Deriving Law-Abiding Instances

Ryan Scott
Vikraman Choudhury
Ryan Newton
Indiana University
{rgscott,vikraman,rrnewton}@indiana.edu

Niki Vazou
University of Maryland
nvazou@cs.umd.edu

Ranjit Jhala
UC San Diego
jhala@cs.ucsd.edu

Abstract

Liquid Haskell augments the Haskell language with theorems proving capabilities, allowing programmers to express and prove class laws. But many of these proofs require routine, boilerplate code and do not scale well, as the size of proof terms can grow superlinearly with the size of the datatypes involved in the proofs.

We present a technique to derive Haskell proof terms by leveraging datatype-generic programming. Our observation is that we can take any algebraic datatype, generate an equivalent representation type, and have Liquid Haskell automatically construct (and prove) an isomorphism between the original type and the representation type. This reduces many proofs down to easy theorems over simple algebraic “building block” types, allowing programmers to write generic proofs cheaply and cheerfully. We applied our technique to derive verified instances of the Eq, Ord, Semigroup, Monoid and Functor Haskell classes for commonly used algebraic datatypes.

ACM Reference format:

1 Introduction

Many widely used type classes abstract over operators that must obey algebraic laws. With Liquid Haskell [14], these type class laws can be encoded as refinement type specifications. For instance, \texttt{TotalOrd} extends the \texttt{Ord} class with the \texttt{total} method that encodes the proof obligation that \((\leq)\) should be total:

\begin{verbatim}
(-@ class Ord a ⇒ TotalOrd a where
  total :: x:a → y:a → (x ≤ y || y ≤ x) @-)
\end{verbatim}

The type specification of \texttt{total}, defined in the special Liquid Haskell comments \((-@ \ldots \)@-\), states that for all values \(x\) and \(y\) there exists a proof that \(x \leq y\) or \(y \leq x\), thus encoding the totality of \((\leq)\). Users of \texttt{TotalOrd} can rest assured that \((\leq)\) is indeed total, but when defining an instance of \texttt{TotalOrd}, a proof of totality must be provided.

\textit{Haskell} programs can be used to encode such proofs [12, 13]. Yet, proof deployment can be tedious. Implementing many proofs can involve excessive amounts of boilerplate code. Even worse, the size of some proofs can grow superlinearly in the size of the data type used, as the proofs can grow extremely quickly due to the sheer number of cases one has to exhaust (§2.4).

In this paper, we set out to minimize this boilerplate and develop a style of proof-carrying programming that scales well as the size of a data type grows. To do so, we adapt a style of datatype-generic programming in the tradition of the Glasgow Haskell Compiler’s \texttt{GHC.Generics} module 1. That is to say, for some data type about which we want to prove a property, we first consider a \texttt{representation type} which is isomorphic to the original data type. This representation type is the composition of several very small data types. By proving the property in question for these small, representational data types, we can compose these proofs and use them to prove the property for the original data type by taking advantage of the isomorphism between the original and representation types.

To use \texttt{TotalOrd} as an example of how this would be accomplished, the author of the \texttt{TotalOrd} class would need to implement (1) definitions for total orderings on the generic representation types, and (2) a way to derive a total ordering for a type \(a\), reusing a proof from its representation type (which is provably isomorphic to \(a\)):

\begin{verbatim}
instance (TotalOrd (Rep a), GenericIso a) ⇒ TotalOrd a where
\end{verbatim}

With this generic derivation in hand, Haskell’s standard class resolution will derive the proper (provably correct) \texttt{TotalOrd} instance for any type that is an instance of \texttt{GenericIso}, a class which carries the proof of isomorphism. We can automate this process of deriving law-abiding instances further by defining a Template Haskell function \texttt{deriveIso} which derives the \texttt{GenericIso} instances with minimal effort. For instance, one can derive a provably total \texttt{Ord} instance of the user-defined data type \texttt{Nat} with just:

\begin{verbatim}
data Nat = Zero | Succ Nat
\end{verbatim}

\footnote{\url{http://hackage.haskell.org/package/base-4.9.1.0/docs/GHC-Generics.html}}
We start with an overview of our approach for deriving class
refinements and allow proofs about Haskell code written as
an explicit proof of the property. This is the premise of
reification reflection [12], a recent addition to Liquid Haskell.
Using this approach, Liquid Haskell lifts Haskell definitions
into the logic, leaving them initially uninterpreted, but un-
folding their definitions once every time they are referenced
in an explicit proof of the property.
Thus Liquid Haskell goes beyond automatically-checked
refinements and allows proofs about Haskell code written as
Haskell code. In these proofs, Haskell’s arrow type encodes
implication, Haskell branches encode proof case-splits, and
recursion encodes induction. Together with a library of
proof combinators included with Liquid Haskell, these enable
proofs that are similar to their pencil-and-paper analogues.
We will see examples of such proofs as we proceed in this
paper.

2.2 Specifying Law-Abiding Classes

Classes  Recall the following simplified definition of the Eq
and Ord type classes that provide abstractions for datatypes
which support equality and ordering checks:

\[
\text{class Eq } a \text{ where } \\
\quad (\equiv) : a \rightarrow a \rightarrow \text{Bool}
\]

\[
\text{class Ord } a \Rightarrow \text{Ord } a \text{ where } \\
\quad (\leq) : a \rightarrow a \rightarrow \text{Bool}
\]

Laws  Typically, we require that any instance of Ord is a
total order that satisfies the following laws:

\[
\begin{align*}
\text{Reflexivity} & \quad \forall x. x \leq x \\
\text{Totality} & \quad \forall x, y. x \leq y \lor y \leq x \\
\text{Antisymmetry} & \quad \forall x, y. x \leq y \land y \leq x \Rightarrow x = y \\
\text{Transitivity} & \quad \forall x, y, z. x \leq y \land y \leq z \Rightarrow x \leq z
\end{align*}
\]

Specifying Laws as Refinement Types  We can encode
the above laws as refined function types:

\[
\begin{align*}
\text{type Refl } a & = x : a \rightarrow \{ x \leq x \} \\
\text{type Total } a & = x : a \rightarrow y : a \rightarrow \{ x \leq y \lor y \leq x \} \\
\text{type Anti } a & = x : a \rightarrow y : a \rightarrow \{ x \leq y \land y \leq x \Rightarrow x = y \} \\
\text{type Trans } a & = x : a \rightarrow y : a \rightarrow z : a \rightarrow \{ x \leq y \land y \leq z \Rightarrow x \leq z \}
\end{align*}
\]

In Liquid Haskell, these type refinements must be written
inside a special comment, recognized by Liquid Haskell and
separated from the plain Haskell types. We show only the
Liquid Haskell type signatures above for brevity. We write
\( (\equiv) \) to abbreviate \((v : \text{Proof} (\rho)) \), that is, the set of values of
type Proof such that the predicate \( \rho \) holds. 2 Refinement
type checking [12] ensures that any inhabitant of Refl \( a \)
(and respectively, Total \( a \), Anti \( a \)) is a concrete
proof that the corresponding law holds for the type \( a \), by
demonstrating that the law holds for all (input) values of
type \( a \).

