Quotient Haskell: Lightweight Quotient Types for All

BRANDON HEWER, University of Nottingham, UK
GRAHAM HUTTON, University of Nottingham, UK

Subtypes and quotient types are dual type abstractions. However, while subtypes are widely used both explicitly and implicitly, quotient types have not seen much practical use outside of proof assistants. A key difficulty to wider adoption of quotient types lies in the significant burden of proof-obligations that arises from their use. In this article, we address this issue by introducing a class of quotient types for which the proof-obligations are decidable by an SMT solver. We demonstrate this idea in practice by presenting *Quotient Haskell*, an extension of Liquid Haskell with support for quotient types.

CCS Concepts: • **Theory of computation → Program reasoning**; • **Software and its engineering → Functional languages**.

Additional Key Words and Phrases: quotient types, refinement types, static verification

ACM Reference Format:
Brandon Hewer and Graham Hutton. 2024. Quotient Haskell: Lightweight Quotient Types for All. *Proc. ACM Program. Lang.* 8, POPL, Article 27 (January 2024), 31 pages. https://doi.org/10.1145/3632869

1 INTRODUCTION

Subtypes are ubiquitous, and are given by a type equipped with a subtyping predicate. Subtypes also have a well-known dual construction, *quotient types*, which are given by a type together with a collection of equations. While few languages currently support their use, there are many practical examples of quotient types, including algebraic structures such as monoids, and data structures such as bags. Intuitively, a subtype requires that we prove its predicate is respected on construction, while a quotient type requires that we prove that its equations are respected on elimination. In this manner, both subtypes and quotient types introduce proof-obligations, which in turn may require tedious manual proof construction in the absence of sufficient automated proof search.

Refinement types [Freeman and Pfenning 1991] are a class of subtypes for which the subtyping predicate, or *refinement*, is SMT-decidable. Restricting ourselves to this class of subtypes has an important practical benefit: a type-checker that utilises an SMT-solver can automate many of the proof obligations that arise from refinements. Indeed, this is a central feature of Liquid Haskell [Vazou 2016], an extension of Haskell with refinement types. A simple example of a refinement type is the even integers, expressed in Liquid Haskell by `{n:Int | n % 2 == 0}`. For any concrete integer, and for each of the common arithmetic operations, Liquid Haskell can check the evenness condition without requiring that we manually construct a proof.

In contrast to subtypes, automated proof for quotient types is much less explored. Indeed, the only languages with quotient types at present seem to be proof assistants. However, quotient types share an important practical utility with subtypes: they allow us to assert static properties that we hope can be validated by a type-checker. For example, the type of bags (multisets) can be expressed by

Authors' addresses: Brandon Hewer, University of Nottingham, , UK, Brandon.Hewer@nottingham.ac.uk; Graham Hutton, University of Nottingham, , UK, Graham.Hutton@nottingham.ac.uk.

ACM 2475-1421/2024/1-ART27
https://doi.org/10.1145/3632869

quotienting the type of lists with a quotient `swap :: x:a -> y:a -> xs:[a] -> x:y:xs == y:x:xs`. In particular, this quotient requires that for every function `f :: Bag a -> b` that takes a bag as an argument, we must have `f (x:y:xs) == f (y:x:xs)`, i.e. the behaviour of `f` must be invariant under permutation of the elements. In the absence of automated proof tools, this means supplying a manually constructed proof of this equality. In practice, this can quickly become burdensome and is a significant barrier to the use of quotient types in general programming.

In this article, we introduce *Quotient Haskell*, an extension of Liquid Haskell that extends the notion of refinement types to support a class of quotient types for which the elimination laws are SMT-decidable. In particular, the system supports quotient inductive types [Altenkirch and Kaposi 2016], a class of quotient types that simultaneously define the data of an inductive type together with equations on that data. Concretely, the paper makes the following contributions:

- We introduce the system using three examples: *mobiles* (commutative trees – Section 2), the *Boom hierarchy* (trees, lists, bags and sets – Section 3), and *rational numbers* (Section 4);
- We present a core language λ_Q for quotient types, by extending the core language λ_L for liquid types with typing (Section 5) and subtyping (Section 6) rules for quotients;
- We show how the notion of equality in the underlying liquid type system can be extended in λ_Q to make use of the equalities introduced by quotients (Section 7);
- We outline how Quotient Haskell is implemented, with a particular focus on how the new quotient typing features are realised (Section 8).

Related work is discussed in Section 9, and we reflect on the design, practical use and limitations of the system in Section 10. We assume a basic knowledge of Haskell, but to make the article more accessible we do not require expertise with type theory, quotient types or Liquid Haskell. While Haskell serves as the implementation vehicle in this article, the ideas introduced by the core language are applicable to any programming language with a liquid type system. The Quotient Haskell system itself is freely available online as supplementary material [Hewer 2023].

2 MOBILES

To describe the class of quotient types that we introduce in this article and are implemented by Quotient Haskell, we present three increasingly involved examples. The provided examples outline the necessary concepts required to use Quotient Haskell. In this section, we explore the first of these examples, *mobiles*, which are commutative trees in which subtrees with the same parent can freely be swapped. For the purposes of this example, we will consider binary trees, defined in Haskell as follows, but the same idea also applies to rose trees and multiway trees:

```
data Tree a = Leaf | Bin a (Tree a) (Tree a)
```

In usual parlance, `Leaf` and `Bin` are the *data constructors* of the `Tree` type. A quotient type extends the typical notion of an algebraic datatype with a new kind of constructor, that we shall refer to as an *equality constructor*. In the type-theory literature, equality constructors can also be referred to as *path constructors*. However, while a data constructor introduces new terms of a data type, an equality constructor introduces new *equalities* between terms of a type. For example, in order to define mobiles we will require an equality constructor that quotients `Tree` to assert the commutativity condition. That is, we require an equality constructor of the following form:

```
swap :: x:a -> l:Tree a -> r:Tree a -> Bin x l r == Bin x r l
```

In the above definition of swap we have made use of the dependent syntax of Liquid Haskell. In particular, a type of the form `x:a -> P` whereby `P` is a proposition can be understood as universal quantification over the elements of a and can be read as 'for all x of type a, P holds'. While the syntax used for equality in the above definition is shared by Haskell's `Eq` typeclass, it is instead a type-level proposition and this syntax is inherited from Liquid Haskell.

Note that the target of the swap constructor introduces an equality between trees. In particular, swap asserts that two trees with swapped children must be treated identically. Concretely, we can define the datatype of binary mobiles in Quotient Haskell as follows:

```
data Mobile a
  =  Tree a
  |/ swap :: x:a -> l:Mobile a -> r:Mobile a -> Bin x l r == Bin x r l
```

As with any refinement in Liquid Haskell, this definition must be given within a *{−@ @−}* block. However, for brevity we omit these additional annotations throughout this article. As shown in the above example, a quotient type in Quotient Haskell is defined with a similar syntax to that of ADTs, with some notable differences. Firstly, an underlying type must directly follow after the equality symbol. The underlying type can be a Liquid Haskell refinement type or even another quotiented type. After providing the underlying type, the equality constructors must follow, each preceded by the `/` character. Any equality constructor defined in a Liquid Haskell block, such as swap, is introduced as a term with the same name within the namespace in which it is defined. Consequently, equality constructors can be used in Liquid Haskell proofs. For example, swap can be understood as a Liquid Haskell proof function of type `x:a -> l:Mobile a -> r:Mobile a -> { Bin x l r == Bin x r l }`.

Importantly, the left-hand term of the equality target of an equality constructor must be in *canonical form*, i.e. the left-hand term must be a valid match term. This is important in circumstances of induction-recursion, whereby a function on an inductive type appears in any of its equality constructors. A formal account of this rule and an explanation for its existence is given in Section 8. The right-hand term of the equality may vary freely, as long as it has the correct type.

Note that quotient types in Quotient Haskell introduce a new type constructor into the Liquid Haskell namespace. That is, a quotient type definition can be read as defining a new type rather than simply giving a refinement for an existing type. This is in contrast to the usual approach of datatype refinement in Liquid Haskell, whereby type constructors refer directly to their underlying GHC counterparts. We adopt this alternative approach for the practical purpose of avoiding having to create a newtype for every quotient type defined on an underlying type. To explain the issue, let us consider how the usual Liquid Haskell approach to data type refinement could be adopted for our `Mobile` example. In particular, this approach would involve redefining `Tree` in a Liquid Haskell block, and then appending the swap constructor. In keeping with the terminology used in Liquid Haskell, we say that the type `Tree` has been *refined* to the type of mobiles. Consequently, any function defined on the type `Tree` must be a function on mobiles, i.e. must respect the swap constructor. As such, to define multiple quotient types on the same underlying `Tree` type, such as sets or bags, we would need require a distinct type definition for each.

Crucially, every binary tree is a binary mobile. As described in more detail in Section 5, this imposes an ordering relation on types subject to quotienting. A key practical concept of quotient types is that, unlike subtypes, there are no proof obligations imposed by quotienting on term construction. Rather, an equality constructor introduces proof obligations on term elimination. Intuitively, subtyping imposes conditions when *building* a term and quotienting imposes conditions when *using* a term. For example, for the swap equality constructor this means that for any function of the form `f :: Mobile a -> b`, we require that `f (Bin x l r) = f (Bin x r l)`. Functions that do not respect this law are not valid functions on mobiles, and we should expect this to be checked by our type checker. Indeed, this is precisely what Quotient Haskell can be used for. For example, the following function on trees cannot be refined to a function on mobiles:

```
isLeft :: Tree Bool -> Bool
isLeft Leaf = False
isLeft (Bin p Leaf z) = p
isLeft (Bin p (Bin q x y) z) = q
```

Evidently, the function `isLeft` distinguishes between the left and right subtrees when they do not contain the same logical value. In contrast, an example of a function on trees that can be refined to a function on mobiles is the following summation function:

```
sum :: Tree Int -> Int
sum Leaf = 0
sum (Bin n l r) = n + sum l + sum r
```

Because addition is commutative, we should expect that the `sum` function can be refined over binary mobiles. That is, we should expect `sum :: Mobile Int -> Int` to be a valid typing judgement. The necessary proof obligation imposed by the `swap` equality constructor is

```
n:Int -> l:Mobile Int -> r:Mobile Int -> sum (Bin n l r) == sum (Bin n r l)
```

After unfolding the definition of `sum` on both sides of the equality, we can observe that the necessary proof follows from commutativity of addition. Indeed, a constraint of this form can be solved by Liquid Haskell with no further intervention.

