Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell

YIYUN LIU, University of Maryland, College Park, USA
JAMES PARKER, University of Maryland, College Park, USA
PATRICK REDMOND, University of California, Santa Cruz, USA
LINDSEY KUPER, University of California, Santa Cruz, USA
MICHAEL HICKS, University of Maryland, College Park, USA
NIKI VAZOU, IMDEA, Madrid, Spain

This paper presents an extension to Liquid Haskell that facilitates stating and semi-automatically proving properties of typeclasses. Liquid Haskell augments Haskell with *refinement types*—our work allows such types to be attached to typeclass method declarations, and ensures that instance implementations respect these types. The engineering of this extension is a modular interaction between GHC, the Glasgow Haskell Compiler, and Liquid Haskell's core proof infrastructure. The design sheds light on the interplay between modular proofs and typeclass resolution, which in Haskell is coherent by default, but in other dependently typed languages is not.

We demonstrate the utility of our extension by using Liquid Haskell to modularly verify that 34 instances satisfy the laws of five standard typeclasses. More substantially, we implement a framework for programming distributed applications based on *conflict-free replicated data types* (CRDTs). We define a typeclass whose Liquid Haskell type captures the mathematical properties CRDTs should satisfy; prove in Liquid Haskell that these properties are sufficient to ensure that replicas' states converge despite out-of-order update delivery; implement (and prove correct) several instances of our CRDT typeclass; and use them to build two realistic applications, a multi-user calendar event planner and a collaborative text editor.

Additional Key Words and Phrases: replicated data types, CRDTs, typeclasses, refinement types, Liquid Haskell

1 INTRODUCTION

Liquid Haskell [Rondon et al. 2008; Vazou et al. 2014] is an extension to Haskell that enables formally verifying logical properties of Haskell programs. Its basis for doing so is *refinement types*, which augment standard Haskell types with predicates that restrict the set of valid values [Rushby et al. 1998; Xi and Pfenning 1998; Constable and Smith 1987]; these predicates are checked automatically by an SMT solver. Liquid Haskell uses a mechanism called *refinement reflection* [Vazou et al. 2017] to lift the ability to state and check single refinements of individual functions to state and prove general properties of entire codebases [Vazou et al. 2018].

Liquid Haskell's features—being based on a widely used, industrial-strength programming language and having built-in support for proof automation—make it appealing as a platform for real-world, verified software development. Indeed, we were motivated to use it to build a platform for distributed computing applications employing data replication to increase reliability and availability. These applications are hard to get right, and we hoped Liquid Haskell's formal verification capabilities could help.

Unfortunately, there was a big stumbling block: Liquid Haskell cannot verify properties of *typeclasses* [Wadler and Blott 1989], which are used extensively throughout the Haskell ecosystem. A typeclass *definition* specifies a type constructor and a collection of method declarations over that type. A typeclass *instance* defines an implementation of that constructor and those methods. For

Authors' addresses: Yiyun Liu, University of Maryland, College Park, USA; James Parker, University of Maryland, College Park, USA; Patrick Redmond, University of California, Santa Cruz, USA; Lindsey Kuper, University of California, Santa Cruz, USA; Michael Hicks, University of Maryland, College Park, USA; Niki Vazou, IMDEA, Madrid, Spain.

example, the `Ord` typeclass from Haskell's standard library declares that its instances `a` must have a method `(<=)` of type `a → a → bool`; numbers, strings, booleans, and many other types are instances of `Ord`. The standard `sort` function can only sort lists of types that are `Ord` instances, since it needs a comparison function; this requirement is expressed as a constraint on `sort`'s type, `Ord a ⇒ [a] → [a]`.

Typeclass refinements for Liquid Haskell. The primary contribution of this paper is an extension to Liquid Haskell that supports stating and proving properties of typeclasses (Section 2). While it was previously possible in Liquid Haskell to prove properties of individual instances of a typeclass, it was not possible to give refinement types to a typeclass definition's methods. As such, Liquid Haskell code and proofs could not then modularly *use* those types when invoking methods from functions whose arguments (like `sort`) have a typeclass constraint. Given the ubiquity of typeclasses in Haskell code, the ability to do this is key to being able to verify interesting properties of real-world Haskell applications.

Implementing typeclass refinements in Liquid Haskell was not straightforward. Its implementation works by verifying properties not of Haskell source code, but rather of *Core* expressions, which are the intermediate representation produced by the Glasgow Haskell Compiler (GHC) [GHC 2020], the de facto Haskell standard. Doing so leverages functionality that GHC already provides (e.g., typechecking and elaboration) and allows Liquid Haskell to evolve semi-independently from GHC, since Core's definition is relatively stable. But there is a problem: typeclasses are not Core expressions—during elaboration, GHC translates them to *dictionaries*, which are basically records of functions. Code that defines a typeclass instance is translated to create a dictionary, and code that expresses a typeclass constraint is translated to use a dictionary; e.g., `sort` will be translated to be passed an `Ord` dictionary, from which it invokes the `(<=)` method. To maintain the current separation between Liquid Haskell and GHC, our implementation (Section 3) transliterates typeclass methods' refinement types to checked invariants over dictionaries, so refinement types on typeclasses are verified when dictionaries are created, and those types can be used by client code. To do this modularly we had to expand the way Liquid Haskell interacts with GHC.

While Liquid Haskell is not the first proof system with typeclass support—Coq, Isabelle, Idris, F*, Agda, and Lean have typeclasses or something like them—our approach represents an interesting point in the design space (see Section 5.2). In particular, our modular approach reuses Haskell's typeclass resolution procedure, which limits typeclass type parameters to normal Haskell types. But, Haskell's resolution is *coherent* by default (it always chooses the same typeclass instance for a given type) [Bottu et al. 2019] and this fact is very useful for some proofs. Our implementation introduces a *checked invariant* during elaboration to express coherence, which is sound even if coherent resolution is overridden by GHC pragma (in which case proof of the invariant could fail at instance creation time). Other systems may allow instance types to be more general, but the cost is a more involved resolution procedure which may be neither coherent nor terminating, complicating its use in programming and proofs.

Case study: Verifying standard typeclass laws. As a simple test of the utility of typeclass refinements, we carried out a small case study: We used Liquid Haskell to verify that instances of standard Haskell typeclasses satisfy the expected typeclass laws (Section 2.3). Significant prior work has focused on this application specifically, employing a variety of techniques, including random testing, term rewriting, contract verification, and conversion to Coq (see Section 5.1). Liquid Haskell typeclass refinements offer a natural, general-purpose approach. In particular, laws can be expressed as refinements to methods of a subclass of the target typeclass, and proofs of them are carried out in a subclass of each implementation of that target typeclass. This approach permits proofs of existing Haskell code without requiring that code be directly modified or annotated. We demonstrate this for several standard typeclasses, including `Semigroup`, `Monoid`, `Functor`, `Applicative`, and `Monad`, proving 34 instantiations satisfy their laws, in all (Section 2.3). Mostly, we find that the proofs are short (just a couple of lines), thanks to Liquid Haskell's SMT automation, and proof checking time is fast (typically a few seconds).

Case study: A platform for programming with verified replicated data types. Spurred by the success of this case study, we set out to build a platform for programming distributed applications based on *replicated data types* (RDTs) [Shapiro et al. 2011b,a; Roh et al. 2011; Oster et al. 2006; Preguica et al. 2009; Weiss et al. 2009; Kleppmann and Beresford 2017] (Section 4). Replication is ubiquitous in distributed systems to guard against machine failures and keep data physically close to clients who need it, but it introduces the problem of keeping replicas consistent with one another in the face of network partitions and unpredictable message latency. RDTs are data structures whose operations must satisfy certain mathematical properties that can be leveraged to ensure *strong convergence* [Shapiro et al. 2011b], meaning that replicas are guaranteed to have equivalent state given that they have received and applied the same *unordered* set of update operations.

Liquid Haskell typeclasses provide a natural, modular, and elegant way to implement and verify RDTs. We define a typeclass `VRDT` with a refinement type that captures the necessary properties, and we use Liquid Haskell to prove that those properties hold for a several primitive instances. We also defined several larger `VRDT` instances by modularly combining both the code and proofs of smaller ones. We state and prove, in Liquid Haskell, the strong convergence property that `VRDT` instances enjoy. Pleasantly, our approach generalizes and relaxes the typical assumption of *causal message delivery*. Our `VRDT` instances are sufficiently expressive that with them we were able to build a shared calendar event planner, and also a collaborative text editor, though the latter relies on a `VRDT` we have not yet fully verified, but expect to. Each application is implemented using a few hundred lines of Haskell code (Section 4.4). Although there exists previous work on mechanized verification of RDTs (Section 5.3), our work is, to our knowledge, the first to use a solver-aided language (Liquid Haskell or otherwise) to implement verified RDTs. Because Liquid Haskell is an extension of standard Haskell, our applications are real, running Haskell applications, but now with mechanically verified RDT implementations.

Contributions. In summary, this paper makes the following contributions:

- We present an extension to Liquid Haskell that supports stating and proving refinements of typeclass methods' types. The engineering of this extension is an interesting interaction between GHC and Liquid Haskell's core proof infrastructure, and our design sheds light on the interplay between coherent typeclass resolution and modular proofs (Sections 2 and 3).
- We use our extension to Liquid Haskell to modularly verify that 34 standard instances satisfy the laws of five widely-used Haskell typeclasses, the `Semigroup`, `Monoid`, `Functor`, `Applicative`, and `Monad` typeclasses (Section 2.3).
- We further use our extension to Liquid Haskell to implement a platform for distributed applications based on replicated data types. We define a typeclass whose Liquid Haskell type captures the mathematical properties that must be true of RDTs, prove in Liquid Haskell that strong convergence does indeed hold if these properties are satisfied, and implement (and prove correct) several instances of our refined typeclass. Using these instances we implement two realistic applications: a shared calendar event planner and a (partially verified) collaborative text editor (Section 4).

We are working with the Liquid Haskell maintainers to integrate our extension into the main implementation, at which point it will be freely available.

```
head :: {xs:[a] | length xs > 0} → a
head (h:_) = h

(++) :: xs:[a] → ys:[a] → { v:[a] | length v == length xs + length ys }
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)

lAssoc :: x:[a] → y:[a] → z:[a] → { x ++ (y ++ z) == (x ++ y) ++ z }
lAssoc []    _ _ = ()
lAssoc (_:x) y z = lAssoc x y z
```

Fig. 1. Haskell's list `head` and append `(++)` functions augmented with refinement types to capture pre- and post-conditions; and `lAssoc`, a statement and proof that append is associative.

2 TYPECLASSES IN LIQUID HASKELL

This section begins with background on Liquid Haskell [Rondon et al. 2008; Vazou et al. 2014], which extends Haskell with refinement types (Section 2.1). Then it presents our extension to Liquid Haskell, which permits annotating a typeclass definition's methods with refinement types, thus allowing a typeclass's clients to assume those richer types, while obligating a typeclass's instances to implement them (Section 2.2). As a demonstration of the effectiveness of this approach, we verify that 34 instances of 5 standard typeclasses satisfy the expected laws (Section 2.3).

