments and blank lines). Out of these, 15 functions and 11 type class instances (270 loc) were deemed "out of scope" and not translated.[11] Our translation produces 2587 lines of Gallina code.

The Set and IntSet data structures come with extensive APIs. We specify and verify a representative subset of commonly used functions covering 68% of the Set API and 49% of the IntSet API. This verified API is complete enough to instantiate Coq's specification of finite sets, 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 1234 lines of Haskell; the verification of Set and IntSet required 8.6 lines of proof per lines of code. This factor is noticeably higher for IntSet (10.0×) 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 sortedness (1489 loc); dyadic intervals, which are used for verifying IntSet (1169 loc); and support for working with lawful Ord instances, including a complete decision procedure (453 loc).

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. But how can we ensure that our specifications of Set and IntSet are two-sided and rich? How do we know that the specifications are not just facts that happen to be true, but are actually 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, as described below. This way, we also ensure that our specifications are rich because they describe the data structures at varying levels of abstraction. Furthermore, by using multiple disparate specifications, we not

[11] More explanation of these choices and further statistics can be found in the extended version [Breitner et al. 2018].

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.

4.1 Implementation Invariants

`Set` and `IntSet` are two examples of abstract types whose correctness depends on invariants. Therefore, we define well-formedness predicates `WF: Set_ e -> Prop` and `WF : IntSet -> Prop` and show that the operations preserve these properties.

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 user-facing API of `containers` (since all exported functions preserve well-formedness), but it 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 `WF` as an inductive predicate, because we find it more useful from a proof engineering perspective. To be sure that our predicate is correct, we 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. In this case, our predicate is stronger than the corresponding `valid` function from the implementation – the Haskell version was missing some necessary conditions. We reported this to the library authors,[12] who have since fixed `valid`.

This fix is an example of using verification to cross-validate specifications. Here, the comments were correct and the code was wrong. However, sometimes we discover that it is the comments 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 the extended version [Breitner et al. 2018].

4.2 Property-based Testing

Next, we show that the implementations of `Set` and `IntSet` are correct according to the implementors of the module. In other words, we specify correctness by deriving a specification directly from the property-based test suite that is distributed with the `containers` library.

This test suite contains many properties expressed using QuickCheck [Claessen and Hughes 2000]. For example, one such property states that the `union` operation is associative:

```
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`:[13]

```
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 5.5, 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 proven that all QuickCheck

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

[13] In the file examples/containers/theories/IntSetPropertyProofs.v

properties about verified `IntSet` functions are theorems (with one exception due to our choice of integer representation – see Section 5.3).

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:[14]

> 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.

In practice, it makes no difference whether the `Int`s used to store sizes are bounded or not, as a set with $(2^{63} - 1)$ elements would require at least 368 exabytes of storage. However, using fixed-width integers would 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 5.3.)

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 is *rewrite rules* [Peyton Jones et al. 2001]. 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 its functions up 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.[15]

[14] http://hackage.haskell.org/package/containers-0.5.11.0/docs/Data-Set.html
[15] In the files examples/containers/theories/SetProofs.v and examples/containers/theories/IntSetProofs.v

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`, and `MonoidLaws`. These classes can only be instantiated if the corresponding instance is lawful.

However, 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`." 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"? To make sure that we can make `Set_` with `(<>)` as `union` an instance of the `MonoidLaws` type class, we require that the two expressions be equal according to their `Eq` instance:`(x <> (y <> z) == (x <> y) <> z) = true`. Alternatively, we could parameterize the `MonoidLaws` class by a separate (potentially undecidable) equality predicate, which would allow us to prove the `MonoidLaws` for the function instance `MonoidLaws b -> MonoidLaws (a -> b)`.
(2) What do we mean by "For all elements of `T`"? The obvious choice is universal quantification over *all* elements of `T`. But this approach again is insufficient for `Set`: the `union` operation 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. Alternatively, we could constrain `MonoidLaws`'s theorems to hold only on members of `T` that are well-formed in some general way.

In both cases, we have designed our specifications to be only as complex as necessary for the verification of the `Set_` and `IntSet` data structures. In future work, we may generalize these classes to accommodate more structures.

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[16] provides module types that cover many common operations and their properties. 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, while `Set_` requires its element type to be an instance of the `Ord` type class. We solve this impedance mismatch with a module that generates an `Ord` instance from the appropriate `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.

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

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 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.)

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 as code written directly in Coq – and for textbook-level examples, this 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 Haskell types should be mapped 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 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. This 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.