A second example of a function on mobiles trees is that of the `map` function. We recall that the typical definition for the `map` function on binary trees is given as follows:

```
map :: (a -> b) -> Tree a -> Tree b
map f Leaf = Leaf
map f (Bin x l r) = Bin (f x) (map f l) (map f r)
```

We expect the type of `map` to refine to `(a -> b) -> Mobile a -> Mobile b`. In this example, after unfolding definitions and eliding quantification, the necessary condition that must hold is:

```
Bin (f x) (map f l) (map f r) == Bin (f x) (map f r) (map f l)
```

This equality is witnessed by the term `swap (f x) (map f l) (map f r)`, and our implementation of Quotient Haskell is capable of automatically applying this single proof step.

By making use of Liquid Haskell's refinement types, we can define the general `fold` on mobiles. Recall that the general `fold` on binary trees is given as follows:

```
fold :: (a -> b -> b -> b) -> b -> Tree a -> b
fold f z Leaf = z
fold f z (Bin x l r) = f x (fold f z l) (fold f z r)
```

We cannot naively refine `fold` by replacing `Tree` with `Mobile`, as in general we do not have:

```
fold f z (Bin x l r) == fold f z (Bin x r l)
```

After unfolding definitions on both sides of the equality we can observe that this condition only holds when the function `f` is symmetric in its second and third arguments. More specifically, this means that we have a family of equalities of the following form:

```
x:a -> l:b -> r:b -> f x l r = f x r l
```

In Liquid Haskell, we can define the type of such functions as follows:

```
type Fun a b = ( f : a -> b -> b -> b ,
                 x:a -> l:b -> r:b -> { f x l r == f x r l } )
```

Unfortunately, this approach to defining a subtype of the function space is not particularly ergonomic, and requires explicitly carrying proof witnesses. However, Liquid Haskell necessarily does not permit higher-order propositions to inhabit the type of propositions `Prop` and as such, this is a work-around. With this definition of `Fun`, we can define a general `fold` on mobiles by

```
fold :: Fun a b -> b -> Mobile a -> b
fold (f, p) z Leaf = Leaf
fold (f, p) z (Bin x l r) = f x (fold (f, p) z l) (fold (f, p) z r)
```

The witness that the `swap` constructor is respected by our new `fold` function follows from the second component of the pair. However, the version of Quotient Haskell introduced by this article cannot implicitly make use of the explicit proof term. As such, this definition will not type-check without additional intervention. In particular, we must explicitly construct the necessary witness that the above `fold` function respects the `swap` equality constructor. To achieve this in Quotient Haskell, we first define the following unrefined proof:

```
foldSwap :: Fun a b -> b -> a -> Tree a -> Tree a -> Proof
foldSwap (f, p) z x l r = p x (fold (f, p) z l) (fold (f, p) z r)
```

This is precisely the proof that `fold` respects the `swap` equality constructor. Notably, the arguments of `swap` are expanded as arguments to `foldSwap`. Any explicit proof that an equality constructor is respected must follow this form in Quotient Haskell. The next step then is to introduce the following refinement inside of a Liquid Haskell block:

```
respects<fold, swap> foldSwap
  :: f:Fun a b -> z:b -> x:a -> l:Mobile a -> r:Mobile a
  -> { fold f z (Bin x l r) == fold f z (Bin x r l) }
```

The function `foldSwap` will be checked in the same manner as any other Liquid Haskell proof. In addition, by adding the prefix `respects<fold, swap>`, Quotient Haskell will check whether `foldSwap` is a valid witness that `fold` respects the `swap` constructor. A valid witness of a quotient being respected is simply a function whose type precisely matches the relevant respectability theorem. As our definition of `foldSwap` is both of the correct type and a valid proof, it can be used as an explicit witness that `fold` respects the `swap` equality constructor. Notably, there are many circumstances that may arise where the SMT-solver of Liquid Haskell is unable to automatically generate the necessary proof that an equality constructor is respected. In such circumstances, the outlined explicit approach to providing the witness can be used.

As an example of using our general `fold` function for mobiles, we can consider using it to redefine our previous `sum` function. To do this, we first define a function `add3 :: Int -> Int -> Int -> Int` that adds three integers, which can then be reflected to define the following proof witness:

```
add3Comm :: x:Int -> y:Int -> z:Int -> { add3 x y z == add3 x z y }
add3Comm x y z = trivial *** QED
```

In Liquid Haskell, the triviality of the above proof follows from the triviality of the commutativity of addition. Finally, we can simply define `sum = fold (add3, add3Comm) 0`.

3 BOOM HIERARCHY

In this section, we explore the family of datatypes comprising trees, lists, bags and sets, which are collectively known as the *Boom hierarchy* [Meertens 1983]. These examples will demonstrate how the ordering relation on quotient types can be used to practical effect.

We begin this section with a notion of a tree data structure that varies from that used in our mobiles example, in which data is only found in the leaves:

```
data Tree a = Empty | Leaf a | Join (Tree a) (Tree a)
```

This notion of tree forms the basis for defining all the other types in the Boom hierarchy. We begin this exploration with a less familiar definition of the list datatype by means of quotienting:

```
data List a
  =  Tree a
  |/ idl :: x:List a -> Join Empty x == x
  |/ idr :: x:List a -> Join x Empty == x
  |/ assoc :: x:List a -> y:List a -> z:List a
           -> Join (Join x y) z == Join x (Join y z)
```

This definition captures the idea that lists can be obtained from trees by requiring that the `Join` constructor is associative, and has `Empty` as the identity element. Intuitively, this definition of a list can be seen as a direct translation of the algebraic definition of a monoid structure on a given type. Indeed, this is precisely why lists are the *free* monoid on the parameter type.

This above formulation of lists might seem rather complex when compared to the standard version: `data List a = Nil | Cons a (List a)`. The benefit of the above quotiented formulation is that concatenation is given simply by the constructor `Join`, and thus has asymptotic runtime complexity of $O(1)$. For example, when we are primarily building lists by their monoid interface, such as when using the `Writer` monad, this can be a more performant representation. Furthermore, using the monoid laws on `[a]` we can give a well-typed unfolding `toList :: List a -> [a]` of tree-based lists into standard lists, and this has runtime complexity of $O(n)$. As such, it can sometimes be more performant to use this tree representation of lists when constructing them through repeated concatenation, and subsequently apply `toList` when required.

Examples of functions on trees that can be refined to the quotiented type `List` include:

```
sum :: Tree Int -> Int
map :: (a -> b) -> Tree a -> Tree b
filter :: (a -> Bool) -> Tree a -> Tree b
```

In contrast, an example of a function on the `Tree` datatype that cannot be refined to a function on lists is the following inductive subtraction function:

```
subtr :: Tree Int -> Int
subtr Empty = 0
subtr (Leaf n) = n
subtr (Join x y) = subtr x - subtr y
```

In particular, `subtr` does not respect the associativity condition introduced by the `assoc` equality constructor, because integer subtraction is not associative.

The next datatype in the Boom hierarchy is multisets, also known as *bags*, which can be used to count the number of occurrences of elements from a collection. Bags can intuitively be thought of as lists for which the order of the elements cannot be used, which means that the `Join` constructor must be commutative. Consequently, while `List` characterises the free monoid construction on the parameter type, our definition for `Bag` will characterise the free commutative monoid construction. In Quotient Haskell, we can define the `Bag` datatype as a quotient of `List` as follows:

```
data Bag a
  =  List a
  |/ comm :: xs:Bag a -> ys:Bag a -> Join xs ys == Join ys xs
```

The definition of Bag is an example of further quotienting an already defined quotient type. Indeed, this is precisely how the datatypes of the Boom hierarchy can be understood to form a hierarchy. In Quotient Haskell, this hierarchy is made explicit in a typing relation derived from the evident ordering of quotient types, which so far comprises the ordering `Tree a <: List a <: Bag a`. Intuitively, `a <: b` can be read as 'every element of `a` is an element of `b`'. Furthermore, by contravariance we also have `(Bag a -> b) <: (List a -> b) <: (Tree a -> b)`. Importantly, this means that in Quotient Haskell it is only necessary to refine a function to the greatest quotient type possible in a given hierarchy. For example, if we were to refine the `sum` function on `Tree` to a function on `Bag a`, then it would also be possible to apply `sum` to an element of type `List a`.

Alongside `sum`, `map` and `filter`, another useful function on trees that can be refined to a function on bags counts the number of elements that satisfy a given property:

```
countIf :: (a -> Bool) -> Tree a -> Int
countIf p Empty = 0
countIf p (Leaf a) = if p a then 1 else 0
countIf p (Join x y) = countIf p x + countIf p y
```

In order to verify that `countIf` can be refined to a function on bags, it is necessary to check that it respects both the `comm` equality constructor and all of the equality constructors given in the definition of `List`. In particular, these laws follow from the fact that integers form a commutative monoid under addition, and can be automatically verified by Quotient Haskell.

An example of a function on trees that can be refined to a function on lists but not on bags, is the following simple function that converts a tree into a list:

```
toList :: Tree a -> [a]
toList Empty = []
toList (Leaf a) = [a]
toList (Join x y) = toList x ++ toList y
```

In this example, we eliminate trees into the non-commutative monoid of the Haskell list type equipped with concatenation. Indeed, because the `toList` function constructs lists using their monoidal interface, the laws introduced by the equality constructors of `List` are evidently satisfied. However, concatenation is not commutative and hence `toList` cannot be refined to a function on bags. Of course, we should not expect to be able to convert an arbitrary bag into a list, as this would require a unique identification of an ordering on its elements.

The final type in the Boom hierarchy is *sets*, unordered collections that can only contain each element once. Sets can also be understood as bags for which any repeated occurrences of elements are forgotten. In Quotient Haskell, we can define the type of sets as follows:

```
data Set a
  =  Bag a
  |/ idem :: xs:Set a -> Join xs xs == xs
```

In this definition, the `idem` equality constructor asserts that we cannot distinguish between a set and that same set unioned with itself. Alongside the `comm` equality constructor introduced in the definition of bags, this guarantees that repeated occurrences of an element cannot be used to alter the behaviour of a function on sets. Note that the term `Join xs xs` is not a valid match term in Haskell because of the duplicated variable `xs`, and therefore does not satisfy the necessary requirement for appearing on the left-hand side of an equality constructor. However, this is indeed valid syntax in Quotient Haskell, and is equivalent to the following formulation:

```
idem :: xs:Set a -> ys:Set a -> {xs == ys} -> Join xs ys == xs
```

However, the original and more concise formulation of idem is generally preferable in practice, as it does not introduce the extra precondition {xs == ys} to the type-checker.