2.1 Refinement Types and Liquid Haskell

A *refinement type* augments a base type T with a predicate ϕ that restricts the set of valid values [Rushby et al. 1998; Xi and Pfenning 1998; Constable and Smith 1987]. In Liquid Haskell, a refinement type has the form `{` x:T `|` ϕ `}`—the base type T is refined according to predicate ϕ, which may refer to values of the base type via the variable x (if it appears free in ϕ). For example, a refinement type for positive integers would be `{ x:int | x > 0 }`, i.e., ϕ = `x > 0`.

In Figure 1, the function `head` uses this kind of refinement on the type of its input list `xs`, stating the *precondition* that the list's length be positive. This refinement thus prevents calling `head` with a possibly empty list, thus precluding the exception that could otherwise result.

Also in the figure we see code for Haskell's standard list append operator, `(++)`, which uses a refinement to state a *postcondition*. The (standard) code states that appending an empty list `[]` with a list `ys` yields `ys` (line 2), while appending a non-empty list (with a head element `x` and a tail `xs`) with a list `ys` is the result of cons'ing `x` to the front of `xs` appended to `ys` (line 3). The refinement type states that the output list's length is equal to the sum of the lengths of the input lists. The refinement type predicate is able to refer to the function's inputs via names `xs` and `ys`, which annotate the parameters' types. Liquid Haskell proves that postconditions such the one on `(++)` hold by generating appropriate verification conditions from the code and delegating to an SMT solver (in particular, Z3 [De Moura and Bjørner 2008]); we say more on this below.

In the refinement types of `head` and `(++)`, `length` refers to the Haskell `length` function on lists. Such references to normal language terms are lifted into the refinement logic through a process called *refinement reflection* [Vazou et al. 2017]. Refinement reflection uses the definitions of Haskell's functions to generate singleton refinement types that precisely describe the result of the function. To ensure soundness of type checking, only provably terminating functions can be reflected.

Refinement reflection makes it possible to write and mechanically verify proofs of independent, general properties, e.g., involving many functions and not just a single one. These are called

```
class Semigroup a where
  (<>) :: a → a → a


instance Semigroup [a] where
  (<>) = (++)
```

(a) Standard `Semigroup` typeclass and the list instance of it

```
class Semigroup a ⇒ VSemigroup a where
  lawAssociativity :: x:a → y:a → z:a →
      {x <> (y <> z) == (x <> y) <> z}

instance Semigroup [a] ⇒ VSemigroup [a] where
  lawAssociativity = lAssoc
```

(b) `VSemigroup` extends `Semigroup` with an associativity law, which its list instance satisfies via `lAssoc`

Fig. 2. Typeclasses with Refinement Types

extrinsic properties, as they are written externally to any particular function's definition, as opposed to *intrinsic properties* like the ones on `head` and `(++)`. For example, `lAssoc` in Figure 1 is an extrinsic property (and proof) that append is associative. The property is the type, which states that for all lists `x`, `y`, `z` we have that `x ++ (y ++ z)` equals `(x ++ y) ++ z`. Note that the postcondition of `lAssoc` is equivalent to `{ v:unit | x ++ (y ++ z) == (x ++ y) ++ z }`—the `v:unit` part is dropped since there is no need to name the result, which is not mentioned in the predicate.

Proofs of extrinsic properties are themselves Liquid Haskell definitions whose type is the desired property. The proof in our example is the body of `lAssoc`, which expresses that the property holds by induction—the base case is for `[]` and the recursive case is for `(_:x) y z`. In the base case, there is nothing specific the programmer has to write other than `()`, the "return type" of the property. For the recursive case, the inductive argument occurs by referring to the property on the strictly smaller input `x y z` (rather than `(_:x) y z`). This proof follows a standard formula [Vazou et al. 2018] in which the handwritten part shown here provides the structure, and the proof details are filled in using a combination of *PLE* (Proof by Logical Evaluation) [Vazou et al. 2017; Leino and Pit-Claudel 2016], which automates function unfolding, and SMT solving, which automates reasoning over specific theories (e.g., equality and linear arithmetic). Both strategies preserve decidable type checking. Such automation helps make it possible to write substantial proofs in Liquid Haskell, as previously demonstrated with a proof of noninterference for the LWeb system [Parker et al. 2019]. Proofs can also be done by hand, as needed/desired [Vazou et al. 2018].

Liquid Haskell's implementation is simplified by making use of GHC, the Glasgow Haskell Compiler [GHC 2020], to partially evaluate programs. Liquid Haskell first parses refinement types, which are written in the comments of normal Haskell code. Then it passes the Haskell code to GHC, and receives back the code as *Core*, which is GHC's simplified intermediate language. Liquid Haskell lifts the Core output into the refinement logic using refinement reflection. Finally, it converts the refinement types and corresponding Core output into SMT-LIB2 queries [Barrett et al. 2010] which can automatically be verified by Z3. If any queries are invalid, Liquid Haskell reports an error message to the user.

2.2 Refinement Types for Typeclasses

We have extended Liquid Haskell to allow typeclass methods to be annotated with refinement types. Doing so allows a developer to state properties that a typeclass's methods should always satisfy. Clients of that typeclass can thus assume those properties in their own proofs. Of course, implementors of the typeclass's instances must prove those properties hold for their instance.

Laws as Refinement Types. We illustrate the utility of our extension by showing how standard typeclass *laws* can be encoded as refinement types. Laws are properties that clients of a typeclass generally assume, and that implementors of a typeclass are supposed to ensure. Of course, without something like our extension, there is no guarantee that they do so.

Figure 2(a) shows the `Semigroup` typeclass, which defines a type `a` that is equipped with a single operator `<>`. One particular implementation of this typeclass for lists (`[a]`) is also shown, where `<>` corresponds to the List append operator. A key law of semigroups is that their operator is associative. Clients of `Semigroup` may assume this law holds of any instance they are given; they may break if it does not. Fortunately, as we proved in the previous subsection, List append is associative, so the List instance of `Semigroup` satisfies the law. How can we show this?

We extend the syntax of typeclasses to allow for refinement types on method declarations. Below is a version of `Semigroup` extended to capture the associativity typeclass law as a refinement type.

```
class VSemigroup a where
  (<>)             :: a → a → a
  lawAssociativity :: x:a → y:a → z:a → {x <> (y <> z) == (x <> y) <> z}
```

`VSemigroup` matches the definition of `Semigroup` from Figure 2(a) but adds typeclass method `lawAssociativity`, which (extrinsically) defines the associativity property. All `VSemigroup` instances are now required to define `lawAssociativity` and provide an explicit associativity proof. The lower portion of Figure 2(b) implements the list instance of `VSemigroup` by extending `Semigroup` list instance and providing the associativity proof `lAssoc` from Section 2.1.

Using the Laws, Modularly. By allowing refinement types on typeclass definitions, we extend the modularity benefits of typeclasses from code to proofs. In particular, clients of a refined typeclass can take advantage of its stated refinement types when conducting their own proofs. For example, below we express and prove an extrinsic property that extends associativity to four elements.

```
assoc2 :: VSemigroup a ⇒ x:a → y:a → z:a → w:a
       → { x <> (y <> (z <> w)) == ((x <> y) <> z) <> w }
assoc2 x y z w = lawAssociativity x y (z <> w)
                 `const` lawAssociativity (x <> y) z w
```

The proof is a consequence of `lawAssociativity`, which is applied twice, combined with Haskell's `const`ant function. The proof is carried out once, independent of any `VSemigroup` instance, but the property holds for all of them.

The code of our VRDT case study (Section 4) is set up similarly. We define a `VRDT` typeclass with operations on data that enjoy particular properties. Relying on these properties, we can prove `strongConvergence` of all VRDTs; this property essentially states that two replicas that start in the same state will end up in the same state if they apply the same operations, in any order.

Refinements in Subclasses. For improved modularity, our extension allows typeclass method refinements to refer to superclass methods. For example, another way to write `VSemigroup` is shown at the top of Figure 2(b), which literally extends `Semigroup` with the added method. Defining properties in subclasses is particularly useful when not wanting to modify typeclasses in other packages (including those in normal, not Liquid, Haskell). It can also be useful when not wanting to necessarily require implementations to prove all possible properties; different subsets of properties of interest can be defined in different subclasses.

Haskell typeclasses can have multiple superclasses, which allows defining a typeclass containing properties of data structures that implement multiple typeclasses. For example, consider the `Monoid` typeclass, which extends `Semigroup` to also include the `mempty` identity element. Since a particular data structure (like a list) can implement both typeclasses, we could define the verified typeclass `VMonoid` that extends `VSemigroup` and `Monoid` with two laws.

```
class (VSemigroup a, Monoid a) ⇒ VMonoid a where
```

```
lawEmpty   :: x:a → { x <> mempty == x && mempty <> x == x }
lawMconcat :: xs:[a] → { mconcat xs == foldr (<>) mempty xs }
```

That `mempty` is an identity for `<>` is encoded in the `lawEmpty` method; it refers to `(<>)`, which is defined in the `VSemigroup` parent typeclass. The law `lawMconcat` guarantees that `mconcat`, defined by `Monoid`, is equivalent to folding over a non-empty list with `(<>)`.

We can also define verified components from other verified components, where proofs of the former's properties can depend on properties that hold of the latter. For example, in our VRDT case study, we define a VRDT `TwoPMap` in terms of any other VRDT; here is the beginning of the instance definition:

```
instance (Ord k, VRDT v) ⇒ VRDT (TwoPMap k v) where ...
```

The proofs of `TwoPMap k v`'s properties make use of the properties that hold for `Ord k` and `VRDT v`.

Coherence. There is an interesting twist in our `VMonoid` example. As mentioned, `Monoid` extends `Semigroup`; as such, proofs of properties in `VMonoid` may wish to assume that the `VSemigroup` instance resolved for `VMonoid` has the *same* parent superclass as that of the resolved `Monoid` instance. Indeed, this assumption is critical for these properties: we require that the `<>` operator in both `Monoid` and `VSemigroup` to be literally the same function. Such an assumption is reasonable because Haskell's typeclass resolution procedure is *coherent* by default—there can always be only one possible typeclass instance at a particular type. While coherence solves the the "diamond problem" [Stroustrup 1989], it is possible for programmers to override coherence via the `INCOHERENT` GHC pragma. In this case, we must take care that proofs of or using refinements do not assume coherence holds. We say more in Section 3 about how our system internally reasons about coherence to ensure soundness and precise reasoning.

Limitation: No Refined Instances. A limitation of our approach is that typeclass instances cannot be defined for refined types, only for base types. For example, we cannot have distinct semigroup instances for positive and negative numbers, i.e., `instance VSemigroup { v:Int | 0 < v }` and `instance VSemigroup { v:Int | v < 0 }`. But this limitation confers the benefit that we can reuse GHC's typeclass resolution procedure in our implementation, and proofs can take advantage of the fact that resolution is coherent. We say more in Section 3.3.

2.3 Verifying Laws of Standard Typeclass Instances

Before getting into the details of how we implemented typeclass refinements (in the next section) we present a case study demonstrating that the pattern we have shown for stating and verifying the laws of standard typeclass instances works well.