Specifying Law-Abiding Classes  We can specify law-abiding
classes by extending the Ord class to a VerifiedOrd subclass
with four more fields that must be inhabited by proofs that
demonstrate that the corresponding laws hold for the in-
stance:

\[
\begin{align*}
\text{class Ord } a \Rightarrow \text{VerifiedOrd } a \text{ where } \\
\quad \text{refl} :: \text{Refl } a \\
\quad \text{total} :: \text{Total } a
\end{align*}
\]

2 Here, \( \text{Proof} \) is simply a type alias for the unit type (\( () \)) in Liquid Haskell’s
library of proof combinators. Since the proofs carry no useful information
at runtime, the unit type suffices as a runtime witness to a proof.
2.3 Law-Abiding Instances: The Direct Approach

Next, let’s create a `VerifiedOrd` instance for a simple data type:

```haskell
data a = Int deriving Eq
instance Ord a where
  (A s1) <= (A s2) = (s1 <= s2)

The reflexivity of `A` can be proved with proof combinators like so:

```haskell
reflA :: Refl A
reflA x@(A s) = x <= x
  =. s <= s
  ** QED
```

The implementation of `reflA` is a function that shows that the re/flexivity law holds for every `x :: A`. The function uses the proof combinators to build refinement proofs in “equational reasoning” style.

Note that the key step for the proof of `reflA` is the line `x <= x`. The underlying SMT solver knows how to reason about `Ints` directly, so Liquid Haskell is able to conclude that `x <= x` for all `Ints x`, without requiring any lemmas about `Int` arithmetic.

We can prove antisymmetry, transitivity and totality for `A` in much the same way as we did for reflexivity:

```haskell
antiA :: Anti A
antiA x@(A s1) y@(A s2) = (x <= y && y <= x)
  =. (s1 <= s2 && s2 <= s1)
  =. (s1 == s2)
  =. (x == y)
  ** QED
```

```haskell
transA :: Trans A
transA x@(A s1) y@(A s2) z@(A s3) = (x <= y && y <= z)
  =. (s1 <= s2 && s2 <= s1)
  =. (s1 <= s3)
  =. (s1 <= s3)
  ** QED
```

2.4 Scaling Up the Direct Approach

Next, let’s see how to repeat the process of writing a `VerifiedOrd` instance for a more complicated data type. We shall see that while this is possible, the proofs quickly start to become unpleasant, as they will require a lot of boilerplate code. To see this, consider a data type with two constructors:

```haskell
data b = Int | Range Int
deriving Eq
instance Ord b where
  (B s1) <= (B s2) = (s1 <= s2)
```

The proof of re/flexivity does not change significantly, as it amounts to adding another case for the additional constructor:

```haskell
reflB :: Refl B
reflB x@(B s) = x <= x
  =. s <= s
  ** QED
```

The proof of antisymmetry becomes a bit more complicated. We now require a case for every pairwise combination of constructors:

```haskell
antiB :: Anti B
antiB x@(B s1) y@(B s2) = (x <= y && y <= x)
  =. (s1 <= s2 && s2 <= s1)
  =. (s2 == s1)
  =. (x == y)
  ** QED
```

The proof of antisymmetry, however, becomes a bit more complicated. We now require a case for every pairwise combination of constructors:
With a proof-reuse technique.

false will end up being a variant of the proof of transitivity has one case, the two-value with a proof.

Glasgow Haskell Compiler (GHC).

Having seen the tedium of manually constructing proofs, we can include that the two values are equal. For the cases where the constructor variant would have eight cases, and a three-constructors. The proof of transitivity has an even more noticeable increase in size growth, since it must match on every pairwise combination of two constructors.

The other proofs needed for Verified Ord also grow quickly. Like antisymmetry, the proof of totality grows quadratically, since it must consider every pairwise combination of two constructors. The proof of transitivity has an even more noticeable increase in size growth, since it must match on every combination of three B values: while the one-constructor variant of the proof of transitivity has one case, the two-constructors would have eight cases, and a three-constructors would have 27 cases.

Perhaps even more troublesome than the size of these proofs themselves is the fact that most of these cases are sheer boilerplate. For instance, the proof of antisymmetry follows a predictable pattern. For the cases where the constructors are both the same, we compare the fields of the constructors, appeal to properties of Int arithmetic, and conclude that the two values are equal. For the cases where different constructors are being matched, one comparison will end up being False, causing the whole hypothesis to be False. This is routine code that is begging to be automated with a proof-reuse technique.

3 Deriving Law-Abiding Instances

Having seen the tedium of manually constructing proofs, we present a solution. Notably, our approach does not require adding new features to Liquid Haskell itself—instead, we use a technique based on extensions already found in the Glasgow Haskell Compiler (GHC).

We adapt an approach from the datatype-generic programming literature where we take an algebraic data type and construct a representation type which is isomorphic to it [8]. The representation type itself is a composition of small data types which represent primitive notions such as single constructors, products, sums, and fields. We also establish a type class for witnessing the isomorphism between a data type and its representation type.

With these tools, we can shift the burden of proof from the original data type (which may be arbitrarily complex) to the handful of simple data types which make up representation types. Moreover, since all Haskell 2010 data types can be expressed in terms of these representational building blocks, proving a property for these data types is enough to prove the property for this whole class of algebraic data types.

3.1 A Primer on Datatype-Generic Programming

To build up representation types, we build upon the API from the GHC.Generics module [8]. First, we utilize a type class which captures the notion of conversion to and from a representation type:

```
class Generic a where
type Rep a :: * -> *
from :: a -> Rep a x
to :: Rep a x -> a
```

The Rep type itself will always be some combination of the following data types:

- `data U1 p = U1. This is used to represent a constructor with no fields.
- `newtype Rec0 c p = Rec0 c. This is used to represent a single field in a constructor.
- `data (f :+: g) p = (f p) :+: (g p). This is used to represent the choice between two consecutive fields in a constructor.
- `data (f :+: g) p = L1 (f p) | R1 (g p). This is used to represent the choice between two consecutive constructors in a data type.

Recalling the B data type from earlier:

```
data B = B1 Int | B2 Int
```

We define its canonical Generic instance like so:

```
instance Generic B where
type Rep B = Rec0 Int :+: Rec0 Int
from (B1 i) = L1 (Rec0 i)
from (B2 i) = R1 (Rec0 i)
to (L1 (Rec0 1)) = B1 i
to (R1 (Rec0 1)) = B2 i
```

[3]The actual implementation features another data type, M1, which is used only for metadata. For the sake of simplicity, we have left it out of the discussion in this paper.
Deriving Law-Abiding Instances

Here, we see that because B has two constructors (B1 and B2), the (::*:) type is used once to represent the choice between B1 and B2. The Int field of each constructor is likewise represented with a Rec type. We call this instance “canonical” because with GHC’s DeriveGeneric extension, this instance is generated automatically with only this line of code:

```
 deriving instance Generic B
```

It should be emphasized that the four types U1, Rec0, (::*:), and (::*:) are enough to represent any Haskell 2010 \(^4\) data type. For instance, if one were to add more fields to the B1 constructor, then its corresponding Rep type would change by adding additional occurrences of (::*:) for each field. Therefore, these four data types conveniently provide a unified way to describe the structure of any data type, a property which will be useful shortly.

While Generic is convenient for quickly coming up with representation types, it alone isn’t enough for our needs, as we need to be able to use the proof that the from and to functions form an isomorphism. In pursuit of that goal, we define a subclass of Generic with two proof methods that express the fact that from and to are mutual inverses.

```
class Generic a ⇒ GenericIso a where
  tof :: x:a → { to (from x) == x }
  fof :: x:Rep a x → { from (to x) == x }
```

To demonstrate how the proofs in a GenericIso instance look, we give an example instance for B:

```
instance GenericIso B where
  tof x@(B1 i) = to (from x) =. to (L1 (Rec0 i))
  tof x@(B2 i) = to (from x) =. to (R1 (Rec0 i))
  fof x@L1 (Rec0 i) = from (to x) =. from (B1 i)
  fof x@R1 (Rec0 i) = from (to x) =. from (B2 i)
```

Unlike Generic, there is no built-in GHC mechanism for deriving instances of GenericIso, so one might reasonably worry that GenericIso is itself a source of boilerplate. We use Template Haskell [11] to mimic GHC’s deriving mechanism and automatically derive GenericIso instances. Concretely, we define the Template Haskell function deriveIso that,

given a name of a type constructor, derives the declarations of the corresponding instances of Generic and GenericIso.

```
deriveIso :: Name → Q [Dec]
```

As a demonstration, all of the code for the Generic and GenericIso instances for B written earlier in this section can be reduced to:

```
data B = B1 Int | B2 Int
deriveIso "B"
```

where “B” is the Template Haskell Name that represents the type constructor B.

3.2 Proofs over Representation Types

Having identified the four basic data types which can be composed in various ways to form representation types, the next task is to write proofs for these four types. We will do so by continuing our earlier VerifiedOrd example from Section 2, and in the process show how one can obtain a valid total ordering for any algebraic data type by using this technique.

The U1 data type has an extremely simple Ord instance:

```
instance Ord (U1 p) where
  U1 ≤ U1 = True
```

The VerifiedOrd instance is similarly straightforward, so we will elide the details here.

The Ord instance for the Rec0 type will look familiar:

```
instance Ord c ⇒ Ord (Rec0 c p) where
  (Rec0 r1) ≤ (Rec0 r2) = (r1 ≤ r2)
```

This is essentially the same Ord instance that we used for A in Section 2.3, except abstracted to an arbitrary field of type c. The VerifiedOrd instance for Rec0 also mirrors that of A, so we will also leave out the details here.

The (::*:) type, which serves the role of representing two fields in a constructor, is also the simplest possible product type, with two conjuncts. We can enforce a valid total order on such a type by using the lexicographic ordering. We first check if the left fields are equal. If so, we compare the right fields. Otherwise, we return the comparison on the left fields:

```
instance Ord (f p), Ord (g p)) ⇒
  Ord ((f :*: g) p) where
  (x1 :*: y1) ≤ (x2 :*: y2) =
    if x1 == x2 then y1 ≤ y2 else x1 ≤ x2
```

It can be shown that given suitable VerifiedOrd proofs for the fields’ types f p and g p, this ordering for (::*:) is reflexive:

```
leqProdRef1 :: (VerifiedOrd (f p), VerifiedOrd (g p))
  ⇒ t:(f :*: g) p → { t ≤ t }
```

5There are many possible orderings on products, but only lexicographic ordering preserves the total order properties.
\[
\text{leqProdRef1 } t @ (x :* y) = \\
(t \leq t) \\
= (\text{if } x == x \text{ then } y \leq y \text{ else } x \leq x) \\
= y \leq y \\
= \text{True } \vdash \text{refl y} \\
\text{** QED}
\]

Note that we use an additional proof combinator \(\vdash\) here:

\(\vdash : (\text{Proof } \rightarrow a) \rightarrow \text{Proof } \rightarrow a\)

\(f : y = f y\)

One should read \(\vdash\) as being "prove the equational step on the left-hand side by using the lemma on the right-hand side". In the case of leqProdRef1, we were able to prove that \(y \leq y\) is true precisely because of the assumption that \(y\) was reflexive. The remaining proofs of antisymmetry, transitivity, and totality for \(\vdash\) can be found in Appendix A.1. Putting all of these proofs together gives us the following VerifiedOrd instance:

\[
\text{instance } (\text{VerifiedOrd } (f p), \text{VerifiedOrd } (g p)) \\
\Rightarrow \text{VerifiedOrd } ((f :*: g) p) \text{ where}
\]

\[
\text{refl } = \text{leqProdRef1} \\
\text{antisym } = \text{leqProdAntisym} \\
\text{trans } = \text{leqProdTrans} \\
\text{total } = \text{leqProdTotal}
\]

In a similar vein, we can come up with a VerifiedOrd instance for the \(\vdash\) type. \(\vdash\) not only represents choice between two constructors, it is also the simplest possible sum type, with two disjuncts. A total ordering on sums is defined so that everything in the \(L1\) constructor is less than everything in the \(R1\) constructor:

\[
\text{instance } (\text{Ord } (f p), \text{Ord } (g p)) \Rightarrow \\
\text{Ord } ((f :*: g) p) \text{ where}
\]

\[
(L1 x) \leq (L1 y) = \; x \leq y \\
(L1 x) \leq (R1 y) = \; \text{True} \\
(R1 x) \leq (L1 y) = \; \text{False} \\
(R1 x) \leq (R1 y) = \; x \leq y
\]

Here is an example of a VerifiedOrd-related proof for \(\vdash\), establishing reflexivity:

\[
\text{leqSumRef1 } : (\text{VerifiedOrd } (f p), \text{VerifiedOrd } (g p)) \\
\Rightarrow u : ((f :*: g) p) \rightarrow \{ u \leq u \}
\]

\[
\text{leqSumRef1 } s @ (L1 x) = \; (s \leq s) \\
\Rightarrow x \leq x \\
\Rightarrow \text{True } \vdash \text{refl x} \\
\text{** QED}
\]

\[
\text{leqSumRef1 } s @ (R1 y) = \; (s \leq s) \\
\Rightarrow y \leq y \\
\Rightarrow \text{True } \vdash \text{refl y} \\
\text{** QED}
\]

This proof bears a strong resemblance to the reflexivity proof for \(\vdash\) in Section 2.3. This similarity is intended, as the structure of the \(B\) data type is quite similar to that of \((\vdash)\). The remaining proofs for \((\vdash)\) can be found in Appendix A.2. Finally, we obtain the following VerifiedOrd instance for \((\vdash)\):

\[
\text{instance } (\text{VerifiedOrd } (f p), \text{VerifiedOrd } (g p)) \\
\Rightarrow \text{VerifiedOrd } ((f :*: g) p) \text{ where}
\]

\[
\text{refl } = \text{leqSumRef1} \\
\text{antisym } = \text{leqSumAntisym} \\
\text{trans } = \text{leqSumTrans} \\
\text{total } = \text{leqSumTotal}
\]

We wish to place particular emphasis on the fact that these VerifiedOrd instances are compositional. That is, we can put together whatever combination of \((\vdash), (:*), U1,\) and \(\text{Rec}\) we wish, and we will ultimately end up with a structure which has a valid VerifiedOrd instance. This is crucial, as it ensures that this technique scales up to real-world data types.

### 3.3 Reusing Proofs

Given a VerifiedOrd instance for a representation type, how can we relate it back to the original data type to which it is isomorphic? The answer lies in the GenericIso class from before. GenericIso has enough power to take a VerifiedOrd proof for one type and reuse it for another type.

To begin, we will need a way to compare two values of a type that is an instance of Generic, given that its representation type \(\text{Rep}\) is an instance of Ord:

\[
\text{leqIso } : (\text{Ord } (\text{Rep } a x), \text{Generic } a) \\
\Rightarrow (a \rightarrow a \rightarrow \text{Bool})
\]

\[
\text{leqIso } x \; y = (\text{from } x) \leq (\text{from } y)
\]

We can straightforwardly prove that leqIso is a total order:

\[
\text{leqIsoRef1 } \\
\vdash (\text{VerifiedOrd } (\text{Rep } a x), \text{GenericIso } a) \\
\Rightarrow x : a \rightarrow (\text{leqIso } x x) \\
\text{leqIsoRef1 } x = \text{leqIso } x x \\
\Rightarrow . \; (\text{from } x) \leq (\text{from } x)
\]

\[
\Rightarrow \; \text{True } \vdash \; \text{refl } (\text{from } x) \\
\text{** QED}
\]

The proof of antisymmetry relies on the fact that \(\text{from}\) is an injection, which follows from the proof of isomorphism.

\[
\text{fromInj } :: \text{GenericIso } a \Rightarrow x : a \rightarrow y : a \\
\Rightarrow \{ \; \text{from } x \; \Rightarrow \; x = y \}
\]

\[
\text{fromInj } x \; y = \\
\Rightarrow \; \text{from } x \; \Rightarrow \; y \\
\text{. \; to } (\text{from } x) \; \Rightarrow \; (\text{from } y)
\]

\[
\Rightarrow \; \text{to } (\text{from } y) \; : \; \text{to } x \\
\Rightarrow \; x = y \; \Rightarrow \; \text{to } y
\]

\[
\text{** QED}
\]

\[
\text{leqIsoAntisym } \\
\vdash (\text{VerifiedOrd } (\text{Rep } a x), \text{GenericIso } a) \\
\Rightarrow x : a \rightarrow y : a
\]
Deriving Law-Abiding Instances

With the above machinery, writing a verifiedord instance becomes a breeze. We can now rewrite the earlier verifiedord B instance, which was written in the direct approach, and greatly simplify it using the generic approach:

```
data Nat = Zero | Suc Nat deriving Eq

instance Ord (Rep a x) where
  (≤) = leqIso

instance (VerifiedOrd (Rep a x), GenericIso a) => Ord a where
  (≤) = leqIso
```

This small amount of code does a tremendous amount of heavy lifting. Recall (§ 3.1) for Generic and GenericIso:

```
instance Generic B where
  type Rep B = Rec0 Int :+: Rec0 Int ... 
```

Our derivation technique, as presented, works for recursive datatypes too. For instance assume the recursive definition of natural numbers:

```
data Nat = Zero | Suc Nat deriving Eq
```

Then we derive a VerifiedOrd instance for Nat simply by deriving all the appropriate Generic, GenericIso and Ord classes:

```
deriveIso 'Nat 
instance Ord Nat 
instance VerifiedOrd Nat 
```

4 Evaluation

To evaluate our approach for deriving lawful instances, we extended a set of commonly used Haskell type classes with associated proof obligations (summarized in Table 1) and implemented proof carrying instances for the Haskell data types of Table 2. Our implementation can be accessed at [http://bit.ly/2qFbe16](http://bit.ly/2qFbe16). In this section, we describe the five lawful type classes (Section 4.1) and the law-abiding instances that we derived for them (Section 4.2). We conclude by summarizing the benefits (Section 4.3) and limitations (Sections 4.4 and 4.5) of our technique.

4.1 Lawful Type Classes

We used refinement types to specify the laws for five standard type classes as presented in Table 1.

1. Total Orders Our primary example from Section 2 was the Ord type class, which can be verified to be a total order.

2. Equivalences Next we specify the equivalence properties in Ord’s superclass, Eq.