A simple example of a function on trees that can be refined to a function on sets determines if a given value is contained in a tree, and can be defined as follows:

```
contains :: Eq a => a -> Tree a -> Bool
contains x Empty = False
contains x (Leaf y) = x == y
contains x (Join t u) = contains x t || contains x u
```

All of the necessary properties that are required to verify that contains refines to a function on sets follows from the fact that Bool forms an idempotent commutative monoid under logical disjunction. For example, in order to show that the comm equality constructor is satisfied we would require that contains x (Join xs xs) == contains x xs, which unfolds to:

```
contains x xs || contains x xs == contains x xs
```

This property is then trivially true because disjunction is idempotent. We have already seen an example of a function on trees that can be refined to a function on bags but not on sets, namely countIf, because the number of occurrences of an element matters for this function.

4 RATIONAL NUMBERS

In this section we show how the *rational numbers*, a classic example of a quotient type, can be captured in Quotient Haskell. The approach is quite different from their typical representation in a language such as Haskell, where a well-behaved interface is presented on an abstract type with a hidden data representation. In contrast, by defining the rationals as a quotient type, their implementation details can be exposed without comprising correctness. As discussed in this article, this increased flexibility comes at the cost of extra proof obligations arising from quotient respectfulness theorems. However, in Quotient Haskell such proof obligations are automatically resolved by the type checker when operating within the SMT-decidable logic of Liquid Haskell.

A well-known constructive definition of the rational numbers involves quotienting pairs of integers representing the numerator and denominator, with the proviso that the second element is non-zero to preclude division by zero. We can define the type of non-zero integers in Liquid Haskell by type NonZero = { n : Int | n /= 0 }. Using this type definition, we can then proceed to define the rational numbers with the following quotient type:

```
data Rational
  = (Int, NonZero)
  |/ eqR :: w:Int -> x:Int -> y:NonZero -> z:NonZero
        -> {w * z == x * y} -> (w , y) == (x , z)
```

Note that the underlying type of Rational uses a refinement predicate, n /= 0, to ensure that the second element of a pair of integers is non-zero. More generally, Quotient Haskell allows any rank 1 liquid type to be used as the underlying type in the definition of a quotient type. The above definition uses the 'cross multiplication' approach to decide if two rationals are equal, but this is not the only possible approach. For example, we could use a greatest common divisor function gcd :: Int -> Int -> Int to define the following equality constructor:

```
eqGCD :: x:Int -> y:NonZero -> (x, y) == (x `div` gcd x y, y `div` gcd x y)
```

While the equality constructor eqR quantifies over four variables, eqGCD only requires quantification over two variables. However, this alternative presentation of rational numbers requires that the definition of gcd be unfolded when the type checker validates many proof obligations that arise for common applications of rational numbers. Moreover, `eqGCD` requires additional work to validate its well-formedness. In particular, it must provably be the case that if `y /= 0` then ``y `div` gcd x y /= 0``. In practice, this approach is often less performant than simply quotienting by `eqR`, and can often result in constraints being generated that require an unreasonable amount of time to be validated by an SMT solver. Indeed, as highlighted by the example of rational numbers, performance can be an important consideration in choosing a particular quotient representation. As such, we proceed with our original presentation of the rationals.

As a first example of defining functions on rational numbers, we consider a simple function that decides if a pair of integers represents a negative rational:

```
isNegative :: (Int, Int) -> Bool
isNegative (m, n) = (m < 0 && n >= 1) || (m > 0 && n <= -1)
```

We can observe that `isNegative` is a valid function on rational numbers, and should therefore expect that we can refine its type to `Rational -> Bool`. A simple but important observation in Liquid Haskell is that subtyping constraints can always be added to function inputs. The dual observation in Quotient Haskell is that outputs of functions can always be quotiented. Both observations follow from the subtyping rules of the type system. In this particular case, `isNegative` can trivially be refined to a function of type `(Int, NonZero) -> Bool`, and the type checker of Quotient Haskell need only consider whether it respects the `eqR` equality constructor. Concretely, the type checker will automatically verify that given variables `w:Int`, `x:Int`, `y:NonZero` and `z:NonZero` along with a precondition `w * z == x * y` the following unfolded condition holds:

```
((w < 0 && y >= 1) || (w > 0 && y <= -1)) == ((x < 0 && z >= 1) || (x > 0 && z <= -1))
```

Included in the large collection of functions that can be defined on the rational numbers are the standard arithmetic operations. Notably, for operations that rationals are closed under, such as addition and multiplication, it is not necessary to simplify the pair of integers to satisfy the `eqR` equality constructor. As demonstrated in previous examples, Quotient Haskell can make use of equality constructors such as `eqR` when eliminating into a quotient type. However, this is not the case if we were to instead attempt to extract the numerator and denominator from a rational number. For example, the following projection cannot be refined to a function on rationals:

```
numerator :: (Int, Int) -> Int
numerator (m, _) = m
```

In particular, it is not the case that if `w * z == x * y` then `w == x`, and attempting to refine `numerator` to the type `Rational -> Int` will yield a type error in Quotient Haskell. Instead, we consider refining the type of the following function, which reduces a rational number to its simplest form:

```
reduce :: (Int, Int) -> (Int, Int)
reduce (m, n)
  | m == 0    = (0, 1)
  | m < 0     = (-m `div` d, -n `div` d)
  | otherwise = (m `div` d, n `div` d)
  where d = gcd m n
```

At first glance, it is natural to ask why the `reduce` function negates both elements of the pair when the first element is negative. In particular, it is not immediately evident why `-1 / 2` should be considered anymore reduced than `1 / -2`. Indeed, if we were to change the condition `m < 0` to `n < 0` the definition of `reduce` would remain equally valid. However, we cannot omit this line altogether, and it is required in order to refine the type of `reduce` to `Rational -> (Int, Int)`.

The above behaviour is not a bug in Quotient Haskell's type checker, but is essential to the correctness of the `reduce` function. In particular, the respectfulness theorem resulting from the `eqR` equality constructor ensures that we cannot distinguish between rational numbers such as `-1 / 2` and `1 / -2`, and hence we must choose a uniform way to reduce such terms. That is, we must choose a uniform way to handle the sign of the rational number. The approach used in the above definition of `reduce` is to always move the sign to the denominator, however it is equally valid to instead always move the sign to the numerator. There are other possible ways to define a `reduce` function that uniformly handles the sign of a rational number, such as by constructing a triple `(Bool, Nat, Nat)` in which a Boolean is used to represent the sign.

In addition to the uniform handling of sign, it is crucial that the `gcd` function used in the definition of `reduce` has the property that if `w * z == x * y` then the following two propositions hold:

```
abs w `div` gcd w y == abs x `div` gcd x z
abs y `div` gcd w y == abs z `div` gcd x z
```

This is evidently the case for any correct implementation of greatest common divisor. However, Liquid Haskell does not currently reflect the standard library function `gcd` in its type system. In practice, this means that `gcd` needs to be redefined. Many of the standard approaches to such a definition, such as Euclid's algorithm or recursively applying the modulus function, verifiably exhibit the necessary property when proof by logical evaluation (PLE) is enabled in Liquid Haskell. In Quotient Haskell, PLE is enabled by default. Importantly, the described property of the `gcd` function, along with the η-rule for the product type, are precisely the equations used by Quotient Haskell to verify that `reduce` can indeed be refined to `reduce :: Rational -> (Int, Int)`. Moreover, we can choose to further refine the type to `reduce :: Rational -> (Int, NonZero)`, whereby the second component result can verifiably be shown to always be non-zero. Using this definition of `reduce`, we can compose with the projections `fst` and `snd` and to obtain the functions `numerator :: Rational -> Int` and `denominator :: Rational -> NonZero`, respectively.

5 CORE LANGUAGE

In this section we present a core language λ_Q, which is a variant of the lambda calculus with patterns [Klop et al. 2008], and a Hindley-Milner type system extended with liquid and quotient types. We introduce λ_Q to give a precise account of how Quotient Haskell extends the type system of Liquid Haskell. In particular, λ_Q is formulated as a conservative extension to a generic underlying liquid type system. In this way, Quotient Haskell can be understood as an implementation of λ_Q for which the underlying refinement type system is Liquid Haskell, while λ_Q can be understood as an extension of λ_L, which was introduced to capture liquid types [Rondon et al. 2008].

The crucial features that λ_Q introduces to an existing liquid type system are quotients and their typing rules. Moreover, constructors and pattern matching by means of λ-case functions are included as the mechanism by which quotients can be used. Extending a Hindley-Milner type system with constructors and pattern matching is straightforward, so we do not discuss the details here. Furthermore, we only introduce the syntax and typing rules of λ_Q that are either novel, or important for understanding key ideas.

A type environment Γ for λ_Q is a sequence of type bindings $x : \tau$, guard predicates ϕ, and typed quotients $Q :: \tau$. The notation $Q :: \tau$ should be read as 'Q is a well-formed quotient on the underlying type τ'. Quotients that appear in a type environment alter the notion of equality between terms of the underlying type, and this context-sensitive notion of equality is described in Section 7. Notably, as with the core language λ_L, a complete account of the typing rules for λ_Q requires that we also allow *guard predicates* to inhabit an environment. Intuitively, guard predicates are propositions in the refinement logic that correspond to assumptions that hold true within the branches of conditionals. That is, guard predicates appear in a type environment to represent the known truth values of conditionals in the branches of an if-expression.

In addition to guard predicates, λ_Q inherits the notion of a qualifier set from λ_L. A qualifier set consists of all well-formed Boolean expressions that can be constructed from the environment variables. In an environment with a qualifier set $\mathbb{Q}$, a well-formed refinement predicate is a conjunction over any subset of $\mathbb{Q}$. We write $\Gamma \vdash_{\mathbb{Q}} x : \tau$ to mean that x has a liquid type τ in the environment Γ with qualifier set $\mathbb{Q}$, and $\Gamma \models \tau$ to mean that the type τ is well-formed in Γ. Moreover, for refinement predicates we write $\Gamma \vdash_{\mathbb{Q}} \phi$ to mean that ϕ is well-formed in Γ with qualifier set $\mathbb{Q}$ and $\Gamma \models \phi$ to mean that the refinement predicate ϕ is well-formed and provable in Γ. Finally, we write $\Gamma \vdash_{\mathbb{Q}} Q :: \sigma$ to mean that the quotient Q is defined on an underlying type σ in context Γ with qualifier set $\mathbb{Q}$. The details of liquid typing can be found in the original work on λ_L [Rondon et al. 2008].