In our case study, we considered five standard typeclasses: `Semigroup`, `Monoid`, `Functor`, `Monad`, and `Applicative`. Then we defined subclasses (`VSemigroup`, `VMonoid`, etc.) that contain the parent's expected typeclass laws. We have shown the definitions of `VMonoid` and `VSemigroup` already; `Functor`, `Monad`, and `Applicative` are shown in Figure 3 with their refined subclasses. We defined and verified instances of the above typeclasses for the `All`, `Any`, `Dual`, `Endo`, `Identity`, `List`, `Maybe`, `Peano`, `Either`, `Const`, `State`, `Reader`, and `Succs` datatypes. Because datatypes are instances of multiple subclasses, we performed 34 instance-verifications in total.

This effort was quite manageable. Table 1 tabulates the results, indicating the instance type in the first column, and the typeclasses it implements in the second. For each implementation we tabulate

Type	Typeclass	# Lines Proof	Verif. Time (Std. dev.)
`All`	`Semigroup`	2	1.233 (0.086)
	`Monoid`	2	
`Any`	`Semigroup`	2	1.211 (0.035)
	`Monoid`	2	
`Dual`	`Semigroup`	2	2.023 (0.086)
	`Monoid`	2	
`Endo`	`Semigroup`	2	1.198 (0.078)
	`Monoid`	2	
`Identity`	`Semigroup`	2	1.560 (0.142)
	`Monoid`	2	
	`Functor`	2	2.874 (0.009)
	`Applicative`	4	
	`Monad`	4	
`List`	`Semigroup`	3	1.360 (0.118)
	`Monoid`	3	
	`Functor`	4	7.801 (0.191)
	`Applicative`	25	
	`Monad`	10	

Type	Typeclass	# Lines Proof	Verif. Time (Std. dev.)
`Maybe`	`Semigroup`	3	2.607 (0.179)
	`Monoid`	3	
	`Functor`	3	4.504 (0.161)
	`Applicative`	8	
	`Monad`	6	
`Peano`	`Semigroup`	3	1.291 (0.117)
	`Monoid`	3	
`Either`	`Functor`	3	4.084 (0.244)
	`Applicative`	8	
	`Monad`	6	
`Const`	`Functor`	2	0.921 (0.169)
`State`	`Functor`	12	1.113 (0.156)
`Reader`	`Functor`	11	2.184 (0.103)
	`Applicative`	21	
`Succs`	`Functor`	2	341.730 (1.794)
	`Applicative`	18	

Table 1. Total lines of proofs for each typeclass instance and the average verification time in seconds. Each reported time covers the laws on its row and those on the following rows up to the next reported time.

the lines of proof required to verify the stated laws. We also report the average (and standard deviation) of the time (in seconds) it took Liquid Haskell to verify each module.[1]

For many of the proofs, Liquid Haskell is able to automatically verify the typeclass properties using PLE (Proof by Logical Evaluation) [Vazou et al. 2017; Leino and Pit-Claudel 2016]. As such, most of the proofs are a couple of lines of code. In general, PLE reduces manual effort but increases verification time, but for most modules the proofs are checked within just a few seconds. There are some exceptions—the `List`, `Reader`, and `Succs Applicative` instances are more involved, with the `Succs` module taking hundreds of seconds to verify. Unlike the other proofs, which usually require no more than two or three lemmas, the proof of the composition law of `Succs` involves applying nine separate lemmas. The lemmas give more candidates for PLE to rewrite, but most of the rewritings do not lead to the correct solution, and just slow things down.

In sum, this case study shows that typeclass refinements constitute a natural and modular approach to stating typeclass laws and proving that they are satisfied by their instances. Section 4 presents further evidence, in the form of our `VRDT` case study, of the utility of typeclass refinements.

3 IMPLEMENTING TYPECLASS REFINEMENTS

Now we turn to the question of how we extended Liquid Haskell to implement typeclass refinements.

Liquid Haskell statically verifies Haskell programs by analyzing *Core* expressions. Core is a small, explicitly-typed variant of System F generated during compilation by GHC, the Glasgow Haskell Compiler. Liquid Haskell can thus ignore many of Haskell's myriad source-level constructs, and focus on a smaller language. This implementation approach is also useful for managing Liquid Haskell as an independent codebase. Even as Haskell is actively modified with new or improved features, Liquid Haskell needs no modification because those features are translated to Core.

The challenge with implementing typeclass refinements is that GHC removes typeclasses entirely during the translation to Core; each typeclass is replaced with a *dictionary* of its various operations.

[1] All experiments in this paper were carried out by the `criterion` Haskell package, which repeatedly reruns benchmarks until the error is small enough [O'Sullivan 2020]. Typically, `criterion` ran up to 15 trials. The experiments were run on a machine with an Intel Xeon CPU with 64GB of RAM, running Ubuntu 16.04 with Z3 version 4.4.1.

```
class Functor f where
  fmap :: (a → b) → f a → f b
  (<$) :: a → f b → f a

class Functor m ⇒ VFunctor m where
  lawFunctorId :: x:m a → {fmap id x = id x}
  lawFunctorComposition :: f:(b → c) → g:(a → b) → x:m a
                        → {fmap (f . g) x = (fmap f . fmap g) x}

class Functor f ⇒ Applicative f where
  pure :: a → f a
  (<*>) :: f (a → b) → f a → f b
  liftA2 :: (a → b → c) → f a → f b → f c
  (*>) :: f a → f b → f b
  (<*) :: f a → f b → f a

class (VFunctor f, Applicative f) ⇒ VApplicative f where
  lawApplicativeId :: v:f a → {pure id <*> v = v}
  lawApplicativeComposition :: u:f (b → c) → v:f (a → b) → w:f a
                            → {pure (.) <*> u <*> v <*> w = u <*> v <*> w}
  lawApplicativeHomomorphism :: g:(a → b) → x:a → {px:f a | px = pure x}
                             → {pure g <*> px = pure (g x)}
  lawApplicativeInterchange :: u:f (a → b) → y:a
                            → {u <*> pure y = pure ($ y) <*> u}

class Applicative m ⇒ Monad m where
  (>>=) :: m a → (a → m b) → m b
  (>>) :: m a → m b → m b
  return :: forall a. a → m a

class (VApplicative m, Monad m) ⇒ VMonad m where
  lawMonad1 :: x:a → f:(a → m b) → {f x = return x >>= f}
  lawMonad2 :: m:m a → {m >>= return = m }
  lawMonad3 :: m:m a → f:(a → m b) → g:(b → m c)
            → {h:(y:a → {v:m c | v = f y >>= g}) | True}
            → {(m >>= f) >>= g = m >>= h}
  lawMonadReturn :: x:a → y:m a → {(y = pure x) ⇔ (y = return x)}
```

Fig. 3. Typeclass definitions for `Functor`, `Applicative`, and `Monad` and their associated laws.

Thus, our extension to Liquid Haskell needs a way to connect the refinements the programmer writes on typeclass methods with the translated Core that comes back from GHC, and it needs to do so in a way that is robust to (at least some) future changes in GHC's elaboration. This section explains how we do this by delegating as much work as possible to GHC. We also explain how we model the fact that typeclass elaboration is coherent by default, to simplify user proofs.

3.1 GHC Typeclass Elaboration

Haskell compilers, including GHC, translate typeclass definitions and instances to datatypes known as *dictionaries* [Sulzmann et al. 2007]. As an example, the Semigroup typeclass definition from Figure 2(a) is translated to a dictionary as the following datatype, `Semigroup` (simplified for clarity).

```
data Semigroup a = CSemigroup { (<>) ::  a → a → a }
```

The datatype `Semigroup a` has a single constructor `CSemigroup` and one field for the `<>` method. In general, one field is defined for each typeclass method.

Typeclass instances are translated into dictionary values. For example, the list Semigroup instance from Figure 2(a) generates a `Semigroup [a]` dictionary, which GHC names `$fSemigroup[]`.

```
$fSemigroup[] :: Semigroup [a]
$fSemigroup[] = CSemigroup ($c<>[])
$c<>[]        = (++)
```

The dictionary's field is the list append method `(++)`, which is assigned to the generated variable `$c<>[]`. (Both the dictionary and field variables are prefixed with `$` to indicate they are internal variable names, and posfixed with `[]` to indicate the list instance.)

Elaboration. The translated dictionaries are inserted after each method call via a procedure known as *elaboration*. For example, the Haskell code `x <> y` that appends two list variables `x, y :: [a]` is elaborated to `(<>) \$fSemigroup [] x y`, where now `(<>)` is the record selector of the `Semigroup` data type. Functions that explicitly mention the `Semigroup a` constraint, as in `f` below, are elaborated to take an explicit dictionary argument; `f` elaborates to `fElab`, on the right.

```
f :: Semigroup a ⇒ a → a → a          fElab :: Semigroup a → a → a → a
f x y = x <> y                         fElab d x y = (<>) d x y
```

Subclass Encoding and Coherence. In Core, subclass dictionaries store references to parent dictionaries as fields. For example, the dictionary of the `VMonoid` typeclass from Section 2.2 has four fields, two for the class methods and two for the superclass dictionaries:

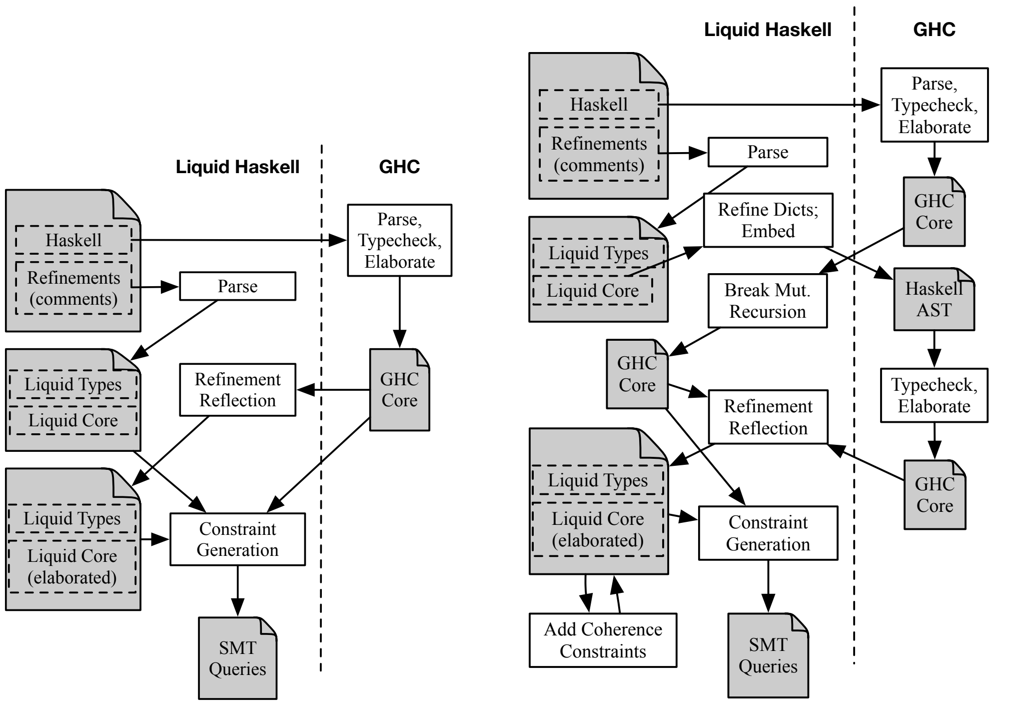
```
data VMonoid a =  CVMonoid {
    p1VMVSemigroup :: VSemigroup a
  , p2VMMonoid     :: Monoid a
  , lawEmpty       :: a → ()
  , lawMconcat     :: [a] → ()
  }
```