```
class Eq a where
  (=) :: a -> a -> Bool
```

Note that the methods in the derived instance are only guaranteed to terminate for strictly positive datatypes.
class Eq a ⇒ VerifiedEq a where
  refl :: ReflEq a
  sym :: SymEq a
  trans :: TransEq a

class Ord a ⇒ VerifiedOrd a where
  refl :: Refl a
  total :: Total a
  anti :: Anti a
  trans :: Trans a

class Semigroup a ⇒ VerifiedSemigroup a where
  assoc :: Assoc a

class Monoid a ⇒ VerifiedMonoid a where
  lident :: LIdent a
  rident :: RIdent a

class Functor f ⇒ VerifiedFunctor f where
  fmapId :: fmapId f
  fmapCompose :: fmapCompose f

Table 1. Summary of the law-abiding type classes.

data Identity a = Identity a
data Maybe a = Nothing | Just a
data Either a b = L a | R b
data List a = Nil | Cons a (List a)
data Triple a b c = MkTriple a b c

Table 2. Summary of the evaluated data-types.

Equality should be an equivalence relation—that is, it should satisfy the laws of reflexivity, symmetry, and transitivity (expressed directly as refined function types):

type ReflEq a = x:a → {x == x}
type SymEq a = x:a → y:a → {x == y ⇒ y == x}
type TransEq a = x:a → y:a → z:a
  → {x == y ∧ y == z ⇒ x == z}

These type signatures are used in the class methods of VerifiedEq in Table 1. The process for generically creating VerifiedEq instances is extremely similar to the process for VerifiedOrd, as outlined in Section 2.

3. Semigroups  Next, we specify the associativity law for semigroups. The Semigroup class comes equipped with a binary operation (<>), that provides a way to combine two values into one.

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

The proof obligation for (<>), that is it is associative:

type Assoc a = x:a → y:a → z:a
  → {x <> (y <> z) = (x <> y) <> z}

The process of generically creating VerifiedSemigroup instances slightly differs from that of VerifiedOrd (from Section 2), since Semigroup features a class method with the type parameter in the result position of a function—that is, the type parameter is used covariantly as well as contravariantly. This means that in order to turn a VerifiedSemigroup a instance into a VerifiedSemigroup b instance with GenericISO, one must use the to function—which was unused up to this point—as well as from.

4. Monoids  On top of Semigroup, its subclass Monoid grants the ability to conjure up an identity element:

class Semigroup a ⇒ Monoid a where
  empty :: a

Monoid has two more proof obligations which dictate how empty should interact with the (<>) operation. empty acts as the left and right identity element:

type LIdent a = x:a → { empty <> x = x }
type RIdent a = x:a → { x <> empty = x }

There is an interesting question to be asked about whether one can sensibly write generic Semigroup or Monoid instances for sum types. Unlike the Eq or Ord classes, where it is straightforward to implement generic instances for types with multiple constructors (represented by the type (:+:+)), for Semigroup and Monoid the choice is not clear. Trying to combine values from different constructors with (<>) would require arbitrarily picking whether the left or right constructor should be used, for instance. As a result, we did not pursue any VerifiedSemigroup or VerifiedMonoid instances for sum types.

5. Functors  Finally we specify the laws on the Functor class:

class Functor (f :: * → *) where
  fmap :: (a → b) → f a → f b

We use the standard Haskell definitions for identity and composition:

type id :: a → a
  id z = z

type (.) :: (b → c) → (a → b) → a → c
  (.) f g x = f (g x)

to specify that functors preserve identity and composition:

type fmapId f
  = z:(f a) → {fmap id z = z}
type fmapCompose f
  = x:(b → c) → y:(a → b) → z:(f a)
Deriving Law-Abiding Instances

Unlike the previous four classes that are defined over types (of kind \((\ast \rightarrow \ast)\), Functor is defined over type constructors (of kind \((\ast \rightarrow \ast))\). To derive law-abiding instances over these kinds of classes, we need to generalize our earlier machinery to work over \((\ast \rightarrow \ast))\)-kinded types.

Generic Derivations for Type Constructors. The Generic1 class handles \((\ast \rightarrow \ast))\)-kinded types.

class Generic1 (f :: \(\ast \rightarrow \ast\)) where
type Rep1 f :: \(\ast \rightarrow \ast\)
from1 :: \(\forall \ast. f \ast \rightarrow \{
it{to1} \ast \rightarrow \{

to1 :: \(\forall \ast. f \ast \rightarrow \{

The GenericIso1 class extends Generic1, expressing that to1 and from1 form a natural isomorphism.

class Generic1 f \Rightarrow GenericIso1 (f :: \(\ast \rightarrow \ast\)) where
to1 :: \(\forall \ast. f \ast \rightarrow \{
from1 :: \(\forall \ast. f \ast \rightarrow \{

Next, it is necessary to increase our set of representational data types slightly, since implementing Functor demands that we ask more interesting questions about the structure of data types. To see why that is the case, observe this data type’s Functor instance:

newtype Phantom a = Phantom Int

instance Functor Phantom where
  fmap f (Phantom i) = Phantom i

This is different than the Functor instance for this very similar data type:

newtype Identity a = Identity a
instance Functor Identity where
  fmap f (Identity x) = Identity (f x)

The only distinction between the internal structure of Phantom and Identity is that Identity’s field is an occurrence of its type parameter. In order to query this property generically, we need additional data types that mark occurrences of the type parameter:

newtype Par1 p = Par1 p
newtype Rec1 f p = Rec1 (f p)
newtype (f :: \(g\)) p = Comp1 (f (g p))

These three types are used in conjunction with Generic1 exclusively. To see how they are used, here is a sample Generic1 instance:

data T a = MKT Int a (Maybe a) [[a]]

instance Generic1 T where
  type Rep1 T =
    Rec0 Int :: Par1 ::: Rec1 Maybe
    ::: (Rec0 Int ::: Rec1 [])[a]
    from1 (T a1 a2 a3 a4) =
      Rec0 a1 :: Par1 a2 ::: Rec1 a3
    ::: Comp1 (fmap Rec1 a4)
    to1 (Rec0 a1 :: Par1 a2 ::: Rec1 a3

4.2 Law-Abiding Instances

We used our approach to derive law-abiding instances of the above type classes for data types of Identity, Maybe, Either, List, and Triple as defined in Table 2. As discussed in Section 4.1, we do not attempt to derive Semigroup and Monoid instances for the sum types Maybe, Either, and List. We selected the five data types in Table 2 because they provide a healthy variety of structure, encompassing types with products, sums and nullary constructors. Moreover, they provide interesting test cases for VerifiedFunctor as, e.g., the List type features the type parameter in both a direct occurrence and underneath the List type constructor (in the Cons constructor).

To recap the advantage of our approach, we describe how each instance was verified, using the VerifiedFunctor instance for List as an example.

At the library site, the developer defines the verified class together with its laws:

type FmapId f = \(\forall \ast. z : (f \ast) \rightarrow (fmap id z = z)\)
type FmapCompose f
  = \(\forall \ast \beta \gamma. x : (\beta \rightarrow \gamma) \rightarrow y : (f \ast) \rightarrow z : (f \ast) \rightarrow (fmap (\ast \rightarrow \ast) \rightarrow (fmap x . fmap y) z)\)

class Functor f \Rightarrow VerifiedFunctor f where
  fmapId :: FmapId f
  fmapCompose :: FmapCompose f

To allow semi-automatic derivation of law-abiding instances, the library developer needs to provide two further pieces of code:

1. the verified instances for the representation types needed to support the original data type, and
2. a way to convert a verified instance for the representation type back to the original data type.

Code 1. In our example, the library-writer must create VerifiedFunctor instances for the U1, Par1, Par1, (:+:), and Rec1 types. These instances will be used to derive the VerifiedFunctor instance for List since it has the following representation type:
We would like to emphasize the differences between our generic approach and the direct approach of writing out the proofs directly. In the direct approach, the library writer does not need to write anything that resembles Code 2, since there are no data type conversions to be found. In this sense, there is a cost to the generic approach that is not present in the direct approach. Importantly, though, this cost only has to be paid once for each class, because this code for converting VerifiedFunctor instances between types can be reused for every subsequent data type that needs a VerifiedFunctor instance.

Additionally, the direct approach’s costs significantly outweigh the generic approach’s costs. To implement Code 1 in the generic approach, one must write proof code for a certain number of “building block” data types, but no more than that. After these proofs have been written, there are no additional costs that arise later when writing other verified instances, as these proofs can be reused for other datatypes that have representation types with the same underlying building block types. In contrast, the direct approach requires writing (and re-writing) proof code for every verified instance.

### 4.3 Proof Burden for Direct and Derived Instances

We would like to emphasize the differences between our generic derivation approach and the direct approach of writing out the proofs directly. In the direct approach, the library writer does not need to write anything that resembles Code 2, since there are no data type conversions to be found. In this sense, there is a cost to the generic approach that is not present in the direct approach. Importantly, though, this cost only has to be paid once for each class, because this code for converting VerifiedFunctor instances between types can be reused for every subsequent data type that needs a VerifiedFunctor instance.

Additionally, the direct approach’s costs significantly outweigh the generic approach’s costs. To implement Code 1 in the generic approach, one must write proof code for a certain number of “building block” data types, but no more than that. After these proofs have been written, there are no additional costs that arise later when writing other verified instances, as these proofs can be reused for other datatypes that have representation types with the same underlying building block types. In contrast, the direct approach requires writing (and re-writing) proof code for every verified instance.

### 4.4 Limitations and Future Work

Our current prototype differs from the presentation in Section 3 in a couple of ways.

#### Liquid Haskell Doesn’t Support Type Classes

First, Liquid Haskell does not fully support refining all features of type classes of the time of writing. This is a limitation which could be overcome with a future implementation. We work around this in our prototype by using an explicit dictionary style that is equivalent to how type classes are desugared internally in GHC. For instance, we reify the Eq type class as