A quotient in λ_Q is represented by a sequence of quantified variables followed by a triple of a refinement predicate ϕ, a pattern p, and a term e. This quantified triple is written as **forall** $v_1 :: \tau_1$ **in** $\cdots$ **forall** $v_k :: \tau_k$ **in** $\phi \Rightarrow p == e$, and we can think of this as an equality constructor in which v_i are variables, ϕ is the quotient precondition, p is the left-hand side of the target equality, and q is the right-hand side. This is characterised by the two following introduction rules:

$$\frac{\Gamma \models \sigma \qquad \Gamma, v : \tau \vdash_{\mathbb{Q}} Q :: \sigma}{\Gamma \vdash_{\mathbb{Q}} \textbf{forall}\ v : \tau\ \textbf{in}\ Q :: \sigma} \qquad\qquad \frac{\Gamma \vdash_{\mathbb{Q}} \phi \qquad \Gamma \vdash_{\mathbb{Q}} p : \sigma \qquad \Gamma \vdash_{\mathbb{Q}} e : \sigma}{\Gamma \vdash_{\mathbb{Q}} \phi \Rightarrow p == e :: \sigma}$$

Note that the first rule implicitly makes use of the weakening rule in the type system of λ_Q. In particular, the judgement $\Gamma, v : \tau \vdash_{\mathbb{Q}} Q :: \sigma$ assumes that $\Gamma, v : \tau \models \sigma$, which requires weakening of the premise $\Gamma \models \sigma$. The necessity of the weakening rule is a consequence of the underlying type not being allowed to depend on the terms over which the quotient varies. That is, λ_Q does not permit quotient inductive families. Furthermore, an additional weakening rule is given for quotients that asserts that every well-typed quotient in a context Γ remains well-typed in any extension of Γ.

Refinements in a refinement type system may depend on terms and hence come equipped with a notion of substitution in types. In λ_Q, the substitution operation is extended over quotient types by first defining substitution in quotients. In particular, for every substitution σ, i.e. a finite map from variables to terms $v_1 \mapsto e_1; \cdots ; v_k \mapsto e_k$, we define substitution for quotients as follows:

$$(\textbf{forall}\ v : \tau\ \textbf{in}\ Q)[\sigma] \ := \ (\textbf{forall}\ w : \tau\ \textbf{in}\ Q[v \mapsto w][\sigma]) \quad w \text{ not free in } \sigma$$

$$(\phi \Rightarrow p == e)[\sigma] \ := \ \phi[\sigma] \Rightarrow p[\sigma] == e[\sigma]$$

That is, we construct $Q[\sigma]$ in a standard way by avoiding the capture of bound variables and by applying σ to both the precondition and equality terms that constitute Q. This definition of substitution for quotients can in turn be used to define substitution in quotient types by simply applying a given substitution to both the underlying type and the quotient.

For every type τ and every quotient $Q :: \tau$, we can form a new type that represents the quotient of τ by Q and is denoted $\tau \ / \ Q$. This type formation or well-formedness rule is given as follows:

$$\frac{\Gamma \models \tau \qquad \Gamma \vdash_{\mathbb{Q}} Q :: \tau}{\Gamma \models \tau \ / \ Q}$$

An important property of quotient types is that every term that inhabits an underlying type τ must also inhabit $\tau \ / \ Q$. This simple property is captured by the following introduction rule:

$$\frac{\Gamma \vdash_{\mathbb{Q}} x : \tau \quad \Gamma \vdash_{\mathbb{Q}} Q :: \tau}{\Gamma \vdash_{\mathbb{Q}} x : \tau \ / \ Q}$$

Note that the above rule introduces an implicit type conversion from τ to $\tau \ / \ Q$, and hence this can lead to ambiguity of expressions in the refinement logic. For example, the proposition that two terms $x, y : \tau$ are equal can be expressed as $x == y$, however this may not be logically equivalent to the same expression where x and y are instead considered to be elements of $\tau \ / \ Q$. In particular, it is possible that the quotient Q may equate previously distinct elements x and y. As such, we will write $x \equiv_\sigma y$ for a given type σ to denote equality in the refinement logic between terms two x, y when considered as elements of σ, or simply $x \equiv y$ if the quotient is clear from the context.

We note that together with the introduction rules for quotients, the introduction rule for quotient types allows us to apply a *sequence* of quotients. That is, any valid quotient Q on an underlying type τ is also a valid quotient on the type $\tau \ / \ P$ for any quotient P. It is essential that any sequence of quotients of a type must be both idempotent and invariant under reordering, i.e. multiple quotients of an underlying type must be considered together as a *set*. We can give a formal characterisation of these two rules, which we term the 'idempotent' and 'permutation' rule, as follows:

$$\frac{\Gamma \vdash_{\mathbb{Q}} x : \tau \ / \ P \ / \ P}{\Gamma \vdash_{\mathbb{Q}} x : \tau \ / \ P} \qquad \frac{\Gamma \vdash_{\mathbb{Q}} x : \tau \ / \ P_{\sigma(1)} \ / \cdots / \ P_{\sigma(n)} \qquad \sigma \text{ is a permutation } n \simeq n}{\Gamma \vdash_{\mathbb{Q}} x : \tau \ / \ P_1 \ / \cdots / \ P_n}$$

Alternatively, we could have used sets directly in the syntactic construction rule for quotient types in λ_Q. This alternative approach would allow us to replace the two rules above with a single typing rule corresponding to the union of sets. However, this would in turn complicate the λ-case formation rule for quotient types that we introduce later in this section. Consequently, we continue with the above approach in our presentation of the typing rules of λ_Q. We note that while a naive implementation approach of this rule such as exhaustively checking every permutation of n quotients is $n!$, in practice the number of quotients used is typically very small.

Another essential property of quotient types is a generalisation of the λ-formation rule. In particular, we expect that any function defined on a quotient type that does not match on its input is always well-typed. Concretely, this generalised λ-formation rule can be expressed as follows:

$$\frac{\Gamma, x : \tau \ / \ Q \vdash_{\mathbb{Q}} e : \sigma \qquad \Gamma \vdash_{\mathbb{Q}} Q :: \tau}{\Gamma \vdash_{\mathbb{Q}} \lambda x.e : (v : \tau \ / \ Q) \to \sigma}$$

In order to give the typing rule for functions that match on their inputs, we will first require a formal notion as to what it means for a particular 'case' of a matching function to respect a quotient. To do this, we will define a context-sensitive binary relation $\Gamma \models \bullet \rightsquigarrow \bullet$ whose first element is a finite sequence of pairs of patterns and terms and whose second element is a quotient. We write elements of this relation in the form $\lambda \ \{p_1 \to e_1; \ \cdots \ ; \ p_k \to e_k\} \rightsquigarrow Q$, or simply $\lambda \ \{p \to e\} \rightsquigarrow Q$. This relation will characterise precisely when a particular λ-case term respects a given quotient. In order to define this relation, we first introduce a number of auxiliary definitions.

First of all, we make use of unification or *matching* of patterns. This is a well-known concept for lambda calculus with patterns, and we do not reintroduce this idea here. We write $x \sim_\sigma y$ to denote that the terms x and y can be unified, with the *most general unifier* given by the substitution σ. The property that σ is the most general unifier is crucial to the correctness of the core language and is necessary for the proof of Proposition 3, which requires uniqueness of the most general unifier. With this in mind, we continue with an inductive definition of the context-sensitive relation $\Gamma \models \lambda \ \{ \ p \to e \ \} \rightsquigarrow Q$. We begin with the more involved base case, given by the following rule:

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathbb{Q}} \rho : \tau \qquad \Gamma \vdash_{\mathbb{Q}} p_1, \ \ldots, \ p_k : \tau \qquad \Gamma \vdash_{\mathbb{Q}} e_1, \ \ldots, \ e_k : v \qquad \Gamma \vdash_{\mathbb{Q}} \rho \sim_\sigma p_k \\ \Gamma, \ \phi[\sigma] \models e_k[\sigma] \equiv_v \lambda \ \{p_1 \to e_1; \ \ldots; \ p_k \to e_k\} \ t[\sigma] \qquad \forall \ i \ j \ \sigma'. \ p_i \sim_{\sigma'} \ p_j \Rightarrow i = j\end{array}}{\Gamma \models \lambda \ \{ \ p \to e \ \} \rightsquigarrow (\phi \Rightarrow \rho == t)}$$

Importantly, this rule makes use of the extended notion of equality in λ_Q described earlier in this section. The condition $\forall i\ j\ \sigma'.\ p_i \sim_{\sigma'} p_j \Rightarrow i = j$ asserts that each pattern must be distinct up to unification. By imposing this condition we ensure there is no overlap between different branches of a λ-case, and as such there can be at most one p_i that unifies with the left-hand side of the quotient's equality target. Intuitively, the above rule can be understood as first identifying a pattern p_i from a sequence p that unifies with the left-hand side of the equality target of a quotient. After applying the unifying substitution, this rule checks in the refinement logic whether the precondition of the quotient implies the corresponding expression e_i from a sequence e is equal to the right-hand side of the equality target applied to the lambda-case function. If the outlined condition is met, then the relation holds between the given sequences p, e and the considered quotient.

We next consider the inductive case, which simply extends the environment with a quantified variable and is characterised by the following rule:

$$\frac{\Gamma, v : \tau \models \lambda\ \{\ p \rightarrow e\ \} \rightsquigarrow Q}{\Gamma \models \lambda\ \{\ p \rightarrow e\ \} \rightsquigarrow (\textbf{forall}\ v : \tau\ \textbf{in}\ Q)}$$

The above definitions allow us to proceed with a formal characterisation of the λ-case formation rule for quotient types. In particular, this rule is given as follows:

$$\frac{\Gamma \models (x : \tau\ /\ Q) \rightarrow \sigma \qquad \Gamma \models \lambda\ \{\ p \rightarrow e\ \} \rightsquigarrow Q \qquad p_1,\ \ldots,\ p_k \text{ is a complete case analysis of } \tau}{\Gamma \vdash_{\mathbb{Q}} \lambda\ \{\ p_1 \rightarrow e_1; \ldots; p_k \rightarrow e_k\ \} : (x : \tau\ /\ Q) \rightarrow \sigma}$$

Note that we impose that any sequence of patterns used to construct a λ-case term must form a complete case analysis of the underlying type. The formal description of this idea is well understood for a type theory with algebraic data types, and we do not reintroduce it here. In the absence of this condition, the partiality of λ-case functions can be used to contradict the necessary correctness result for quotients. The λ-case formation rule is the crucial component of the typing rules for λ_Q and ensures that the equality constructors of quotient types are respected by functions that match on their input. Importantly, when only the total functions of λ_Q are considered it must always be the case that there exists a pattern p_i that unifies with the left-hand target of a quotient Q.