Interestingly, `Semigroup` is a superclass of both `Monoid` and `VSemigroup`, which leads to the "diamond problem." When the user writes `x <> y`, it is unclear if GHC's elaboration will access `(<>)` via the `Monoid` or via the `VSemigroup` field. That is, GHC can elaborate the `coherence` code below to either `coherenceElab1` or `coherenceElab2`.

```
coherence :: VMonoid a ⇒ a → a → a
coherence x y = x <> y

coherenceElab1, coherenceElab1 :: VMonoid a → a → a → a
coherenceElab1 d x y = (<>) (p1VSSemigroup (p1VMVSemigroup d)) x y
coherenceElab2 d x y = (<>) (p1MSemigroup  (p2VMMonoid d)) x y
```

Here, `p1VSSemigroup` and `p1MSemigroup` access the semigroup dictionary from the `VSemigroup` and `Monoid`, respectively. Such nondeterminism of elaboration could lead to problems, as the runtime semantics of `coherence` could change with GHC's elaboration decision. Fortunately, by default GHC's elaboration is *coherent* [Bottu et al. 2019], meaning that the dictionary for each typeclass instance at a given type is unique; as such, we know that the `Semigroup` dictionary is the same irrespective of how it is accessed, i.e., `(p1VSSemigroup . p1VMVSemigroup) = (p1MSemigroup . p2VMMonoid)`. Such an equality may be needed in a proof, so our implementation reflects it (in a safe manner) in the proof state, as discussed in Section 3.3.



(a) Liquid Haskell's original architecture. (b) Liquid Haskell's architecture with typeclass support.

Fig. 4. Changes to Liquid Haskell's architecture.

3.2 Interaction with GHC

Now we explain how we modified Liquid Haskell's interaction with GHC so that we can verify typeclass refinements.

Refinements are ported to Dictionaries. The core intuition of typeclass verification is that refinements on typeclasses should be turned into refinements on the respective GHC-translated dictionaries. For example, the dictionary for the `VSemigroup` refined typeclass of Figure 2(b) should be refined to carry the associativity proof obligation (as a normal refinement):

```
data VSemigroup a = CSemigroup {
  p1VSSemigroup    :: Semigroup a
, lawAssociativity :: x:a → y:a → z:a → {x <> (y <> z) == (x <> y) <> z}
}
```

(Here, the first field of the dictionary is a link to the parent typeclass.) But of course we cannot literally do this because `lawAssociativity` is not well-formed Core. We make it so by expanding Liquid Haskell's interaction with GHC, using it to parse, typecheck, and elaborate refinements.

Liquid Haskell's Architecture. Figure 4 summarizes Liquid Haskell's architecture and interaction with GHC API before and after support for refined typeclasses.

The first step is similar in both architectures: send Haskell source to GHC, which comes back as Core, and parse out refinement types appearing in comments. *Before* typeclass support (i.e., Figure 4a), the Core expressions were used by Liquid Haskell to strengthen the exact types for reflected functions, via refinement reflection, and to generate the verification constraints that were finally checked by an SMT. *After* typeclass support, the returned Core may include dictionary definitions, elaborated implementations of those dictionaries, and elaborated clients that use them, as explained in Section 3.1. These dictionaries need to be connected to the refinement types retrieved from typeclass methods.

To connect typeclass method refinements to elaborated dictionaries, Liquid Haskell converts the parsed-out refinements into Haskell abstract syntax trees that make explicit reference to the relevant typeclasses. This occurs in the *Refine Dicts; Embed* step of the architecture diagram. As an example, for the `VSemigroup` method's refinement of `lawAssociativity` (Figure 2), the following Haskell source-code AST expression is constructed:

```
(\x y z () → x <> (y <> z) = (x <> y) <> z) :: VSemigroup a ⇒
    a → a → a → () → Bool
```

GHC typechecks and elaborates the expression in the context of the `VSemigroup` definition it saw previously, and as such applies the appropriate dictionary arguments to typeclass methods. It returns the following:

```
(\d x y z () → x `<> (p1VSemigroup d)` (y `<> (p1VSemigroup d)` z) =
  (x `<> (p1VSemigroup d)` y) `<> (p1VSemigroup d)` z)
:: VSemigroup a → a → a → a → () → Bool
```

Now the dictionary `d` is explicit in the elaborated Core expression. Liquid Haskell converts this into a refinement expression using refinement reflection, and then combines it with the Core code returned from GHC in the first step; for the example, the constructor and selectors for `VSemigroup` in Figure 2 are the following:

```
data VSemigroup a = CVSemigroup {
  p1VSSemigroup    :: Semigroup a
, lawAssociativity :: x:a → y:a → z:a
    → {x `<> p1VSSemigroup` (y `<> p1VSSemigroup` z)
       == (x `<> p1VSSemigroup` y) `<> p1VSSemigroup` z}
}

lawAssociativity :: d:VSemigroup a → x:a → y:a → z:a
  → {x `<> (p1VSemigroup d)` (y `<> (p1VSemigroup d)` z)
     == (x `<> (p1VSemigroup d)` y) `<> (p1VSemigroup d)` z}
```

First, notice that `lawAssociativity` basically matches the elaborated Core expression returned by GHC, but it has been converted to match Liquid Haskell's refinement type syntax. Second, notice that the data constructor for `VSemigroup` also makes use of the returned Core, but has applied one additional step of transformation so that it can refer to the superclass' (i.e., `Semigroup`) operator. Now, `VSemigroup` instances must satisfy the required properties since the dictionary datatype has been refined.

Typeclass methods do not only appear in the refinements of typeclasses. Functions with typeclass constraints may also contain typeclass methods in their refinements. We also elaborate the refinement expressions of these functions so that the appropriate dictionary arguments to typeclass methods are applied.

This whole process corresponds to the *refine dicts; embed*, *typecheck, elaborate*, and *refinement reflection* steps in the diagram. The *breaking mutual recursion* step inlines selector calls in the derived GHC dictionaries to break superficial mutual recursion since Liquid Haskell requires explicit proof of termination. The *adding coherence constraints* step is detailed in Section 3.3.

Our implementation of all of this amounts to about 2000 lines of code, and is part of a fork Liquid Haskell's codebase which is up to date with the main trunk as of May, 2020. Around 400 lines of code is used to define functions that communicate with the GHC API. The top-level driver function that orchestrates GHC's *Typecheck; Elaborate* and Liquid Haskell's *Refine Dicts; Embed* step takes another 700 lines of code. This also includes the embedding functions from Liquid Haskell predicates to the Haskell AST. The rest of the code roughly corresponds to the *Refinement Reflection* step, where elaborated dictionaries are being converted into ordinary refined data types which Liquid already knows how to process and verify.

Limitation: Incompatibility with SMT interpreted Operators. One limitation with our implementation's approach is that it is incompatible with the embedding of SMT theories. Liquid Haskell embeds operations from `Num`, `Ord` and `Eq` into the corresponding theories provided by the SMT solver. This allows Liquid Haskell to efficiently discharge theory-related proofs using existing decision procedures. However, if we elaborate an expression that uses the + operation, then + would no longer be treated a binary numerical operation, but rather as a data accessor that retrieves a binary operation from a `Num` dictionary. Currently, we simply add a special case which drops the dictionary if it corresponds to an instance of one of those three classes. By pretending those classes do not exist, we are still able to utilize the full power of the SMT solver, but we lose the ability to verify interesting instances, such as the `Num` instance of the free algebraic graph as defined Mokhov [2017]. It would be interesting to explore how to get the best of both worlds: quickly discharging theory-related proofs and accessing the concrete definitions, as needed.

3.3 Reasoning About Coherence

In Section 3.1 we mentioned that GHC's elaboration is, by default, coherent. Various proofs on typeclass methods rely on coherence of elaboration (i.e., only hold when instances are globally unique). Here, we give an example of such a proof and detail how coherence is automatically encoded and checked using refinement types.

lawMconcat for the VMonoid Dual Instance Requires Coherence. Given any binary operation `<>`, we can always define a dual operation `<+>`:

```
x <+> y = y <> x
```

Therefore, if we have a `Semigroup` instance for some type `a`, we can also define a `Dual` instance of a `Semigroup`. Indeed, we can define `Semigroup`s of `Dual`s *of any type* `a` once and for all:

```
newtype Dual a = Dual {getDual :: a}
instance Semigroup a ⇒ Semigroup (Dual a) where
  Dual (v :: a) <> Dual (v' :: a) = Dual (v' <> v)
```

Now, whenever we define a `Semigroup` instance for some type, GHC will automatically create its corresponding `Dual` instance. We do the same for `Monoid`, `VSemigroup`, and `VMonoid`.

```
instance Monoid a ⇒ Monoid (Dual a) where
  mempty     = Dual mempty
  mconcat xs = foldr (<>) mempty xs

instance VSemigroup a ⇒ VSemigroup (Dual a) where
  lawAssociative (Dual v) (Dual v') (Dual v'') = lawAssociative v'' v' v

instance VMonoid a ⇒ VMonoid (Dual a) where
  lawEmpty (Dual v) = lawEmpty v
  -- lawMconcat :: VMonoid a ⇒ xs:_ → {mconcat xs = foldr (<>) mempty xs}
  lawMconcat xs = () `const` mconcat xs
```

The proof of `lawMconcat` proceeds by a simple unfolding of the `mconcat` definition, which is expressed as a call to that function (the full proof must be then cast to unit via Haskell's `const ()`).

This proof requires coherence to hold. To see why, consider the proofs for the elaborated definitions. The elaborated equality `mconcat xs = foldr (<>) mempty xs` is as follows, for the dictionary `d :: VMonoid a`.

```
    mconcat xs
=   by elaboration
    mconcat ($fSemigroupDual (p2VMMonoid d)) xs
=   by unfolding of mconcat
    foldr ((<>) ($fSemigroupDual (p1SMonoid (p2VMMonoid d))))
               (mempty ($fMonoidDual (p2VMMonoid d)))
=   by coherence
    foldr ((<>) ($fSemigroupDual (p1VSSemigroup (p1VMVSemigroup d))))
               (mempty ($fMonoidDual (p2VMMonoid d)))
=   by de-elaboration
    foldr (<>) mempty xs
```

The proof requires a coherence step to equate the two different ways that the semigroup operator `<>` is accessed. Concretely, the above equational proof only holds when `p1VSSemigroup (p1VMVSemigroup d)` equals `p1SMonoid (p2VMMonoid d)`, for all `d :: VMonoid a`. This equality cannot be asserted by the programmer, as dictionaries do not appear in the source.