```
data Eq a = Eq (==) :: a -> a -> Bool
```

We then explicitly pass around Eq “instances” as data type values. This makes the implementation a bit more verbose, but is otherwise functionally equivalent to our presentation earlier in the paper.

#### Template Haskell Doesn’t Support Comments

The other limitation which our prototype must work around is the lack of Template Haskell support for generating comments. Recall that Liquid Haskell refinements are expressed in comments of the form `{M@ NNN @N}`. This poses a challenge for us, as we use Template Haskell to implement the deriveIso function, which is intended to create GenericIso instances and the associated refinement-containing comments that accompany the instances. That is, ideally

```
data Foo = Foo
deriveIso 'Foo
```

would suffice to generate the following Haskell code:

```
instance Generic Foo where
  to = ...
  from = ...

instance GenericIso Foo where
{-@ tof :: x:Foo -> {to (from x) == x} @-}
tof = ...
{-@ fot :: x:Rep Foo x -> {from (to x) == x} @-}
fot = ...
```

Unfortunately, Template Haskell currently does not support splicing in declarations that contain comments as in the code above, so doing everything in one fell swoop is not possible at the moment. To work around this limitation, we require users to write the comments themselves:

```
data Foo = Foo
deriveIso 'Foo
{-@ tof :: x:Foo -> {to (from x) == x} @-}
{-@ fot :: x:Rep Foo x -> {from (to x) == x} @-}
```

We intend to resolve this by extending Template Haskell to support comment generation.

### 4.5 A Note on Performance

One limitation to watch out for is the efficiency of the verified instances at runtime. A consequence of using GHC.Generics is that there are many intermediate data types used, and this can lead to runtime performance overheads if GHC does not optimize away the conversions to and from the intermediate types. It is sometimes possible to tune GHC’s optimization
flags to achieve performance that is comparable to direct, hand-written code [9], but as a general rule, code written with GHC.Generics tends to be slower overall.

We do not offer a solution to this problem in this paper, but it is worth noting that many of the classes that we discuss can be derived in GHC through other means. For instance, one can derive efficient implementations of the Eq, Ord, and Functor classes by writing