To conclude our introduction of the typing system of λ_Q, we formulate and prove a correctness result for quotients. Notably, soundness of quotient type checking follows directly from the soundness of the underlying refinement type system and the assumption that a liquid type judgement $\Gamma \vdash_{\mathbb{Q}} e : \tau$ implies $\Gamma \vdash e : \tau$. In particular, the proof of soundness for quotient type checking follows in the same manner as described for λ_L in [Rondon et al. 2008]. The correctness result for quotient type checking states that from the typing rules for λ_Q, we can conclude that function congruence correctly extends over quotients. Indeed, this is the essential and defining property of quotient types. In order to state this correctness result, we begin by defining precisely what it means for an arbitrary function on a quotient type to respect the relevant quotient.

Definition. Given an environment Γ and a function $\Gamma \vdash_{\mathbb{Q}} f : (x : \tau) \rightarrow \sigma$, we write $\Gamma \models f * Q$ to denote that f respects the quotient Q in the environment Γ, which is inductively defined by:

$$\frac{\Gamma, v : \tau \models f * Q}{\Gamma \models f * (\textbf{forall}\ v : \tau\ \textbf{in}\ Q)} \qquad \frac{\Gamma \vdash_{\mathbb{Q}} p : \tau \qquad \Gamma \vdash_{\mathbb{Q}} e : \tau \qquad \Gamma, \phi \models f\ p \equiv_{\sigma} f\ e}{\Gamma \models f * (\phi \Rightarrow p == e)}$$

The above definition directly corresponds to the functorial action of a function on an equality that is constructed by means of a quotient. An important property of the proposition $\Gamma \models f * Q$ is invariance with respect to context extension by a binding, which is captured as follows.

Proposition 1. Given $\Gamma \models f * Q$ and $\Gamma \vdash_{\mathbb{Q}} e : \tau$ then we can conclude $\Gamma, e : \tau \models f * Q$. This can be understood as a weakening rule for the property that every function must respect the quotients of its inputs. The proof follows by induction on the quotient Q:

- For $Q = \textbf{forall } v : \gamma \textbf{ in } P$, by the inductive hypothesis we can weaken $\Gamma, v : \gamma \models f * P$ to obtain $\Gamma, v : \gamma, e : \tau \models f * P$ as required;
- For $Q = \phi \Rightarrow p == h$, we apply the standard weakening rule for typing judgements to both $\Gamma \vdash_{\mathbb{Q}} p : \tau$ and $\Gamma \vdash_{\mathbb{Q}} h : \tau$ to extend their context with $e : \tau$, and similarly apply the weakening rule for equality judgements in λ_Q to $\Gamma, \phi \models f\ p \equiv_\sigma f\ h$.

An important property of the typing system of λ_Q and a key component in the proof of our correctness theorem is preservation of equality under quotient rewriting. That is, when a quotient $\phi \Rightarrow p == e$ appears in a context Γ then given any expression in Γ the two expressions that can be constructed by substituting a free variable for either p or e should be considered equal by the type system of λ_Q. We formally express this property in the following proposition.

Proposition 2. Given a typing judgement $\Gamma, v : \tau \ / \ (\phi \Rightarrow p == h) \vdash_{\mathbb{Q}} e : \sigma$, we can conclude $\Gamma; (\phi \Rightarrow p == h) :: \tau \models e[v \mapsto p] \equiv_\sigma e[v \mapsto h]$. The proof follows by induction on the expression e, for which the interesting cases proceeds as follows:

- For $e = x$, if $x = v$ then x must have type $\tau \ / \ (\phi \Rightarrow p == h)$ and we can conclude $p \equiv h$ from the equality checking rules of λ_Q;
- For $e = \lambda \ \{p_1 \to e_1; \ \ldots; \ p_k \to e_k\}$, for each p_i where v appears freely, applying either $[v \mapsto p]$ or $[v \mapsto h]$ to e does not change e_i and the equality trivially holds, otherwise we continue our induction on e_i in the appropriately extended context to obtain $e_i[v \mapsto p] \equiv_\sigma e_i[v \mapsto h]$ and finally we aggregate the resulting equalities in the obvious manner to obtain an equality between lambda-case functions.

In advance of stating our correctness result for quotients in λ_Q, we first provide a proof of a contextual version. In particular, in Proposition 3 we assume that we are working within a context Γ whereby all functions in Γ that are defined on quotient types respect the relevant quotient. From this assumption, we can prove that any other well-typed function on quotient types that can be constructed from this context will also respect its quotients.

Proposition 3. Given an environment Γ such that all functions in Γ respect the relevant quotient, i.e. every $g : (x : \alpha \ / \ P) \to \beta \in \Gamma$ satisfies $\Gamma \models g * P$, it follows that every constructible function $\Gamma \vdash_{\mathbb{Q}} f : (x : \tau \ / \ Q) \to \tau'$ has the property $\Gamma \models f * Q$. The proof follows by induction on Q and f:

- For $Q = \textbf{forall } v : \gamma \textbf{ in } R$, we first apply weakening (Proposition 1) to our initial assumption to conclude that every $g : (x : \alpha \ / \ P) \to \beta \in \Gamma$ has the property $\Gamma, v : \gamma \models g * P$, then we continue our induction with this new assumption alongside the extended environment $\Gamma, v : \gamma$ and the weakened term $\Gamma, v : \gamma \vdash_{\mathbb{Q}} f : (x : \tau \ / \ R) \to \tau'$;
- For $Q = \phi \Rightarrow p == h$, we consider each case for f:
  - For $f = x$ and $f = c$, the result follows from the initial assumption;
  - For $f = \lambda x.e$, after unfolding definitions it suffices to show that $\Gamma, \phi \models e[x \mapsto p] \equiv_{\tau'} e[x \mapsto h]$ which follows directly from Proposition 2;

- For $f = \lambda\ \{p_1 \rightarrow e_1;\ \ldots;\ p_k \rightarrow e_k\}$, by the condition that a well-typed λ-case must perform a complete case analysis, there must exist precisely one p_i such that $p_i \sim_\sigma p$ and it consequently suffices to show that $\Gamma, \phi \models e_i[\sigma] == \lambda\ \{p_1 \rightarrow e_1;\ \ldots;\ p_k \rightarrow e_k\}\ h[\sigma]$ and since σ is the most general unifier, this follows from the typing rule for λ-case functions.

Finally, we can now conclude our introduction of the core language λ_Q with the key correctness result for quotient types, which is captured by the following theorem.

Theorem 1 (*Correctness of quotienting*). Every closed function in λ_Q of the form $f : (x : \tau\ /\ Q) \rightarrow \sigma$ respects the relevant quotient Q, i.e. $\emptyset \models f * Q$. That is, function congruence correctly extends over quotients. The proof is given simply by specialising Γ to the empty context in Proposition 3.

6 SUBTYPING

Quotient types in the core language λ_Q come equipped with an ordering relation that we use to extend our type system with subtyping rules. In practice, these subtyping rules provide us with a framework for deciding when it is sound to substitute one quotiented type for another. As such, the subtyping rules for quotient types improve the reusability of code by allowing functions on quotient types to be applied to more 'weakly' quotiented types subject to the ordering relation. For example, a valid function on the propositional truncation of a type, i.e. the quotient which relates all terms, must also respect any other quotient on the same underlying type. In this section, we introduce the subtyping rules for quotient types in λ_Q.

The first of the λ_Q subtyping rules is for quotient generalisation, and asserts that any quotient type is a supertype of its underlying type. For example, the type of lists can be understood as a subtype of bags, or bags as a subtype of sets. This rule is formally expressed as follows:

$$\frac{\Gamma \vdash_{\mathbb{Q}} Q :: \tau}{\Gamma \vdash_{\mathbb{Q}} \tau <: \tau\ /\ Q}$$

We write $\tau <: \sigma$ above to denote that τ is a subtype of σ. When considered alongside the λ_Q typing rule for subtypes, this rule is equivalent to the introduction form for quotient types.

To introduce the next subtyping rule for quotient types, we first present an ordering relation on quotients such that $P <: Q$ means that P is a *subquotient* of Q, defined using four rules. The first rule specifies when the precondition and equality of a quotient are subsumed by another. For this rule, we make use of a join-semilattice on patterns, whereby $p \subseteq_\sigma q$ iff the pattern p is subsumed by q with substitution σ. Consequently, when $p \subseteq_\sigma q$ we have a definitional equality $q[\sigma] = p$. For example, given a constructor $S : \tau \rightarrow \tau$ and variables $v : \tau$, the term $S\ (S\ v)$ is subsumed by $S\ v$ with substitution $[v \mapsto S\ v]$. We write $\Gamma \models p \subseteq_\sigma q$ to denote that p is subsumed by q in context Γ, and introduce the following subquotient rule:

$$\frac{\Gamma \vdash_{\mathbb{Q}} p, q, e, f : \tau \qquad \Gamma \models p \subseteq_\sigma q \qquad \Gamma, \phi \models \psi[\sigma] \qquad \Gamma, \phi \models e \equiv_\tau f[\sigma]}{\Gamma \models (\phi \Rightarrow p == e) <: (\psi \Rightarrow q == f)}$$

In this definition, $\equiv_\tau$ is an extended notion of equality between terms of type τ that can make use of substitutions presented by quotients, which we will define in Section 7. Intuitively, the above rule states that if one quotient is implied by another in the refinement logic, then the former is a subquotient of the latter. For example, given a constructor $S : \tau \rightarrow \tau$ and variables $u, v : \tau$, we can conclude $(\textbf{true} \Rightarrow S\ (S\ (S\ v)) == S\ v) <: (\textbf{true} \Rightarrow S\ (S\ u) == u)$ using the substitution $[u \mapsto S\ v]$.