Coherence as Dictionary Refinements. We represent the expected effect of coherent resolution as a refinement type on the datatype dictionary definitions for typeclasses. In particular, the equality of ancestor typeclasses is expressed as a refinement type on the fields of parent typeclasses. For example, the Core representation of `VMonoid` is as follows.

```
data VMonoid a = CVMonoid {
  p1VMVSemigroup :: VSemigroup a
, p2VMMonoid :: { v:Monoid a | p1VSSemigroup p1VMVSemigroup = p1MSemigroup v }
, ... -- as before
```

The added refinement on the second field states that the `Semigroup` from the `VSemigroup` parent (i.e., `p1VSSemigroup p1VMVSemigroup`) must be equal to the `Semigroup` from the `Monoid` parent (i.e., `p1MSemigroup v`). As a result, coherence becomes a *checked invariant* for each `VMonoid`—Liquid Haskell assumes the property for any `VMonoid` but checks it for instance declarations. This approach is sound even when GHC's elaboration is potentially incoherent due to use of the `INCOHERENT` language pragma. Clients of dictionaries will still assume the invariant, but constructed dictionaries (i.e., typeclass instances) will induce an error from Liquid Haskell if the invariant cannot be proved.

Being able to take advantage of Haskell's coherent typeclass resolution is the silver lining of our limitation that refined types cannot be distinct typeclass instances, as discussed at the of end of Section 2.2. If we allowed different instances of a class `TC` for both `{ v:int | v > 0 }` and `int`, say, then we could not use Haskell's proved-coherent mechanism, and would have to develop our own and/or allow one to be customized as part of proof search. There is no guarantee that the result would be coherent. In other dependently typed systems with typeclasses, e.g., Coq, the user has to explicitly encode and prove coherence requirements, case by case (see Section 5).

4 CASE STUDY: VERIFIED REPLICATED DATA TYPES

This section presents our platform for programming distributed applications based on *Conflict-free Replicated Data Types* (CRDTs) [Shapiro et al. 2011b]. We define a Haskell typeclass `VRDT` to represent CRDTs and use type refinements to state the mathematical properties that CRDTs are expected to satisfy. We then prove in Liquid Haskell that `VRDT` instances (which must satisfy these properties) are sure to enjoy *strong convergence*. We have implemented a several primitive instances of the `VRDT` typeclass as well a mechanism for building compound `VRDT`s based on smaller components, where the necessary properties of the former automatically follow from the latter. With this infrastructure, as well as libraries for message delivery and user interaction we developed, we constructed two substantial applications, a shared event planner and a collaborative text editor (though the latter relies on a `VRDT` we have not fully verified, per Section 4.3).

4.1 Background: Conflict-free Replicated Data Types (CRDTs)

A CRDT is a data structure with certain mathematical properties, designed for use in a distributed system that replicates its state. These properties enable proving that such a system enjoys the *strong eventual consistency* (SEC) property, a key aspect of which is *strong convergence* (SC), which states that replicas of a CRDT that have received and applied the same set of updates will always have the same state, regardless of the order in which those updates are received and applied.

Shapiro et al. [2011b] describe two styles of CRDT specifications: *state-based*, in which replicas apply updates locally and periodically broadcast their local state to other replicas, which then merge the received state with their own, and *operation-based*, in which every state-updating operation is broadcast and applied at each replica. Shapiro et al. [2011b] prove that state-based and operation-based CRDTs are equivalent in the sense that each can be implemented in terms of the other (although practical implementation considerations may motivate the choice of one or the other in a particular application). In this work, we focus our attention on the operation-based style, which is suitable for implementing CRDTs such as ordered lists, which are key for applications such as collaborative text editing.

The key to proving convergence of (operation-based) CRDTs is to require that, under appropriate circumstances, the CRDT's operations commute. A replica that receives an operation from a client can update its own state and broadcast that operation to the other replicas. Since the operations commute, they can be applied in any order and produce the same final state, as required by SC. However, requiring operations to commute under all circumstances is too restrictive. Therefore, Shapiro et al. relax commutativity to exclude *causally ordered* operations. Such operations tend to affect the same parts of the state, and are usually issued in a strict order. For example, the operation of inserting a $k \to v$ pair in a map is causally ordered before an operation that updates k's mapped-to value. The assumption made by Shapiro et al. is that the underlying communication mechanism will ensure causal delivery to the CRDT, e.g., by employing vector clocks [Fidge 1988; Mattern 1989] and buffering received operations until all operations that causally precede them have been applied [Birman et al. 1991]. They also assume that any preconditions that must be satisfied to enable an operation's execution (e.g., that a key must be present in a map if its value is to be updated) are ensured by causal delivery. Sometimes particular CRDTs make global assumptions for correctness, e.g., that generated keys are unique.

4.2 Verifying CRDTs with Typeclass Refinements

We define a CRDT as the Liquid Haskell typeclass `VRDT`, shown in Figure 5(a). Each VRDT has an associated `Op` type that specifies how the VRDT's state is updated. The `apply` method takes a VRDT state `t` runs the given operation `Op t` on it, and returns the updated state.

```
class VRDT t where
  type Op t
  apply :: t → Op t → t
  compat :: Op t → Op t → Bool
  compatS :: t → Op t → Bool

  lawCommut :: x:t → op1:Op t → op2:Op t
      → {(compat op1 op2 &&
           compatS x op1 && compatS x op2)
             ⇒ (apply (apply x op1) op2
               = apply (apply x op2) op1
               && compatS (apply x op1) op2)}

  lawCompatCommut :: op1:Op t → op2:Op t
      → {compat op1 op2 = compat op2 op1}
```

(a) The `VRDT` typeclass for verified CRDTs

```
data Max a = Max a

instance Ord a ⇒ VRDT (Max a) where
  type Op (Max a) = Max a

  apply (Max a) (Max b) | a > b = Max a
  apply (Max a) (Max b)         = Max b

  compat op1 op2 = True
  compatS max op = True

  lawCommut max op1 op2 = ()
  lawCompatCommut op1 op2 = ()
```

(b) An instance for `Max a` of `VRDT`

Fig. 5. Definition of the `VRDT` typeclass and its `Max` instance

Rather than formalize a general notion of causal delivery and additionally formalize any global correctness assumptions, we combine both together in a pair of predicates, `compat` and `compatS`. The former should return `True` for non-causally ordered operations (and perhaps others) that satisfy the global correctness assumptions. The latter will return `True` when the current state is compatible with the given operation, according to the global correctness assumptions. For example, in our `TwoPMap` key-value map implementation given below, we express the assumption of unique keys by deeming two insertion operations incompatible if they offer the same key, and deeming an insertion incompatible with a state that already contains the offered key. Our notion of compatibility is more flexible than a one-size-fits-all notion of causality. For example, while inserting a key-value pair is causally ordered with deleting that pair, these two operations can be deemed compatible (and are, in our `TwoPMap`) by internally buffering the latter until the former is delivered. It is up to the `VRDT` instance to decide, and reflect in its specifications of `compat` and `compatS`, what to do itself, and what to expect of the delivery mechanism.

The required mathematical properties of a `VRDT` are specified extrinsically as methods `lawCommut` and `lawCompatCommut`. The former's type specifies the property that operations compatible with each other and the current state must commute. The latter's type expresses that the operation-compatibility predicate `compat` must also be commutative.

Primitive VRDTs. An example `VRDT` instance, `Max`, is given in Figure 5(b). It contains a polymorphic value with a defined ordering (specified with an `Ord` instance) and tracks the maximum value of that type. Its corresponding operation's type `Op` is itself. All pairs of operations are compatible, so `compat` and `compatS` always return `True`. The `apply` function updates `Max`'s state by taking the greatest value of the two arguments. Proofs of `lawCommut` and `lawCompatCommut` for `Max` are trivial and Liquid Haskell proves them automatically.

In addition to `Max` we have implemented and mechanically verified four more primitive VRDTs.

- `Min v` is the dual of `Max v`, and tracks the smallest value seen.
- `Sum v` is an implementation of Shapiro et al. [2011a]'s Counter. `Op`s are numbers and the state is their sum.

- `LWW t v` is an implementation of Shapiro et al.'s *Last Writer Wins* Register. When a node writes to a register, it attaches a (polymorphic) timestamp to the value. A receiving node only updates its value if the timestamp is greater than the current timestamp. `LWW` assumes that all timestamps are unique.
- `MultiSet v` maintains a collection of values, like a `Set`, but each member has an associated count; a non-positive count indicates logical non-membership. `Op`s include value insertion and removal, each with an associated count. Besides using Liquid Haskell to prove `Multiset` is a proper `VRDT`, we also proved its semantics simulates the semantics of mathematical multisets; details are in Appendix A.

We have also implemented Grishchenko's *causal trees* [2010], which maintain an ordered sequence of values, but only partially verified their correctness. In a `CausalTree`, each value is assumed to have a unique identifier, and each value knows the identifier of the previous value. The relationship to the previous value creates a tree data structure that can be traversed (in preorder) to recover a converging ordering. Causal trees, like other RDTs representing ordered sequences (e.g., Roh et al. [2011]; Oster et al. [2006]; Preguica et al. [2009]; Weiss et al. [2009]), are useful for implementing collaborative text editing, but their behavior is considered especially challenging to specify and verify [Attiya et al. 2016; Gomes et al. 2017]. We are prevented from completing our proof by a bug in Liquid Haskell (see Section 4.3), but hope to rectify the problem soon.

A Compound VRDT: Two-phase Map. We can also define VRDTs by reusing other VRDTs. In doing so, proofs of a compound VRDT's required properties can be proved (in part) by using the properties of the VRDTs it is building on. As an example compound VRDT, we have implemented a *two-phase map*, shown in Figure 6. `TwoPMap` implements a map from keys to values, where values are themselves VRDTs. A `TwoPMap`'s operations are given by the datatype `TwoPMapOp`. Operation `TwoPMapInsert k v` inserts a `k→v` mapping; operation `TwoPMapDelete k` deletes a key; and `TwoPMapApply k (Op v)` applies a VRDT operation `Op` on `k`'s value `v`. An important restriction on a `TwoPMap`'s operation is that a key can only be used once in the map; even once a key is deleted it can never be re-added. This restriction is expressed in the definition of `compat` and `compatS`, which also lifts the requirement that the value VRDT's operations are compatible. A few additional cases are omitted from the compatible predicates for brevity.

A two-phase map would naturally require causal delivery because a happens-before relationship exists between inserting and updating (or inserting and deleting) a value in the map. `TwoPMap` avoids the need for causal delivery (and thus does not specify it via `compat` or `compatS`) by buffering pending operations on a given key. The `apply` code for `TwoPMapApply` stores operations on the value of `k` in a separate operations buffer if `k` does not yet exist in the map. The `apply` code for `TwoPMapInsert` checks `k`'s operation buffer and applies any operations on the given value before inserting it with the mapping. The `apply` code for `TwoPMapDelete` clears `k` from the map and operations buffer, and adds it to the *tombstone*; future attempted insertions of `k` will be ignored due to its presence there. In general we assume that, like `TwoPMap`, instances of `VRDT` do not require causal delivery. Like `TwoPMap`, this requirement is easily satisfied by pushing a buffer of pending operations into the data type itself.