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`. 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 capture the semantics of `ptrEq` more faithfully by making the definition of `ptrEq` *opaque* and partially specifing its behavior.[17]

```
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 hides this implementation of `ptrEq`. While the specification `ptrEq_eq` is vacuously true, the opaque definition forces verification to proceed down both paths.

We must still 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 Partiality in Total Functions

Some functions on `Set`s 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, but be total for well-formed `Set`s.

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`. 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.

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

There is one caveat to this technique: we remove the first argument to Haskell's `seq` function during translation, as it cannot affect the value of the expression (`seq` is only used to control evaluation order). Therefore, we cannot reason about errors that occur in this position.

This encoding of partiality 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.

5.3 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 Coq's binary representation of natural numbers (`N`) or a fixed width integer type.

In this project, we choose to use `N` as the element type for `IntSet`. This choice is suitable for applications that do not wish to reason about overflow (although overflow is not an issue for the library itself). This simplifies our proofs, as Coq's standard library provides a fairly comprehensive library of lemmas about `N` and decision procedures (`omega` and `lia`) that work with it.

Using `N` also required making `hs-to-coq` more powerful. Although the `N` is a binary representation of natural numbers, the `IntSet` code uses a few 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* such as

```
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.

We can translate most of the highly tuned bit-twiddling algorithms used by `IntSet`. However, there are some that are beyond the current capabilities of `hs-to-coq`. For example, the function `indexOfTheOnlyBit`, which was contributed by Edward Kmett,[18] takes a number with exactly one bit set and calculates the index of said bit. It does so by unboxing 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. We currently cannot translate this code because `hs-to-coq` does not yet support unboxed arrays or unboxed integers. We therefore replace it with a simple definition based on a integer logarithm function provided by Coq's standard library. Similarly, we provide simpler definitions for the low-level bit-twiddling functions `branchMask`, `mask`, `zero` and `suffixBitMask`.

5.4 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 approaches.

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

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.

Well-founded recursion. Another common recursion pattern can be found in binary operations such as `link` in `Set`. In this function, every recursive call shrinks *either or both* of its arguments to immediate subterms of the originals, leaving the others unchanged, and is beyond the capabilities of Coq's termination checker. Other functions recurse in a non-structural way; for example, the `foldlBits` on `IntSets` recurses on the input after clearing its least-significant set bit. We therefore use edits to instruct `hs-to-coq` to use Coq's **`Program Fixpoint`** command to translate these functions in terms of well-founded recursion.

Coq's **`Program Fixpoint`** only supports top-level functions, but we frequently encounter local recursive functions. Therefore, we needed to extend `hs-to-coq` to translate local recursive functions using the same well-founded-recursion-based fixed-point combinator as **`Program Fixpoint`**.

Deferred recursion. Finally, we encounter some functions that require elaborate termination arguments, such as `fromDistinctAscList` in `Set`. This function uses two local recursive functions, `go` and `create`, that require subtle reasoning about their termination behavior.

For hard cases like these, we added *deferred termination checking* to `hs-to-coq`. This translation allows us to defer all reasoning about the function until after it is defined. To defer this reasoning, we use the following axiom as a 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 5.2). 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 we verify `fromDistinctAscList`, we need to reason about `deferredFix`. 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.

These axioms are consistent with Coq. We can implement `deferredFix` and `deferredFix_eq_on` in terms of classical logic and the axiom of choice (as provided by the Coq standard library module `Coq.Logic.Epsilon`).[19] 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 extensional approach to defining recursive functions was inspired by Isabelle's function package [Krauss 2006]. It is also an instance of the recursion schemes described by Charguéraud [2010].

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

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.

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 QuickCheck's computational `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.

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

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:

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

6 THE FORMALIZATION GAP

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?

As always, when a theorem is stated about an object that is not purely mathematical, its validity depends on a number of assumptions. First, we have to assume that Coq behaves as documented and is consistent. This is particularly relevant because we rely on fine details of Coq's machinery (e.g., opacity, Section 5.2) and optionally add consistent axioms (see Section 5.4).

The biggest assumption is that the semantics of a Gallina program 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 is no formal semantics for Haskell. We know that “Fast and Loose Reasoning is Morally Correct” [Danielsson et al. 2006], which says that theorems about the total fragment of a non-total language carry over to the full language. But since we relate two very different languages, this argument alone is not enough to bridge the gap.

Similarly, we rely on hs-to-coq translating Haskell code into the correct Gallina code. The translator itself is a sizable piece of code, which is unverified (not least because there is no formal semantics of Haskell). We get some confidence in it by manually inspecting its output and observing that it is indeed what we would consider the “right” translation from Haskell into Gallina, and gain more 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 (Section 3) also stress-tests the translation.

Finally, the translation was not completely automatic and required manual edits. With each edit, we add another assumption (for example, that our underspecification of pointer equality matches the actual behavior of GHC). We list and justify our manual interventions in Section 5.

The formalization gap in 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, this is less critical. Even if one of our assumptions is flawed, it is much more likely that the flaw will stop us from completing the proofs, rather than allowing us to conclude the proofs without noticing a bug. Incomplete proofs can uncover bugs, too.

7 RELATED WORK

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.

Similarly to hs-to-coq, LiquidHaskell [Vazou et al. 2014] can be used to verify existing Haskell code. LiquidHaskell allows users to annotate their code with refinement types, and then it discharges the proof obligations described by these types using SMT solvers. Vazou et al. [2017] compared the experience of using LiquidHaskell vs. that of using plain Coq, and found that both have advantages.