When two quotients quantify over a variable of the same type, the ordering relation is defined by the following rule, which simply extends the context by the quantified variables and checks whether the remaining body of one quotient is a subquotient of the other:

$$\frac{\Gamma, v : \tau, u : \tau \vdash_{\mathbb{Q}} P <: Q}{\Gamma \vdash_{\mathbb{Q}} (\textbf{forall}\ v : \tau\ \textbf{in}\ P) <: (\textbf{forall}\ u : \tau\ \textbf{in}\ Q)}$$

In contrast, when two quotients are quantified over types that are not definitionally equal, it is possible that one is a subquotient of the other, by means of the following permutation rule:

$$\frac{\Gamma,\ v_1 : \tau_1,\ \ldots,\ v_n : \tau_n \vdash_{\mathbb{Q}} P <: Q \qquad \Gamma \models \forall\ i.\ \tau_i \neq \tau_{\sigma(i)} \qquad \sigma \text{ is a permutation on } n}{\begin{array}{c}\Gamma \vdash_{\mathbb{Q}} (\textbf{forall}\ v_1 : \tau_1\ \textbf{in}\ \ldots\ \textbf{forall}\ v_n : \tau_n\ \textbf{in}\ P) <: \\ (\textbf{forall}\ v_{\sigma(1)} : \tau_{\sigma(1)}\ \textbf{in}\ \ldots\ \textbf{forall}\ v_{\sigma(n)} : \tau_{\sigma(n)}\ \textbf{in}\ Q)\end{array}}$$

In this rule, $\Gamma \models \forall\ i.\ \tau_i \neq \tau_{\sigma(i)}$ means that when τ is reordered under the permutation σ, the type in position i is not definitionally equal to the old type in the same position. This condition ensures that the rule does not overlap with the prior rule for quotients that quantify over the same type. That is, by imposing this constraint, we can translate these rules to well-founded induction on two quotients. Understood in this manner, an implementation of the above rule involves constructing two sequences of types $\tau_1, \ldots, \tau_n$ and $\rho_1, \ldots, \rho_n$, whereby we continue until there are no more quantifiers or we find $\tau_{n+1} = \rho_{n+1}$, and finally we check if ρ is a permutation of τ.

The final case we consider is when a subquotient quantifies over a variable and the superceding quotient does not. In particular, we introduce the following ordering rule for quotients:

$$\frac{\Gamma \models \tau \qquad \Gamma, v : \tau \vdash_{\mathbb{Q}} P <: (\phi \Rightarrow p == e) \qquad v \text{ not free in } \phi, p \text{ or } e}{\Gamma \vdash_{\mathbb{Q}} (\textbf{forall}\ v : \tau\ \textbf{in}\ P) <: (\phi \Rightarrow p == e)}$$

In particular, this rule holds because every quotient in λ_Q specifies a *proposition*. That is, we should consider **forall** $v : \tau$ **in** P to be analogous to $\forall\ (v : \tau).\ P$, rather than the dependent function type $\Pi\ (v : \tau)\ P$. If we were instead to consider a calculus for higher quotients, this subquotient rule would not be correct. It is important that the quantified variable is not free in each term of the superceding quotient, otherwise context extension may change its meaning.

From the definition of our ordering relation on quotients, we can extend our subtyping rules for quotient types in the language λ_Q with the following rule:

$$\frac{\Gamma \vdash_{\mathbb{Q}} P <: Q}{\Gamma \vdash_{\mathbb{Q}} \tau\ /\ P <: \tau\ /\ Q}$$

That is, a subquotienting relationship can be lifted directly to a subtyping relationship. Combining this rule with the subtyping rule for functions yields the following important typing derivation:

$$\frac{\Gamma \vdash_{\mathbb{Q}} f : (\tau\ /\ Q) \rightarrow \tau' \qquad \Gamma \vdash_{\mathbb{Q}} P <: Q}{\Gamma \vdash_{\mathbb{Q}} f : (\tau\ /\ P) \rightarrow \tau'}$$

That is, if we know that P is a subquotient of Q and that a function f respects Q, then it must also respect P. We can similarly derive a typing rule that allows us to apply any function on a quotient type to a term of the underlying type. For practical purposes, such as in Quotient Haskell, this means that only the most restrictive quotient is needed in the type definition of a function, which can subsequently be applied to terms inhabiting any subquotients.

Finally, the subquotient relationship can be shown to satisfy a correctness theorem that states that if a function respects a quotient Q it must also respect any subquotient of Q. In order to prove this correctness rule for the subquotienting relationship, we make use of Proposition 4, which is presented in Section 7. In particular, an equality $x \equiv y$ is invariant with respect to substitution. We can then proceed with our correctness theorem for the subtyping rules of λ_Q.

Theorem 2 (*Correctness of subquotienting*). Given a function $\Gamma \vdash_{\mathbb{Q}} f : (x : \tau) \to \tau'$ and quotients $\Gamma \models P, Q :: \tau$ such that $\Gamma \vdash_{\mathbb{Q}} P <: Q$, then if $\Gamma \models f * Q$ it follows that $\Gamma \models f * P$. The proof is by induction on the subquotienting relation, for which the interesting case proceeds as follows:

- For $\phi \Rightarrow p == e <: \psi \Rightarrow q == r$, unfolding the definitions for the judgement $\Gamma \models f * Q$ and the subquotient $P <: Q$ yields the following four assumptions:

$$(1)\ \Gamma, \psi \models f\ q \equiv_{\tau'} f\ r \qquad (2)\ p \subseteq_\sigma q \qquad (3)\ \Gamma, \phi \models e \equiv_\tau r[\sigma] \qquad (4)\ \Gamma, \phi \models \psi[\sigma]$$

  As stated in Proposition 4, a key property of equality is invariance with respect to term substitution, and consequently by function congruence and assumption (1) we can conclude $\Gamma, \psi[\sigma] \models f\ q[\sigma] \equiv_{\tau'} f\ r[\sigma]$. From assumption (2) it follows that $p := q[\sigma]$, and by function congruence and transitivity of equality we can conclude $\Gamma, \psi[\sigma] \models f\ p \equiv_{\tau'} f\ r[\sigma]$. We proceed by applying function congruence and symmetry of equality to (3) to show that $\Gamma, \psi[\sigma] \models f\ r[\sigma] \equiv_\tau f\ e$. Finally, by transitivity of equality and by generalising the guard predicate through (4), we can conclude $\Gamma, \phi \models f\ p \equiv_{\tau'} f\ e$, as required.

7 EQUALITY

Quotients that appear in a typing context introduce a substitution rule that can be used for deciding the equality of terms during type-checking. This rule changes the notion of equality between terms, and we write $x ==_\tau y$ to represent an equality between terms of type τ in the underlying refinement type system, and $x \equiv_\tau y$ to represent the extended notion of equality in λ_Q that can make use of substitutions presented by quotients. The precise definition of equality in the underlying refinement type system can vary, but it remains crucial that it is an equivalence relation. Furthermore, we assume several additional rules hold in the underlying refinement type system:

- *(Function congruence)* For every closed function $\emptyset \models f : (x : \tau) \to \tau'$, if we have an equality $\Gamma \models x ==_\tau y$ then we must also have an equality $\Gamma \models f\ x ==_{\tau'} f\ y$;
- *(Equality substitution)* For every term $\Gamma \vdash_{\mathbb{Q}} e : \tau$, every equality $\Gamma \models x ==_{\tau'} y$ and every variable v, it must hold that $\Gamma \models e[v \mapsto x] ==_\tau e[v \mapsto y]$;
- *(Substitution invariance)* For every equality $\Gamma \models x ==_\tau y$ and substitution σ, the equality must also hold under the substitution, i.e. $\Gamma \models x[\sigma] ==_\tau y[\sigma]$.

Crucially, while $x ==_\tau y$ is a proposition in the refinement logic, $x \equiv_\tau y$ is a judgement in the type system. Consequently, in λ_Q the definition of $\Gamma \models \phi$ differs from the underlying liquid type system. In particular, any equality $x ==_\tau y$ that appears in the logical proposition ϕ is lifted to the extended notion of equality $x \equiv_\tau$ y. For example, we check the judgement $\Gamma \models x == y \Rightarrow \psi$ by checking whether either $\Gamma \not\models x \equiv y$ or $\Gamma \models \psi$. Intuitively, our extended notion of equality can make use of quotients that appear in a context in order to transform the original equality by means of substitutions. This decidable process results in building a new equality in the refinement logic which can then be checked by an SMT-solver.

Importantly, our extended notion of equality is strictly weaker than that of the underlying liquid type system, and this property is expressed by the following rule:

$$\frac{\Gamma \vdash_{\mathbb{Q}} x, y : \tau \qquad \Gamma \models x ==_\tau y \qquad \tau \text{ is not a quotient type}}{\Gamma \models x \equiv_\tau y}$$

That is, two terms are considered equal in the language λ_Q if they are equal in the underlying refinement type system. Consequently, for any term that inhabits a type that has not been quotiented, the notion of equality in λ_Q and the underlying refinement logic is identical. However, for terms that inhabit a quotient type, it is necessary to extend our notion of equality to make use of the equalities introduced by quotients. To do this, we begin by giving a formal account of when a context is extended by a quotient in the process of equality checking.

In contrast to types and guard predicates, we make use of a more involved form of context extension for quotients that ensures a given context contains only the most general form of a quotient. As discussed later in this section, this is necessary to ensure termination of equality checking in our type system. As such, given a context Γ and quotient $Q :: \tau$, we write Γ, Q for regular context extension and $\Gamma; Q$ for the more involved notion. In order to define $\Gamma; Q$, we will consider three separate cases concerning whether Γ already contains a quotient $P :: \tau$ that is related to Q by a subquotient relationship. A complete account of the ordering rules for quotients is given in Section 6, and we write $Q <: P$ to mean that Q is a subquotient of P. We begin by considering the case when there does not exist a quotient $P :: \tau$ in Γ such that Q is related to P by means of the subquotient relationship. In this case, we simply apply the usual notion of context extension and as such we have $\Gamma; Q = \Gamma, Q$. The two remaining cases consider when there exists a quotient $P :: \tau$ in Γ such that either $Q <: P$ or $P <: Q$. In the case when $Q <: P$, then Γ is left unchanged and we define $\Gamma; Q = \Gamma$. In turn, for $P <: Q$ we replace P in Γ with Q. Using this notion of extending a context by a quotient, we can state the following equality rule for quotient types:

$$\frac{\Gamma \vdash_{\mathbb{Q}} x, y : \tau \mathbin{/} (\phi \Rightarrow p == e) \qquad \Gamma; (\phi \Rightarrow p == e) :: \tau \models x \equiv_{\tau} y}{\Gamma, \phi \models x \equiv_{\tau/(\phi \Rightarrow p == e)} y}$$

This rule states that two terms inhabiting a quotiented type are to be considered equal when they are equal as inhabitants of their underlying type in a context extended by their quotient. Note that the rule formulated above only considers quotients that do not contain bindings, and indeed, this is the only form of a quotient that may be used to extend a context. For the case of quotients that contain bindings, the following equality rule applies:

$$\frac{\Gamma \vdash_{\mathbb{Q}} x, y : \tau \mathbin{/} (\textbf{forall } v : \sigma \textbf{ in } Q) \qquad \Gamma, v : \sigma \vdash_{\mathbb{Q}} x \equiv_{\tau/Q} y}{\Gamma \vdash_{\mathbb{Q}} x \equiv_{\tau/(\textbf{forall } v:\sigma \textbf{ in } Q)} y}$$

That is, for quotients with bindings we simply consider equality in the context extended by the bindings. The final equality rule we introduce to the typing system of λ_Q corresponds to checking the equality of terms in a context containing a compatible quotient. This rule allows us to rewrite an equality using a quotient in a given context, and is formulated as follows:

$$\frac{\Gamma \vdash_{\mathbb{Q}} x, y, p, e : \tau \qquad \Gamma \models \phi[\sigma] \qquad \Gamma \models y \equiv_{\tau} e[\sigma] \qquad x \subseteq_{\sigma} p}{\Gamma; (\phi \Rightarrow p == e) :: \tau \models x \equiv_{\tau} y}$$

We write $x \subseteq_{\sigma} p$ here to denote that a term x is subsumed by the pattern p with substitution σ. With this in mind, the above rule states that any two terms $x, y : \tau$ are considered equal in a context Γ if there is a compatible quotient $\phi \Rightarrow p == e :: \tau$ in Γ such that p subsumes x with substitution σ, and such that y can be show to be equal in the underlying refinement type system to the term obtained by applying σ to e. Furthermore, we have an additional symmetric rule whereby we check if p instead subsumes y and $x \equiv_{\tau} e[\sigma]$, which is otherwise identical to the above rule.