Automatically Deriving Compound VRDTs. Generally speaking, we might like to collect together several VRDTs to create an aggregate whose operations delegate to the operations of the components. For example, the shared event planner we discuss in Section 4.4 represents a calendar event as a record with a title, description, time, and guest RSVP tally. To implement such a record as a VRDT, we can represent the first three fields as `LWW` registers and the last as a `MultiSet`, as shown in Figure 7. However, just collecting separate VRDTs together is not enough to show that the result is

```
data TwoPMap k v = TwoPMap {
    twoPMap          :: Map k v
  , twoPMapTombstone :: Set k
  , twoPMapPending   :: Map k [Op v]
  }
```

```
data TwoPMapOp k v =
    TwoPMapInsert k v
  | TwoPMapDelete k
  | TwoPMapApply k (Op v)
```

```
instance (VRDT v, Ord k, Ord (Op v)) ⇒ VRDT (TwoPMap k v) where
 type Op (TwoPMap k v) = TwoPMapOp k v

 compat (TwoPMapInsert k v) (TwoPMapInsert k' v') | k == k' = False
 compat (TwoPMapApply k op) (TwoPMapApply k' op') | k == k' = compat op op'
 compat _                   _                               = True

 compatS (TwoPMap m t p) (TwoPMapInsert k v) = Map.lookup k m == Nothing
 compatS (TwoPMap m t p) (TwoPMapApply k o)  | Just v ← Map.lookup k m = compatS v o
 compatS _               _                   = True

 apply (TwoPMap m p t) (TwoPMapInsert k v) | Set.member k t = TwoPMap m p t
 apply (TwoPMap m p t) (TwoPMapInsert k v) =
   -- Apply pending operations.
   let (opsM, p') = Map.updateLookupWithKey (const (const Nothing)) k p in
   let v' = maybe v (foldr (\op v → apply v op) v) opsM in
   let m' = Map.insert k v' m in
   TwoPMap m' p' t

 apply (TwoPMap m p t) (TwoPMapApply k op) | Set.member k t = TwoPMap m p t
 apply (TwoPMap m p t) (TwoPMapApply k op) =
   let (updatedM, m') = Map.updateLookupWithKey (\_ v → Just (apply v op)) k m in
   -- Add to pending if ¬inserted.
   let p' = if isJust updatedM then p else insertPending k op p in
   TwoPMap m' p' t
```

```
  apply (TwoPMap m p t) (TwoPMapDelete k) =
    let m' = Map.delete k m in
    let p' = Map.delete k p in
    let t' = Set.insert k t in
    TwoPMap m' p' t'
```

```
 lawCommut _ _ _ = ...
 lawCompatCommut _ _ = ...
```

Fig. 6. Implementation of TwoPMap

```
data Event = Event {
    eventTitle :: LWW Timestamp Text
  , eventDescription :: LWW Timestamp Text
  , eventStartTime :: LWW Timestamp UTCTime
  , eventRSVPs :: MultiSet Text
  }
```

Fig. 7. Data type for a calendar event that is made up of VRDTs.

a VRDT: we need to define a corresponding `Op` data type and a `VRDT` instance for `Event`. Fortunately, since the fields of `Event` are `VRDT` instances, it is possible to derive the `Event` operation and `VRDT` instance automatically. We use Template Haskell [Sheard and Peyton Jones 2002] to, at compile time, generate operations and `VRDT` instances for data types that are composed of other `VRDTs`. Liquid Haskell can automatically verify that the generated code satisfies the `VRDT` properties.

4.3 Proofs

We have proved, in Liquid Haskell, that Strong Convergence, the key safety property required by Strong Eventual Consistency [Shapiro et al. 2011b], holds for `VRDT` instances.

```
strongConvergence ::
  (Eq (Op a), VRDT a) ⇒
   s0:a → ops1:[Op a] → ops2:[Op a] →
    { (isPermutation ops1 ops2 && allCompatible ops1 && allCompatibleState s0 ops1)
       ⇒ (applyAll s0 ops1 = applyAll s0 ops2)}
```

The theorem states that if two lists of operations are permutations of one another, then applying either one to the same input VRDT state will produce the same output state, assuming the list contains mutually compatible operations, and that all of these operations are compatible with the initial state. The proof is by induction over the operation lists and makes use of `lawCommut` and `lawCompatCommut` laws of `VRDT`. Importantly, the proof is independent of any *particular* `VRDT` instance, and thus applies to all of them.

Table 3 summarizes the lines of proof and verification time for the VRDTs we built. The development totals 2092 lines of code. These also include duplicate definition of Haskell functions in a way amenable to verification. For example, `Data.Map` was redefined to prove it satisfies the sortedness invariant, while common list functions were redefined to be reflected, as required by extrinsic proofs. As expected, Liquid Haskell's PLE and SMT automation over intrinsic properties (e.g., sortedness invariant on `Data.Map`) aided proof generation. That said, there are still some issues to iron out. For example, there are difficulties proving properties of code that makes use of typeclasses that have SMT-interpreted theories in Liquid Haskell, e.g., set theory used by the verified `Data.Map`. In fact, an existing limitation of this combination is blocking the verification of the `CausalTree` instance that we will resume once the Liquid Haskell limitation is addressed. The proof of `TwoPMap` also ran very slowly; because of the large search space (9 case splits between the 3 operations), the verification took more than 90 minutes. The long verification time can be attributed to PLE's expansion and the discharging of verification conditions by the SMT solver. The bloated verification conditions consume a significant amount of memory space as well; when verifying the insert/apply case of `TwoPMap`, Liquid Haskell exhausted the 16 GiB physical memory and consumed no less than 1 GiB of the swap space.

In short, the verification effort was strenuous, which was expected as the first, real-world case study of refined typeclasses. Nevertheless, this case study increases our confidence that Liquid Haskell's automation reduces proof effort and, since most of the implementation limitations we faced are already addressed, refined typeclasses in Liquid Haskell can actually be used to verify sophisticated properties of real-world applications.

4.4 Applications

We built two realistic applications that are backed by `VRDT` instances: a *shared event planner* and a *collaborative text editor*. We close out this section by briefly describing these applications and some of the other infrastructure we built beyond VRDTs to put them together.

VRDT	Property	# Lines Proof	Verif. Time (Std. dev.)
-	`strongConvergence`	320	122.043 (2.415)
`Max`	`lawCommut`	1	0.544 (0.050)
	`lawCompatCommut`	1	
`Min`	`lawCommut`	1	0.565 (0.037)
	`lawCompatCommut`	1	
`Sum`	`lawCommut`	1	0.473 (0.028)
	`lawCompatCommut`	1	

VRDT	Property	# Lines Proof	Verif. Time (Std. dev.)
`LWW`	`lawCommut`	1	0.835 (0.048)
	`lawCompatCommut`	1	
`Multiset`	`lawCommut`	315	48.555 (0.943)
	`lawCompatCommut`	1	
	`simulation`	72	45.705 (4.473)
`TwoPMap`	`lawCommut`	1253	5666.866 (56.797)
	`lawCompatCommut`	3	

Table 3. Total lines of proofs for each typeclass instance and the average verification time in seconds. Verifications times for `lawCommut` and `lawCompatCommut` are combined.

Message delivery and UI components. Both of our applications build on Haskell libraries we developed for message delivery and user interfaces. In particular, we developed a message delivery client and server to broadcast un-ordered messages from each client to all other clients. We also developed an application programming interface (API) to the client which transparently handles network disconnections by buffering and re-sending outgoing messages. Applications provide to the client API a function to receive messages, and the client API produces a function with which the application may send messages. Appendix B provides further details.

Our user interface library is based around *functional reactive programming* (FRP), a programming paradigm that models values that change over time [Elliott and Hudak 1997]. FRP values are either continuous, called *behaviors*, or discrete, called *events*. We can treat replicated data types as FRP values whose state changes as a result of actions by the local user or update messages from a remote replica. We use Reflex [2], a Haskell FRP library, to integrate FRP applications with our message delivery system. Any `VRDT` instance whose operations can be marshalled and sent over a network, e.g., as JSON, can be used as the state of these distributed applications. We provide the following library function, which internally calls the client API to connect a FRP application client to the server.

```
connectToStore :: (VRDT a, Serialize (Op a), MonadIO m)
    ⇒ ServerSettings → a → Event (Op a) → m (Dynamic a)
```

`connectToStore` takes the settings of the server to connect to and an initial state. It also receives an `Event` of `Op`s. Any time the FRP client performs an operation, the event fires and this function sends the operation to the server. `Dynamic` is a special Reflex type that is both an event and a behavior. Whenever its value changes, an event fires as well. Since `connectToStore` returns a `Dynamic` of the current state, the FRP application automatically updates its interface whenever an operation is received and applied to the `VRDT` state.

Event planner. Our shared event planner application allows multiple users to create and manage calendar events and RSVP to event invitations. The planner's state (`TwoPMap UniqueId Event`) is a two-phase map where elements are the VRDT automatically derived from the `Event` type described in Figure 7. `UniqueId` is a pair of `ClientId` and an integer that is always incremented locally by the client application. It is used to ensure that the keys are unique as required by `TwoPMap`. The event planner has a terminal interface that supports viewing the list of events, creating events, updating events, and displaying event details. Since the application's state is a `VRDT` instance, updates are quickly displayed on all clients once they receive the corresponding operations. In this application, 12 lines of code define the types associated with the application's state, and one line of code invokes

[2] https://reflex-frp.org/

Template Haskell to generate the operation type for `Event` and its `VRDT` instance. The rest of the 400 lines of code in the application implement the user interface. The small amount of code necessary for managing replicated data highlights how `VRDT`s make it easy to build a distributed application.

Collaborative text editor. Our collaborative text editor represents the state of the text document being edited as a `CausalTree`. The majority of the code in the text editor (278 lines, out of roughly 350) is the `causalTreeInput` function, which has the following signature:

```
causalTreeInput :: Dynamic (CausalTree id Char)
  → Widget (Event (Op (CausalTree id Char)))
```

`causalTreeInput` creates a Reflex `Widget` that builds a text box in the terminal interface that displays the contents of the `CausalTree`, handles scrolling, and processes keystrokes by the user. It takes a `Dynamic` of the `CausalTree` as input so that the view is updated when operations from the network are received. It returns an `Event` of `CausalTree` operations that fires whenever keystrokes update the state of the document.

5 RELATED WORK

5.1 Verification of Haskell's Typeclass Laws

Verification of inductive properties, including per-instance typeclass laws, is possible in Haskell using dependently typed features [Eisenberg 2016; Weirich et al. 2019; Xie et al. 2019; McBride 2002]. In work closely related to ours, Scott and Newton [2019] verify algebraic laws of typeclasses using a singletons encoding of dependent types [Eisenberg and Weirich 2012], and they employ generic programming to greatly reduce the proof burden. Even though their generic boilerplate technique is very effective for verifying typeclass instances, it is unclear how the encoding of typeclass laws interacts with the rest of the Haskell code that uses those instances. In our approach, typeclass laws are expressed as refinement types and smoothly co-exist with refinement type specifications of clients of typeclass methods. In fact, Scott and Newton initially attempted to use Liquid Haskell, but it was impossible to do so at the time since Liquid Haskell did not yet support refinement types for typeclasses.