```haskell
data Pair a = MkPair a a
    deriving (Eq, Ord, Functor)
```

One thing we wish to explore in the future is verifying instances derived in this fashion. This will be non-trivial as the code that GHC derives often uses primitive operations that can be tricky to reason about. If this were implemented, we could quickly verify a set of commonly used type classes and have them be fast, too.

5 Aside: Logic

The idea of proof reuse is motivated from model theory in mathematical logic. First-order model theory studies properties of models of first-order theories using tools from universal algebra. In particular, preservation theorems study the closure properties of classes of models across algebraic operations. By interpreting Haskell type classes and verified type classes as algebraic objects, we can borrow these ideas to do generic proving and verified programming.

A Haskell type class can be interpreted as a signature in the sense of universal algebra, that is, a collection of function and relation symbols with fixed arities. Relations are identified with propositions, that is, functions whose codomain is Bool. For example, the type class Eq corresponds to the signature

\[ \sigma_{\text{Eq}} := (\equiv) \]

and the Ord class corresponds to the signature

\[ \sigma_{\text{Ord}} := (\leq, =) \]

"Type class laws", expressed as first-order axioms using refinement reflection are identified as a first-order theory, that is, a set of first-order statements (identified up to logical equivalence). For example, for VerifiedOrd, we have the theory of total orders given by

\[ T_{\text{Ord}} \]

with the axioms for reflexivity, antisymmetry, transitivity, and totality.

We can now interpret building an instance of a verified type class model-theoretically. A type is an instance of a verified type class, if it forms a structure in that signature, and is also a model of the first-order theory. For example, a type a is an instance of VerifiedOrd if there are operations \( =^a \) \( \leq^a \) so that \( A := (a, =^a, \leq^a) \) is a \( \sigma_{\text{Ord}} \) structure, and \( A \models T_{\text{Ord}} \), that is, A is a model of \( T_{\text{Ord}} \).

Given a first-order theory \( T \) and \( K \), the class of models of \( T \), one can ask if \( K \) is closed under algebraic operations like products (\( P(K) \)), coproducts (\( C(K) \)), substructures (\( S(K) \)), homomorphic images (\( H(K) \)), isomorphic images (\( I(K) \)). The answers to some of these are well known [6].

- \( I(K) = K \) for any \( T \).
- \( (\text{Łoś-Tarski}) \) \( S(K) = K \) if \( T \) is universal.
- \( SP(K) = K \) if \( T \) is a Horn-clause theory.

6 Related Work

Several languages with dependent types offer some degree of automation via datatype-generic programming. Dagand [5] develops a dependent type theory in Agda which, by encoding inductive data types in a universe of descriptions, allows deriving decidable (and boolean) equality in a straightforward manner. Al-Sibahi [1] presents a similar implementation of described types in Idris, based off of the dependent type theory by Chapman et al. [4], and demonstrates its utility in deriving instances of decidable equality, Functor, pretty-printing, and generic traversals. Altenkirch et al. also develop several universes of types in Epigram, which can be used to implement generic zipper options [2].

Liquid Haskell takes a somewhat different approach to equational reasoning than Agda and Idris. With refinement reflection, the programmer states the propositions as refinements, and Liquid Haskell is tasked with finding the proofs (with some gentle assistance by the programmer). The proof code simply acts as a guide to the SMT solver in determining satisfiability. In Agda and Idris, however, more responsibility is placed on the programmer to implement the details of proofs, as their typecheckers do not leverage a solver. In this way, refinement reflection inverts the relative importance of propositions and proofs, and by incorporating statements from propositions into the SMT solver, Liquid Haskell makes propositions "whole-program".

One thing to note is that while the datatype generic programming techniques in dependently typed languages like Agda, Idris, and Epigram are strictly more powerful, as they need to support a richer universe of datatypes than what Haskell offers, it comes with a burden of a higher learning curve. For instance, Al Sibahi notes that in the generic programming library he developed for Idris, "it requires considerable effort to understand the type signatures for even simple operations." [1] In contrast, the generic programming library we use here is designed to be relatively straightforward to implement, simple to explain, and give decently understandable type error messages.
The notion of reusing proofs over isomorphic types is also a familiar idea in the dependent types community. Barthe and Pons [3] formalize a theory of type isomorphisms in a modified version of the Calculus of Inductive Constructions. Type isomorphisms are extremely similar to the GenericIso class in Section 3.1. A type isomorphism between types $A$ and $B$ is essentially a pair of two well typed functions $f : A \rightarrow B$ and $g : B \rightarrow A$ that are mutual inverses (i.e., that $f(g(x)) = x$ and $g(f(x)) = x$ for all $x$) which allow one to take a proof of a property over $A$ and reuse it for $B$, and vice versa. Barthe and Pons use as motivation the ability to, for instance, reuse a proof of Peano (unary) natural numbers, which can be easier to reason about, for binary natural numbers, which can be used for more efficient algorithms. The technique could be adapted for inductive data types and their corresponding representations as well.