In order to establish that our extended notion of equality is an equivalence on terms, we require an additional axiomatic rule for transitivity of equality as defined on quotient types:

$$\frac{\Gamma \vdash_{\mathbb{Q}} x, y, z : \tau \mathbin{/} Q \qquad \Gamma \models x \equiv_{\tau/Q} y \qquad \Gamma \models y \equiv_{\tau/Q} z}{\Gamma \models x \equiv_{\tau/Q} z}$$

Crucially, if the equality of the underlying refinement type system is an equivalence relation then so is our extended notion of equality in λ_Q. Reflexivity of equality follows evidently from the rule that every equality in the underlying refinement type system infers equality under our extended notion of equality. Transitivity is given by the axiomatic rule postulated above, while symmetry follows from the fact that the third equality rule can be symmetrised. Later in this article, we make use of the fact that our extended notion of equality is indeed an equivalence relation in correctness proofs for both quotients and a subquotienting relation.

A practical implementation for type-checking that applies the equality rules for λ_Q requires that we handle cases for which there is more than one compatible quotient. Importantly, the manner in which we extend a context by a quotient ensures that a context will only ever contain a single instance of a quotient. In addition, when a quotient is applied during equality checking it is removed from the context. When considered together, these properties ensure that every equality checking path will terminate for terms that inhabit a quotient type. As such, given that a context Γ can only contain a finite number of quotients of the form $Q :: \tau$, then if Γ has n such quotients there can be at most $n!$ different paths to consider for equality checking. This pathological case occurs when every permutation of quotients $Q_1 :: \tau, \ldots, Q_n :: \tau \in \Gamma$ corresponds to a is sequence of rewrites. For example, if the left-hand side of the equality produced by each Q_k were simply a variable, then each quotient would evidently unify with any term of type τ, and consequently every permutation would indeed specify a valid sequence of rewrites. In practice, however, the number of quotients that appear in a context for a type is typically small, and as such exhaustively checking each valid permutation does not present an issue even in the pathological case.

As an example of how the extended equality rule of λ_Q is applied in practice, recall that in Section 3 we defined a type `List` by quotienting an underlying type `Tree`. To check that the `map` function on trees refines to a function on lists, we must show it respects the quotients of `List`. One such quotient is the left-identity law `idl :: x:List a -> Join Empty x == x`, and to show that `map` respects `idl` we must show `map f (Join Empty x) == map f x`. After normalising the term on the left, the stated equality becomes `Join Empty (map f x) == map f x`. Importantly, when considering the refinement of `map` to a function on lists, this is an equality between terms of the quotient type `List`. Consequently, when checking whether this equality holds we first extend our context by the quotients of `List`, and then check if the pattern appearing on the left of the equality target of any of these quotients subsumes either `Join Empty (map f x)` or `map f x`. We can observe that the left identity law has precisely this form, as `Join Empty x` subsumes `map f x` with substitution `x` $\longmapsto$ `map f x`. Finally, we check whether `map f x == x`[`x` $\longmapsto$ `map f x`], or simply `map f x == map f x`, which holds definitionally. This same sequence of steps can similarly be used to show that `map` respects each of the quotients of `List` and as such `map` is refinable to a function on lists.

We conclude our discussion on the extended notion of equality used in the type system of λ_Q by discussing and giving a formal account of three essential properties of equality substitution, substitution invariance and function congruence. The first of these properties we consider is substitution invariance which is critical for the correctness of the subquotienting relation defined in Section 6. Concretely, we state this property as follows:

Proposition 4. Given a context Γ and a substitution σ, then for every equality $\Gamma \models x \equiv_\tau y$ we have $\Gamma \models x[\sigma] \equiv_\tau y[\sigma]$. In the case when τ is not a quotient type, the proof follows from the substitution invariance property of the underlying equality. Otherwise, the proof follows by inversion whereby we consider each of the extended equality rules.

In addition to being invariant under substitution, equality in λ_Q must also be respected by the substitution operator. Intuitively, this means that if two terms are equal then substituting them for a variable in any expression should produce two equal terms, and is stated as follows:

Proposition 5. For every equality $\Gamma \models x \equiv_\tau y$, term $\Gamma \vdash_{\mathbb{Q}} e : \sigma$ and any choice of variable u, we can construct an equality $\Gamma \models e[u \mapsto x] \equiv e[u \mapsto y]$. The proof follows by induction on e and is evident for each case after suitably unfolding the definition of substitution.

The final property of equality we consider is function congruence, which we describe in two parts. Firstly, in Proposition 6 we consider open terms whereby we assume that we are working in a context which respects the congruence law for functions. We then specialise to the empty context in Proposition 7 to state the true congruence property for closed functions.

Proposition 6. Given a context Γ with function congruence, i.e. for every function $f : (x : \tau) \to \tau' \in \Gamma$ and every equality $\Gamma \models x \equiv_\tau y$ we can construct an equality $\Gamma \models f\ x \equiv_{\tau'} f\ y$, then every constructible function in Γ must also obey the function congruence rule. In order to prove this we must show that for every function $\Gamma \vdash_{\mathbb{Q}} f : (x : \tau) \to \tau'$ and every equality $\Gamma \models x \equiv_\tau y$, we can conclude $\Gamma \models f\ x \equiv_{\tau'} f\ y$. The proof follows by induction on f and we consider only the interesting cases whereby f is either a λ function or a λ-case function. In both cases, after unfolding definitions we can observe that the proof follows evidently from Proposition 5.

Proposition 7. For every closed function $\emptyset \vdash_{\mathbb{Q}} f : (x : \tau) \to \tau'$ and every equality $\emptyset \vdash_{\mathbb{Q}} x \equiv_\tau y$, we can show that there is an equality $\emptyset \vdash_{\mathbb{Q}} f\ x \equiv_{\tau'} f\ y$. The proof follows immediately by simply instantiating the context in Proposition 6 to the empty context.

8 IMPLEMENTATION

In order to demonstrate the utility of the quotient types of the core language λ_Q in a practical setting, we have developed Quotient Haskell. In particular, Quotient Haskell is an extension to Liquid Haskell that adds quotient types through the mechanism of datatype refinement, and checks whether functions defined on quotient types respect the necessary equations by generating constraints for an SMT solver. More specifically, our implementation of Quotient Haskell makes notable changes to Liquid Haskell parsing, syntax trees, equality constraint generation for quotient types, and type-checking and constraint generation for case expressions.

The first step of the Liquid Haskell pipeline that is modified by our implementation is the parser. In particular, the parser is extended to support the syntax illustrated by the examples in Sections 2, 3 and 4. As expressed in these examples and the core language presented in Section 5, in Quotient Haskell we require that the left-hand side of an equality appearing in the codomain of an equality constructor is a *pattern*. In practice, this requirement ensures that when type-checking a case expression for which an element of a quotiented type is the scrutinee, proof requirements corresponding to equality constructors can be either unfolded or erased. The syntax for equality patterns in Quotient Haskell is formally specified by the following grammar:

lpat	::=	*var*	variables
	|	*literal*	literals
	|	*qcon* $lpat_1$ ... $lpat_k$	constructors ($k \geq 0$)
	|	*qcon* { $qvar_1$ = $lpat_1$, ..., $qvar_k$ = $lpat_k$ }	records ($k \geq 0$)
	|	(*lpat*)	parenthesised pattern
	|	($lpat_1$, ..., $lpat_k$)	tuples ($k \geq 1$)

In the above formulation of the *lpat* syntax, we write *var*, *literal*, *qvar*, *qcon* to denote the Haskell non-terminals for variables, literals, qualified variables and qualified constructors, respectively. The *lpat* rule can be understood as a more restricted version of the Haskell grammar rule for patterns that can appear on the left-hand side of a function. In particular, some patterns do not make sense for equality constructors, such as wildcards, irrefutable patterns and as-patterns.

We now turn to the syntax for equality constructors. For this definition, we make use of the Haskell non-terminals *expr* and *cxt*, which correspond to Haskell expressions and typeclass contexts. In addition, we use the Liquid Haskell non-terminal *rbind*, which corresponds to dependent bindings of a variable, which can appear to the left of an arrow. We also use *bpred*, which corresponds to bare predicates in Liquid Haskell, which are propositions surrounded by braces such as `{x == y}`. The syntax for an equality constructor is then given by the following rule, where $j, k \geq 0$:

$$\mathit{eqcons} \quad ::= \quad \mathit{var} \; \mathbf{::} \; [(\mathrm{cxt}) \; \mathtt{=>}] \; \mathit{rbind}_1 \; \mathtt{->} \cdots \mathtt{->} \; \mathit{rbind}_j \; \mathtt{->} \; \mathit{pred}_1 \; \mathtt{->} \cdots \mathtt{->} \; \mathit{pred}_k \; \mathtt{->} \; \mathit{lpat} \; \mathtt{==} \; \mathit{expr}$$

Notably, the *eqcons* rule allows for quantified variables to be constrained by typeclasses. A typeclass context that appears in an equality constructor can be used to further constrain when respectability theorems are generated. In particular, after matching a case alternative with the left-hand side of an equality, Quotient Haskell will only generate the resulting respectability theorem if there exists the necessary instances for each of the matched variables.