Haskell researchers have developed various techniques outside of Haskell itself to increase their confidence that typeclass laws actually hold. For example, Jeuring et al. [2012] and Claessen and Hughes [2011] used QuickCheck, the property-based random testing tool, to falsify typeclass laws. Zeno [Sonnex et al. 2012] and HERMIT [Farmer et al. 2015] generate typeclass law proofs by term rewriting while HALO [Vytiniotis et al. 2013] uses an axiomatic encoding to verify Haskell contracts. HipSpec [Arvidsson et al. 2016; Claessen et al. 2012] reduces typeclass laws to an external, automated-over-induction theorem prover. `hs-to-coq` [Spector-Zabusky et al. 2018] converts Haskell typeclasses and instances to equivalent ones in Coq which can then be proved to satisfy the respective laws.

Compared to these approaches, our technique has three main advantages. First, our proofs are Haskell programs, highly automated by SMT and PLE; unlike the other approaches, when proof automation fails, the user does not need to debug the external solver. Second, our proofs co-exist and interact with non-typeclass-specific Haskell code, so Haskell functions can use class laws to prove further properties (as in the `assoc2` example of Section 2.2). Finally, our within-Haskell verification approach gives the developer the ability to distinguish between verified and original (i.e., non-verified) typeclasses (as in the `Semigroup` example of Section 2.2) and the flexibility to only use verified methods on critical code fragments, thus saving verification time.

5.2 Type System Expressiveness vs. Coherence of Elaboration

Typeclasses, introduced by Haskell [Peyton Jones 2003], have been adopted by PureScript [Freeman 2017] and have inspired related abstractions in many programming languages, including Scala's implicits [Odersky et al. 2017] and Rust's traits [Team Mozilla Research 2017]. These languages (like vanilla Haskell) are not designed for proving rich logical properties, e.g., by making use of dependent types. But such simpler type systems make it possible to implement coherent typeclass resolution; for Haskell in particular, Bottu et al. [2019] prove coherence of GHC's elaboration by showing global uniqueness of dictionary creations. Coherence means that decisions made by typeclass elaboration cannot change the runtime semantics of the program, making it easier to reason about.

Fully dependently-typed languages such as Coq [Sozeau and Oury 2008], Isabelle [Haftmann and Wenzel 2006], Agda [Devriese and Piessens 2011], Lean [de Moura et al. 2015], and F⋆ [Martínez et al. 2019] permit proofs of rich logical properties, and also support typeclasses. However, to maximize expressiveness, their typeclass resolution procedures can end up being divergent or incoherent. For example, in Coq's typeclasses [Sozeau and Oury 2008], instantiation can diverge and is not guaranteed to be coherent since it is not always possible to decide whether two instances overlap [Lampropoulos and Pierce 2018].

In our work, we attempt to strike a balance between these two extremes. We use Liquid Haskell's expressiveness to prove typeclass properties, while we use GHC's less expressive type system to perform resolution. This design reduces our flexibility, as we cannot have distinct typeclasses for two refined types that have the same base type. But, in turn, we gain two nice benefits. First, we reuse GHC's mature elaboration implementation. More importantly, using elaboration on the coherent [Bottu et al. 2019], less expressive type system of Haskell, we break the dilemma between expressiveness of the type system and coherence of elaboration.

5.3 Verifying Replicated Data Types

No verification can take place without a specification, and precisely *specifying* the behavior of replicated data types is a significant challenge in itself. Most work proposing new designs and implementations of replicated data structures (e.g., [Shapiro et al. 2011b,a; Roh et al. 2011]) does not provide formal specifications. An exception is Attiya et al.'s work [2016], which precisely specifies a replicated list object and gives a (non-mechanized) proof that an implementation satisfies the specification.

Burckhardt et al. [2014] proposed a comprehensive framework for formally specifying and verifying the correctness of RDTs, using an approach inspired by axiomatic specifications of weak shared-memory models. Although it is not obvious how to automate Burckhardt et al.'s verification approach, the Quelea [Sivaramakrishnan et al. 2015] programming model uses the Burckhardt et al. specification framework as a contract language embedded in Haskell that allows programmers to attach axiomatic contracts to RDT operations; an SMT solver analyzes these contracts and determines the weakest consistency level at which an operation can be executed while satisfying its contract.[3] Gotsman et al. [2016] develop an SMT-automatable proof rule that can establish whether a particular choice of consistency guarantees for operations on RDTs is enough to ensure preservation of a given application-level data integrity invariant. This approach is implemented in Najafzadeh et al.'s CISE tool. Houshmand and Lesani's Hamsaz system [2019] improves on CISE by

[3]Implementation-wise, in contrast to our approach which uses Liquid Haskell's solver-aided type system, Quelea is implemented in Haskell by directly querying the underlying SMT solver through the Z3 Haskell bindings at compile time, via Template Haskell.

automatically synthesizing a conflict relation that specifies which operations conflict with each other (whereas this conflict relation has to be provided as input to CISE).

Unlike our approach, tools like Quelea, CISE, and Hamsaz do not, in and of themselves, prove correctness properties of RDT implementations, e.g., strong convergence of replicas. Rather, they determine whether or not it is safe to execute a given RDT operation under the assumption that that replicas satisfy a given consistency policy (in the case of Quelea), or whether or not an application-level invariant will be satisfied, given the consistency policies satisfied by individual operations (in the case of CISE and Hamsaz). The goals of these lines of work are therefore complementary to ours: we *prove* a property of RDT implementations (strong convergence) that such tools could then leverage as an assumption to prove *application-level* properties, e.g., that a replicated bank account never has a negative balance. Verification of these application-level properties is important because CRDT correctness alone is not enough to ensure application correctness. (Of course, it would also be possible to prove such application-level properties directly in Liquid Haskell as well.)

Other works [Zeller et al. 2014; Nair et al. 2020; Gomes et al. 2017; Nagar and Jagannathan 2019] directly address proving the correctness of RDT implementations. Zeller et al. [2014] specify and prove SC and SEC for a variety of state-based counter, register, and set CRDTs using the Isabelle/HOL proof assistant. Nair et al. [2020] present an automatic, SMT-based verification tool for specifying state-based CRDTs and verifying application-level properties of them. Neither Zeller et al. nor Nair et al. consider operation-based CRDTs, the focus of this paper.

Gomes et al. [2017] also use Isabelle/HOL to prove SC and SEC; like us, they focus on operation-based CRDTs. In addition to proving that RDT operations commute for three operation-based CRDTs—Shapiro et al.'s counters and observed-remove sets, and Roh et al.'s replicated growable arrays [2011]—Gomes et al. formalize in Isabelle/HOL a network model in which messages may be lost and replicas may crash, and prove that SC and SEC hold (under any behavior of the network model). Although it is possible to extract executable implementations from Isabelle definitions, our semi-automated Liquid Haskell-based approach has the advantage that the programmer can write, and use, mechanically verified RDT implementations without ever leaving Haskell. Gomes et al. bake causal delivery of updates into their network model (following Shapiro et al. [2011b], who assume causal delivery of updates in their proof of SEC for operation-baesd CRDTs); however, we observe that causal delivery is neither necessary nor sufficient to guarantee strong convergence [Nagar and Jagannathan 2019].

Nagar and Jagannathan [2019] address the question of automatically verifying strong convergence of various operation-based CRDTs (sets, lists, graphs) under different consistency policies provided by the underlying data store. They develop an SMT-automatable proof rule to show that all pairs of operations either commute or are guaranteed by the consistency policy to be applied in a given order. Given a CRDT specification, their framework will determine which consistency policy is required for that CRDT. Their CRDT specifications are written in an abstract specification language designed to correspond to the first-order logic theories that SMT solvers support, whereas our verified RDTs are running Haskell code, directly usable in real applications.

6 CONCLUSION

We have presented an extension of Liquid Haskell to allow refinement types on typeclasses. Clients of a typeclass may assume its methods' refinement predicates hold, while instances of the typeclass are obligated to prove that they do. Implementing this extension was challenged by the fact that Liquid Haskell verifies properties of *Core*, the intermediate representation of the Glasgow Haskell Compiler, but typeclasses are replaced with dictionaries (records of functions) during translation to Core. Our implementation expands the interaction between Liquid Haskell and GHC to carry over refinements to those dictionaries during verification, and does so in a way that takes advantage of

Haskell's typeclass resolution procedure being coherent. We have carried out two case studies to demonstrate the utility of our extension. First, we have used typeclass refinements to encode the algebraic laws for the `Semigroup`, `Monoid`, `Functor`, `Applicative`, and `Monad` standard typeclasses, and verified these properties hold of many of their instances. Second, we have used our extension to construct a platform for for distributed applications based on replicated data types. We define a typeclass whose Liquid Haskell type captures the mathematical properties of RDTs needed to prove the property of strong convergence; implement several instances of this typeclass; and use them to build two substantial applications.

REFERENCES

Andreas Arvidsson, Moa Johansson, and Robin Touche. 2016. Proving Type Class Laws for Haskell. In *International Symposium on Trends in Functional Programming*. Springer, 61–74.

Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. 2016. Specification and Complexity of Collaborative Text Editing. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing* (Chicago, Illinois, USA) *(PODC '16)*. ACM, New York, NY, USA, 259–268. https://doi.org/10.1145/2933057.2933090

Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, Vol. 13. 14.

Ken Birman, André Schiper, and Pat Stephenson. 1991. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems* 9 (08 1991), 272–. https://doi.org/10.1145/128738.128742

Gert-Jan Bottu, Ningning Xie, Koar Marntirosian, and Tom Schrijvers. 2019. Coherence of Type Class Resolution. *Proc. ACM Program. Lang.* 3, ICFP, Article Article 91 (July 2019), 28 pages.

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 271–284. https://doi.org/10.1145/2535838.2535848

Koen Claessen and John Hughes. 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices* 46, 4 (2011), 53–64.

Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2012. HipSpec: Automating Inductive Proofs of Program Properties.. In *ATx/WInG at IJCAR*. 16–25.

Robert L Constable and Scott Fraser Smith. 1987. *Partial objects in constructive type theory*. Technical Report. Cornell University.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 378–388.

Dominique Devriese and Frank Piessens. 2011. On the bright side of type classes: instance arguments in Agda. *ACM SIGPLAN Notices* 46, 9 (2011), 143–155.