Isomorphisms (or equivalences) are also well studied in Homotopy Type Theory, and having a computational interpretation for univalence would mean that all type constructors act functorially on isomorphisms. This allows one to rewrite terms between isomorphic types, witnessed by a path, which facilitates type-generic programming. Some possible applications to generic programming are discussed by Licata and Harper in their work on 2-dimensional type theory [7].

7 Conclusion

We presented how law-abiding type class instances can be derived via generic programming. Class laws are encoded as refinement type specifications. The library author’s only responsibility is to provide proofs of the laws on generic representation types, and to implement a way to derive a verified instance for a type by reusing the proofs from its (provably isomorphic) representation type. Then, Haskell’s standard class resolution will derive provably law-abiding instances. We used this technique on the commonly used Haskell classes Eq, Ord, Semigroup, Monoid and Functor. Even though our technique currently suffers from various engineering limitations, it suggests a clean route towards semi-automated verification of class proofs by combining datatype-generic programming and type class resolution.

References


A Appendix

A.1 Full VerifiedOrd instance for (\(\ast\ast\))

\[
\text{instance} \ (\text{Ord} \ (f \ p), \ \text{Ord} \ (g \ p)) \Rightarrow \\
\text{Ord} \ ((f \ \ast\ast \ g) \ p) \ where \\
(x1 \ \ast\ast \ y1) \leq (x2 \ \ast\ast \ y2) = \\
\quad \text{if } x1 = x2 \ \text{then } y1 \leq y2 \ \text{else } x1 \leq x2
\]

\text{leqProdRefl} :: \ (\text{VerifiedOrd} \ (f \ p), \ \text{VerifiedOrd} \ (g \ p)) \\
\Rightarrow \ \text{Refl} \ ((f \ \ast\ast \ g) \ p)

\text{leqProdRefl} \ t@(x \ \ast\ast \ y) = \\
(t \leq t) \\
\quad =. \ (\text{if } x == x \ \text{then } y \leq y \ \text{else } x \leq x)

\text{leqProdAntisym} :: \ (\text{VerifiedOrd} \ (f \ p), \ \text{VerifiedOrd} \ (g \ p)) \\
\Rightarrow \ \text{Anti} \ ((f \ \ast\ast \ g) \ p)

\text{leqProdAntisym} \ p@(x1 \ \ast\ast \ y1) \ q@(x2 \ \ast\ast \ y2) = \\
\quad ((\text{if } x1 == x2 \ \text{then } y1 \leq y2 \ \text{else } x1 \leq x2) \ \&\& \\
\quad \ (\text{if } x2 == x1 \ \text{then } y2 \leq y1 \ \text{else } x2 \leq x1))

\text{leqProdTotal} :: \ (\text{VerifiedOrd} \ (f \ p), \ \text{VerifiedOrd} \ (g \ p)) \\
\Rightarrow \ \text{Total} \ ((f \ \ast\ast \ g) \ p)

\text{leqProdTotal} \ p@(x1 \ \ast\ast \ y1) \ q@(x2 \ \ast\ast \ y2) \ r@(x3 \ \ast\ast \ y3) = \\
\quad \text{case } x1 == x2 \ \text{of} \\
\quad \quad \text{True } \rightarrow \ \text{case } x2 == x3 \ \text{of} \\
\quad \quad \quad \text{True } \rightarrow \ (p \leq q \ \&\& q \leq r) \\
\quad \quad \quad \quad =. \ (y1 \leq y2 \ \&\& y2 \leq y3)

\text{leqProdTotal} :: \ (\text{VerifiedOrd} \ (f \ p), \ \text{VerifiedOrd} \ (g \ p)) \\
\Rightarrow \ \text{Total} \ ((f \ \ast\ast \ g) \ p)

\text{leqProdTotal} \ p@(x1 \ \ast\ast \ y1) \ q@(x2 \ \ast\ast \ y2) \ r@(x3 \ \ast\ast \ y3) = \\
\quad \text{case } x1 == x2 \ \text{of} \\
\quad \quad \text{True } \rightarrow \ \text{case } x2 == x3 \ \text{of} \\
\quad \quad \quad \text{True } \rightarrow \ (p \leq q \ \&\& q \leq r) \\
\quad \quad \quad \quad =. \ (y1 \leq y2 \ \&\& y2 \leq y3)

\text{leqProdTotal} :: \ (\text{VerifiedOrd} \ (f \ p), \ \text{VerifiedOrd} \ (g \ p)) \\
\Rightarrow \ \text{Total} \ ((f \ \ast\ast \ g) \ p)

\text{leqProdTotal} \ p@(x1 \ \ast\ast \ y1) \ q@(x2 \ \ast\ast \ y2) \ r@(x3 \ \ast\ast \ y3) = \\
\quad \text{case } x1 == x2 \ \text{of} \\
\quad \quad \text{True } \rightarrow \ \text{case } x2 == x3 \ \text{of} \\
\quad \quad \quad \text{True } \rightarrow \ (p \leq q \ \&\& q \leq r) \\
\quad \quad \quad \quad =. \ (y1 \leq y2 \ \&\& y2 \leq y3)
A.2 Full VerifiedOrd instance for (:+:)

instance (Ord (f p), Ord (g p)) ⇒ Ord ((f :+: g) p)
where
  refl  = leqProdRef1
  antisym = leqProdAntisym
  trans  = leqProdTrans
  total  = leqProdTotal

leqSumRef1
:: (VerifiedOrd (f p), VerifiedOrd (g p))
⇒ Ref1 ((f :+: g) p)
leqSumRef1 s@L1 x = (s ≤ s)
  =. x ≤ x
  =. True :: refl x
  ** QED

leqSumRef1 s@L1 y = (s ≤ s)
  =. y ≤ y
  =. True :: refl y
  ** QED

leqSumAntisym
:: (VerifiedOrd (f p), VerifiedOrd (g p))
⇒ Anti ((f :+: g) p)
leqSumAntisym p@L1 x q@L1 y = (p ≤ q ∧ q ≤ p)
  =. (x ≤ y ∧ y ≤ x)
  =. x == y :: antisym x y
  ** QED

leqSumAntisym p@L1 x q@L1 y = (p ≤ q ∧ q ≤ p)
  =. (True ∧ False)
  =. False
  =. p == q
  ** QED

leqSumAntisym p@R1 x q@L1 y = (p ≤ q ∧ q ≤ p)
  =. (False ∧ True)
  =. False
  =. p == q
  ** QED

leqSumAntisym p@R1 x q@R1 y = (p ≤ q ∧ q ≤ p)
  =. (False ∧ False)
  =. False
  =. p == q
  ** QED

leqSumTrans
:: (VerifiedOrd (f p), VerifiedOrd (g p))
⇒ Trans ((f :+: g) p)
leqSumTrans p@L1 x q@L1 y r@L1 z = (p ≤ q ∧ q ≤ r)
  =. (x ≤ y ∧ y ≤ z)
  =. x ≤ z :: trans x y z
  =. (p ≤ r)
  ** QED

leqSumTrans p@L1 x q@R1 y r@R1 z = (p ≤ q ∧ q ≤ r)
  =. (x ≤ y ∧ True)
  =. (p ≤ r)
  ** QED

leqSumTrans p@R1 x q@L1 y r@L1 z = (p ≤ q ∧ q ≤ r)
  =. (True ∧ False)
  =. (p ≤ r)
  ** QED

leqSumTotal
:: (VerifiedOrd (f p), VerifiedOrd (g p))
⇒ Total ((f :+: g) p)
leqSumTotal p@L1 x q@L1 y = (p ≤ q || q ≤ p)
  =. (x ≤ y || y ≤ x)
  =. True :: total x y
  ** QED

leqSumTotal p@L1 x q@R1 y =
Deriving Law-Abiding Instances

\[
(p \leq q || q \leq p) = (\text{True} || \text{False})
\]
** QED

\[
\text{leqSumTotal } p@(R1 x) \ q@(L1 y) = (p \leq q || q \leq p) = (\text{False} || \text{True})
\]
** QED

\[
\text{leqSumTotal } p@(R1 x) \ q@(R1 y) = (p \leq q || q \leq p) = (x \leq y || y \leq x)
\]

=. \text{True} :\!: \text{total } x \ y
** QED

\text{instance} \ (\text{VerifiedOrd } (f \ p), \text{VerifiedOrd } (g \ p)) \Rightarrow \text{VerifiedOrd } ((f :+; g) \ p) \text{ where}
\text{refl} = \text{leqSumRef}
\text{antisym} = \text{leqSumAntisym}
\text{trans} = \text{leqSumTrans}
\text{total} = \text{leqSumTotal}