Vazou et al. [2013] used LiquidHaskell to verify Data.Map. This code shares the same underlying data structure (weight-balanced trees) to Data.Set. Similar to 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 verify, 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. Although it might be possible to replicate our specifications by relying on theories of finite sets and maps in SMT [Kröning et al. 2009] using refinement types in LiquidHaskell, this approach has not been explored.

Furthermore, our specification includes type class laws, and we 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 alowed us to reason about `fromDistinctAscList` (see Section 5.4) and prove that it is indeed terminating; on the other hand, Vazou et al. deactivate the termination check for this function.

Outside of Haskell, Hirai and Yamamoto [2011] implemented a weight-balanced tree similar to Haskell's `Data.Set` library (albeit using the balancing condition of Nievergelt and Reingold [1972]) and verified its balancing properties in Coq. Nipkow and Dirix [2018] extended this work by formalizing similar trees in Isabelle and verifying the functional correctness of `insert` and `delete`.

Hirai and Yamamoto defined the `union`, `intersection`, and `difference` functions based on the "hedge union" algorithm, but `containers` has since changed to use "divide and conquer". Moreover, Hirai and 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. The extended version [Breitner et al. 2018] also describes the new insights about the balancing conditions that we gained through our verification effort.

8 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. We have also started to verify `Data.Map` and `Data.IntMap`. Because these modules use the same algorithms as `Data.Set` and `Data.IntSet`, we already adapted some of our existing proofs to this setting.

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

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

Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017). DOI: http://dx.doi.org/10.1098/rsta.2016.0331

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda - A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, 73–78. DOI: http://dx.doi.org/10.1007/978-3-642-03359-9_6

Edwin Brady. 2017. *Type-driven Development With Idris*. Manning.

Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, `Set`, Verify! Applying `hs-to-coq` to real-world Haskell code. *CoRR* abs/1803.06960 (2018). arXiv:1803.06960 http://arxiv.org/abs/1803.06960

Arthur Charguéraud. 2010. The Optimal Fixed Point Combinator. In *Proceedings of the First International Conference on Interactive Theorem Proving (ITP'10)*. Springer-Verlag, Berlin, Heidelberg, 195–210. DOI: http://dx.doi.org/10.1007/978-3-642-14052-5_15

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

Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. 2006. Fast and loose reasoning is morally correct. In *POPL*. ACM, 206–217. DOI: http://dx.doi.org/10.1145/1111037.1111056

Yoichi Hirai and Kazuhiko Yamamoto. 2011. Balancing weight-balanced trees. *Journal of Functional Programming* 21, 3 (2011), 287–307. DOI: http://dx.doi.org/10.1017/S0956796811000104

Alexander Krauss. 2006. Partial Recursive Functions in Higher-Order Logic. In *IJCAR (LNCS)*, Vol. 4130. Springer, 589–603.

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

The Coq development team. 2016. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr Version 8.6.1.

Donald R. Morrison. 1968. PATRICIA&Mdash;Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (Oct. 1968), 514–534. DOI: http://dx.doi.org/10.1145/321479.321481

Jürg Nievergelt and Edward M. Reingold. 1972. Binary Search Trees of Bounded Balance. In *STOC*. ACM, 137–142. DOI: http://dx.doi.org/10.1145/800152.804906

Tobias Nipkow and Stefan Dirix. 2018. Weight-Balanced Trees. *Archive of Formal Proofs* (March 2018). http://isa-afp.org/entries/Weight_Balanced_Trees.html, Formal proof development.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer. DOI: http://dx.doi.org/10.1007/3-540-45949-9

Chris Okasaki. 1999. *Purely functional data structures*. Cambridge University Press. DOI: http://dx.doi.org/10.1017/CBO9780511530104

Chris Okasaki and Andrew Gill. 1998. Fast Mergeable Integer Maps. In *In Workshop on ML*. 77–86.

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

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is reasonable Coq. In *CPP*. ACM, 14–27. DOI: http://dx.doi.org/10.1145/3167092

Milan Straka. 2010. The Performance of the Haskell Containers Package. In *Proceedings of the Third ACM Haskell Symposium on Haskell (Haskell '10)*. ACM, New York, NY, USA, 13–24. DOI: http://dx.doi.org/10.1145/1863523.1863526

Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. 2017. A Tale of Two Provers: Verifying Monoidal String Matching in Liquid Haskell and Coq. In *Haskell Symposium*. ACM, 63–74. DOI: http://dx.doi.org/10.1145/3122955.3122963

Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems (ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 209–228. DOI: http://dx.doi.org/10.1007/978-3-642-37036-6_13

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *ICFP*. ACM, 269–282. DOI: http://dx.doi.org/10.1145/2628136.2628161

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement reflection: complete verification with SMT. *PACMPL* 2, POPL (2018), 53:1–53:31. DOI: http://dx.doi.org/10.1145/3158141