Richard A. Eisenberg. 2016. Dependent Types in Haskell: Theory and Practice. *CoRR* abs/1610.07978 (2016). arXiv:1610.07978 http://arxiv.org/abs/1610.07978

Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently Typed Programming with Singletons. *SIGPLAN Not.* 47, 12 (Sept. 2012), 117–130. https://doi.org/10.1145/2430532.2364522

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *International Conference on Functional Programming*. http://conal.net/papers/icfp97/

Andrew Farmer, Neil Sculthorpe, and Andy Gill. 2015. Reasoning with the HERMIT: Tool Support for Equational Reasoning on GHC Core Programs. *SIGPLAN Not.* 50, 12 (Aug. 2015), 23–34. https://doi.org/10.1145/2887747.2804303

C. J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference* 10, 1 (1988), 56–66. http://sky.scitech.qut.edu.au/~fidgec/Publications/fidge88a.pdf

Phil Freeman. 2017. *PureScript by Example*. https://doi.org/10.1145/2887747.2804303

GHC 2020. GHC: The Glasgow Haskell compiler. https://www.haskell.org/ghc/.

Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying Strong Eventual Consistency in Distributed Systems. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 109 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133933

Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 371–384. https://doi.org/10.1145/2837614.2837625

Victor Grishchenko. 2010. Deep Hypertext with Embedded Revision Control Implemented in Regular Expressions. In *Proceedings of the 6th International Symposium on Wikis and Open Collaboration* (Gdansk, Poland) *(WikiSym '10)*. Association for Computing Machinery, New York, NY, USA, Article 3, 10 pages. https://doi.org/10.1145/1832772.1832777

Florian Haftmann and Makarius Wenzel. 2006. Constructive type classes in Isabelle. In *International Workshop on Types for Proofs and Programs*. Springer, 160–174.

Farzin Houshmand and Mohsen Lesani. 2019. Hamsaz: Replication Coordination Analysis and Synthesis. *Proc. ACM Program. Lang.* 3, POPL, Article 74 (Jan. 2019), 32 pages. https://doi.org/10.1145/3290387

Johan Jeuring, Patrik Jansson, and Cláudio Amaral. 2012. Testing type class laws. In *Proceedings of the 2012 Haskell Symposium*. 49–60.

M. Kleppmann and A. R. Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 2733–2746.

Leonidas Lampropoulos and Benjamin C. Pierce. 2018. *QuickChick: Property-Based Testing in Coq*.

K Rustan M Leino and Clément Pit-Claudel. 2016. Trigger selection strategies to stabilize program verifiers. In *International Conference on Computer Aided Verification*. Springer, 361–381.

Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hriţcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, et al. 2019. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *European Symposium on Programming*. 30–59.

Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Proc. Workshop on Parallel and Distributed Algorithms*, Cosnard M. et al. (Ed.). North-Holland / Elsevier, 215–226. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

Conor McBride. 2002. Faking it Simulating dependent types in Haskell. *Journal of functional programming* 12, 4-5 (2002), 375–392.

Andrey Mokhov. 2017. Algebraic Graphs with Class (Functional Pearl). In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*.

Kartik Nagar and Suresh Jagannathan. 2019. Automated Parameterized Verification of CRDTs. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 459–477.

Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. 2020. Proving the Safety of Highly-Available Distributed Objects. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 544–571.

Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. 2016. The CISE Tool: Proving Weakly-Consistent Applications Correct. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data* (London, United Kingdom) *(PaPoC '16)*. Association for Computing Machinery, New York, NY, USA, Article 2, 3 pages. https://doi.org/10.1145/2911151.2911160

Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2017. Simplicitly: Foundations and Applications of Implicit Function Types. *Proc. ACM Program. Lang.* 2, POPL, Article Article 42 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158130

Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. 2006. Data Consistency for P2P Collaborative Editing. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work* (Banff, Alberta, Canada) *(CSCW '06)*. Association for Computing Machinery, New York, NY, USA, 259–268. https://doi.org/10.1145/1180875.1180916

Bryan O'Sullivan. 2020. criterion: Robust, reliable performance measurement and analysis. https://hackage.haskell.org/package/criterion.

James Parker, Niki Vazou, and Michael Hicks. 2019. LWeb: Information flow security for multi-tier web applications. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

Simon Peyton Jones. 2003. *Haskell 98 Language and Libraries: The Revised Report*. Vol. 13.

Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. 2009. A Commutative Replicated Data Type for Cooperative Editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems (ICDCS '09)*. IEEE Computer Society, USA, 395–403. https://doi.org/10.1109/ICDCS.2009.20

Hyun-Gul Roh, Myeongjae Jeon, Jinsoo Kim, and Joonwon Lee. 2011. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.* 71, 3 (2011), 354–368. https://doi.org/10.1016/j.jpdc.2010.12.006

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

John Rushby, Sam Owre, and Natarajan Shankar. 1998. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering* 24, 9 (1998), 709–720.

Ryan G. Scott and Ryan R. Newton. 2019. Generic and Flexible Defaults for Verified, Law-Abiding Type-Class Instances. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell* (Berlin, Germany) *(Haskell 2019)*. Association for Computing Machinery, New York, NY, USA, 15–29. https://doi.org/10.1145/3331545.3342591

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011a. A comprehensive study of convergent and commutative replicated data types. (2011).

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011b. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400.

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. *SIGPLAN Not.* 37, 12 (Dec. 2002), 60–75. https://doi.org/10.1145/636517.636528

KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981

William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. 2012. Zeno: An Automated Prover for Properties of Recursive Data Structures. In *Tools and Algorithms for the Construction and Analysis of Systems*, Cormac Flanagan and Barbara König (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 407–421.

Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 278–293.

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is Reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 14–27. https://doi.org/10.1145/3167092

Bjarne Stroustrup. 1989. Multiple inheritance for C++. *Computing Systems* 2, 4 (1989), 367–395.

Martin Sulzmann, Manuel MT Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*. 53–66.

Team Mozilla Research. 2017. *The Rust Programming Language*. https://www.rust-lang.org/en-US/

Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. 2018. Theorem proving for all: equational reasoning in liquid Haskell (functional pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Nicolas Wu (Ed.).

Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. 269–282.

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.* 2, POPL, Article Article 53 (Dec. 2017), 31 pages. https://doi.org/10.1145/3158141

Dimitrios Vytiniotis, Simon L. Peyton Jones, Koen Claessen, and Dan Rosén. 2013. HALO: haskell to logic through denotational semantics. In *POPL*.

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 60–76.

Stephanie Weirich, Pritam Choudhury, Antoine Voizard, and Richard A. Eisenberg. 2019. A Role for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 3, ICFP, Article Article 101 (July 2019), 29 pages. https://doi.org/10.1145/3341705

Stephane Weiss, Pascal Urso, and Pascal Molli. 2009. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems (ICDCS '09)*. IEEE Computer Society, USA, 404–412. https://doi.org/10.1109/ICDCS.2009.75

Hongwei Xi and Frank Pfenning. 1998. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 249–257.

Ningning Xie, Richard A. Eisenberg, and Bruno C. d. S. Oliveira. 2019. Kind Inference for Datatypes. *Proc. ACM Program. Lang.* 4, POPL, Article Article 53 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371121

Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. 2014. Formal Specification and Verification of CRDTs. In *Formal Techniques for Distributed Objects, Components, and Systems*, Erika Ábrahám and Catuscia Palamidessi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 33–48.

A VERIFYING CRDT SEMANTICS

Just because a data type satisfies the required `VRDT` typeclass laws does not mean its implementation is correct. Fortunately, since the data type is defined in Liquid Haskell, it is possible to verify that its implementation matches an ideal semantics. To demonstrate this, we prove that the behavior of our

```
type DMultiSet a = (a → Integer)

toDenotation :: Ord a ⇒ MultiSet a → DMultiSet a
toDenotation (MultiSet p n) t | Just v ← Map.lookup t p = v
toDenotation (MultiSet p n) t | Just v ← Map.lookup t n = v
toDenotation _              _                           = 0

dApply :: Eq a ⇒ DMultiSet a → MultiSetOp a → DMultiSet a
dApply f (MultiSetOpAdd    v c) t = if t == v then f t + c else f t
dApply f (MultiSetOpRemove v c) t = if t == v then f t - c else f t

simulation :: x:MultiSet a → op:{MultiSetOp a | enabled x op} → t:a
           → {toDenotation (apply x op) t == dApply (toDenotation x) op t}
```

Fig. 8. Denotational semantics of `Multiset`.

`Multiset VRDT` (introduced in Section 4.2) simulates the mathematical (denotational) semantics of multisets.

Our implementation of `Multiset` maintains a *positive* map `p` and a *negative* map `n`; the former contains members with positive counts while the latter contains members with non-positive counts. The `Op`s are `MultiSetOpAdd` and `MultiSetOpRemove` ; they shift a value between maps as its count crosses 0. We define the denotation of a `MultiSet` to be a function from an element of the `Multiset` to the number of copies of that element. This is represented by the type alias `DMultiSet` in Figure 8. The `toDenotation` function is a straightforward mapping from a `MultiSet` to a `DMultiSet` that looks up the element in the positive and negative `Map`'s of the `MultiSet`. `dApply` defines how to run a `MultiSet` operation on a `DMultiSet` by adding the number of new copies to the existing count. The `DMultiSet` denotation serves as a simple specification of how we expect `Multiset` to operate. We prove that `Multiset` and `DMultiSet` have the same behavior: The `simulation` theorem states that for all `MultiSet`'s and enabled operations on that `MultiSet`, looking up an element when you apply the operation on the `MultiSet` and then convert it to its denotation returns the same result as when you first convert it to a `DMultiSet` and run the operation on the denotation.

B MESSAGE DELIVERY LIBRARY

The message delivery library consists of a client and a server which interface over HTTP and a client API which abstracts network interruptions away from the application.

The client API entry point is `runSer`. Applications pass in a function to receive messages, and `runSer` produces a function with which the application may send messages. A client is identified with a random UUID generated before its initial connection to a server. A client will not receive any messages from a server that were sent by a client with the same UUID.

The client API handles disconnects transparently. Messages passed from the application to the Client API via `Send a` are buffered in a transactional memory queue from which a thread removes one, attempts to send it, and if unsuccessful returns it to the queue. On a separate thread a stream of messages is received from the server and passed back to the application via `Recv a` until a disconnection occurs. When a send attempt is unsuccessful or the receiving thread disconnects, exponential backoff is employed to space out retries until they are every five minutes or until they are successful.

The server and client interface which each other via HTTP calls expressed using the `servant` library. There is a *send* call and a *listen* call. *Send call:* A client sends a message to a server with

```
newtype Recv a = Recv (a → IO ())
newtype Send a = Send (a → IO ())
runSer
    :: Serialize u
   ⇒ Server -- URL of a server
    → StoreId -- Namespace for all server interaction
    → ClientId -- Random UUID to prevent receipt of own messages
    → Recv (Either String u) -- Function to receive broadcast messages
    → IO (Async (), Send u)
    -- ˆ Handle for background threads; Function to send broadcast messages
```

Fig. 9. A conceptually accurate, but simplified, signature for the client API entry point.

an HTTP Post request which completes immediately to indicate that the server has received the message. A server stores each incoming message in a log and also broadcasts it to currently connected clients. *Listen call:* A client receives a stream of messages from a server with a long-polling HTTP Post request which remains connected indefinitely. A server responds with the contents of its log to a newly connected client and then responds with any live broadcast messages.