To define the syntax for quotient types, we use the Liquid Haskell non-terminal *stype*, which corresponds to refinement types excluding bindings and bare predicates, and the Haskell non-terminal *simpletype*, which corresponds to the name of a type followed by type variables. The syntax for quotient type definitions is then given by the following rule, where $k \geq 1$:

$$\mathit{quotty} \quad ::= \quad \mathbf{data} \; \mathit{simpletype} = \mathit{stype} \; |/ \; \mathit{eqcons}_1 \; |/ \cdots |/ \; \mathit{eqcons}_k$$

In addition to the above syntax for quotient constructors, Quotient Haskell has syntax for providing explicit proofs that equality constructors are respected, where $k \geq 0$:

$$\mathit{quotproof} \quad ::= \quad \mathbf{respects} \; \mathtt{<}\mathit{qvar}\mathtt{,}\mathit{qvar}\mathtt{>} \; \mathit{var} \; \mathbf{::} \; \mathit{rtype}_1 \; \mathtt{->} \cdots \mathtt{->} \; \mathit{rtype}_k \; \mathtt{->} \; \mathbf{\{} \; \mathit{expr} \; \mathtt{==} \; \mathit{expr} \; \mathbf{\}}$$

Quotient type definitions and explicit proofs of respectability, as specified by *quotty* and *quotproof* respectively, are only valid within a Liquid Haskell block.

To support the above extensions, the syntax trees within Liquid Haskell are modified to include both quotient constructors and explicit quotient respectability proofs. In Liquid Haskell, a datatype refinement must typically have the same name as the datatype being refined. However, for quotient types the name must instead be unique with respect to any other type constructors in scope. As such, we extend the type-checking environment with the names of any quotient type constructors. Moreover, the names of quotient constructors must be unique with respect to any term identifiers in scope. After parsing and construction of the type-checking environment, each of the required checks associated with quotients are performed during the existing refinement checking phase. In the remainder of this section we describe the core steps involved in quotient type checking.

Quotient Wellformedness. The wellformedness check for each quotient ensures that the domain of each equality constructor is well-formed, and that both sides of the target equality are terms of the underlying type. We also check that both the name of a quotient type and its quotient constructors are unique in the scope. The name of a quotient type must be unique with respect to type constructors, while quotient constructors must be unique with respect to term identifiers.

Equality Lifting. As described in Section 7, to make use of the equalities introduced by quotients within arbitrary refinements, it is necessary to extend the notion of equality between terms that inhabit a quotient type. To do this, Quotient Haskell includes a transformation step that traverses every refinement type in the type-checking environment. We proceed with a description of this transformation as it applies to each refinement type. Firstly, we consider the underlying refinement predicates which contain an equality between terms of a quotient type. For each such equality, we generate a set of implication expressions by rewriting the equality with suitable quotients, as described in Section 7, and adding the preconditions of each used quotient as the antecedent of the implication. We then replace this equality with the generated disjunction when constructing the constraint to be passed to the SMT solver. For example, consider the following quotient type that represents a list of integers for which any integer ≤ 0 acts as a unit:

```
data Positives
  = [Int]
  |/ unit :: xs:[Int] -> x:{ x:Int | x <= 0 } -> x :: xs == xs
```

Intuitively, we can understand this type as ensuring that we can only consider the strictly positive integers within a list. In addition, we consider a function `f` of the following form:

```
f :: xs:Positives -> x:{ x:Int | (x + 1) :: xs == xs } -> ...
```

The precise codomain and definition of `f` are not important here. Instead, we focus on the argument `x`, which has a refinement type whose predicate is given by an equality between terms of a quotient type. In particular, `x` must be an integer such that when `x+1` is prepended to a term of type `Positives` it does not change the list. This equation evidently does not hold for the underlying type of lists of integers. However, as we are considering an equality between terms of a quotient type, we must also consider the equality constructors, in this case `unit`. The equality lifting transformation is precisely the approach by which Quotient Haskell includes equality constructors in its logic.

Equality lifting involves considering every possible sequence of rewrites by means of suitable equality constructors. In particular, an equality constructor can be used to rewrite an equation of the form `x == y` precisely when the left-hand side of the equality constructor's target equality unifies with either `x` or `y`. This rewrite occurs by first applying the unifying substitution to both equalities, then composing them, and finally building an implication from the preconditions of the equality constructor to the composite equality. For example, this approach can be used to rewrite the equation `(x + 1) :: xs == xs` using the `unit` equality constructor. This rewrite is possible because `(x + 1) :: xs` unifies with the left-hand side `x :: xs` of the target equality of `unit`, with substitution `x` := `x + 1`. We then proceed to apply this substitution to the equality target of `unit` to obtain the equation `(x + 1) :: xs == xs`. Finally, we compose the two equations and build an implication from the precondition of `unit` to the composite equality and obtain the proposition `{ x + 1 <= 0 } => xs == xs`. Notably, in Liquid Haskell this logical proposition is simply equivalent to `{ x + 1 <= 0 }`. This proposition is combined by disjunction with all other valid sequences of rewrites. For example, after equality lifting is applied, the type of the term `f` becomes

```
f :: xs:Positives -> x:{ x:Int | (x + 1) :: xs == xs || x + 1 <= 0 } -> ...
```

Note that rewriting using a quotient may change the number of free variables in a refinement predicate. Consequently, this step occurs when generating the constraints from refinement predicates, in which predicates and their variables are used to construct independent definitions.

Quotient Respectability. Our implementation of Quotient Haskell covers two possible cases when considering quotient respectability: refined functions that take a term of a quotient type as input and do not match on that input, and case expressions that match on a single variable. We are interested only in cases where we have been provided with a type declaration that asserts that the input being considered must inhabit a quotient type. For functions that do not match on their quotiented input, we require no additional checks beyond those imposed by Liquid Haskell. As such, we only need consider matching functions and case expressions.

In the GHC API that is used by Liquid Haskell, both matching functions and case expressions are represented by single argument case expressions that match at most one level deep. In particular, we intuitively think of this representation as taking the form

```
let x = e in case e of { p_1 -> e_1; ...; p_k -> e_k }
```

where each p_i is either a variable or a constructor applied to a finite number of variables. In the proceeding transformation phase, Liquid Haskell transforms such case expressions into a sequence of conditional expressions, which make use of both distinguished testing predicates and selection functions for constructors to remove the let-binding and the free variables introduced by each case pattern. However, in the checking phase, the prior representation of case expressions are maintained and this representation is more suitable for checking quotient respectability.

A particularly useful consequence of working with case expressions that only permit matching on a single layer is the simplification of the unification check between the patterns of the case expression and the left-hand side of a quotient's equality target. In particular, we need only check whether either pattern is a mere variable, or the leading constructors match. The construction of the unifying substitution is similarly straightforward in this setting. With this in mind, we proceed to check quotient respectability of single argument case expressions as follows:

- We first check if the type of the scrutinee of the case expression is in the typing environment, and if this is a quotient type we proceed to check respectability, otherwise we are done;
- For each match $p \rightarrow e$ of the case expression we filter the relevant quotients by whether the left-hand side of their target equality unifies with p;
- For each quotient that passes the filter we apply the unifying substitution to both the right-hand side of the quotient's target equality and the expression e;
- We extract the refinements from the quotient's domain, from which we construct a logical implication that states that if the conjunction of these refinements hold then e is equal to the original case expression applied to the right-hand side of the quotient's target equality;
- If e is known to inhabit a quotient type, then for each quotient such that e unifies with the left-hand side of its equality target we apply the equality lifting transformation;
- We include the constructed constraint in the `.smt2` output generated by Liquid Haskell.

Note that the constraint generated by the above procedure will be a proposition quantified over the free variables that appear in the disjunction of the generated logical expressions. It is important that we normalise the terms that appear in the generated constraints, and as such the proof-by-logical-evaluation feature of Liquid Haskell is always enabled in Quotient Haskell.

To demonstrate the above procedure, we consider the example of the `map` function as refined on the quotient type `List` given in Section 3. In particular, only the definition

```
map f (Join x y) = Join (map f x) (map f y)
```

is relevant here, as the remaining definitions for `map` do not match with any of the quotients of `List`. When checking whether `map` is well-typed when refined to `List`, the first step of the above procedure is to check respectability, because `List` is a quotient type. In the next step, we filter the quotients of `List` to obtain only those that match with `Join x y`, which in this case happens to be every quotient for `List`. After the third step of the procedure, we obtain equations corresponding to the quotients `idl`, `idr` and `assoc`. For example, after proof-by-logical-evaluation the equation generated for `idl` is `Join (map f x) Empty == map f x`. As the quotients of `List` do not contain any refinements in the domain, we continue with the quotient substitution step. In the proceeding steps, we first confirm that the result type of `map` is a quotient type, which in this case is again `List`. We then apply all possible rewrites, as constructed from the quotients of `List`, to each of the generated equations. For example, we can rewrite `Join (map f x) Empty == map f x` using the `idl` quotient to obtain a new equation `map f x == map f x`. Finally, we generate constraints by taking the disjunction of equations built for each quotient. Continuing with our example of `idl`, we would generate a respectability constraint of the following form:

```
f:(a -> b) -> x:Mobile a ->
   Join (map f x) Empty == map f x || map f x == map f x || ...
```

This constraint will pass the SMT checking phase because the expression `map f x == map f x` holds definitionally. Consequently, the `map` function is shown to respect the `idl` function, and similar constraints are generated and shown to hold for both `idr` and `assoc`.

Quotient Subtyping. In practice, when types are checked rather than being inferred, subtyping rules need only be considered when type-checking function application. Because quotient types cannot yet be inferred in Quotient Haskell, this translates to a simple extension of the function application case of Liquid Haskell's constraint generation algorithm. In particular, our extension is relevant in the case when the function being applied takes a quotient type as input, and the argument it is applied to is of a different quotient type. We proceed by checking whether for every quotient of the argument's type, there is a quotient of the function's input type that supercedes it with respect to the ordering relation on quotients given in Section 6. If this is the case, then we retype the argument to the input type of the function and continue with constraint generation. Otherwise, constraint generation fails and we report a type-checking error.

To reduce the performance cost of automatically deciding quotient subtyping, it is possible to cache the computed results of the subtyping relation for quotient types. In particular, quotient types in Quotient Haskell can be uniquely identified by their name, and caching can hence be achieved using a map from pairs of names to the results of the subtyping relation. Consequently, the performance cost of automatically inferring subtyping between quotient types is primarily determined by a one-time check of the subquotienting rules described in Section 6. In practice, this means that automatically inferring subtyping for quotient types does not usually have a significant impact on the runtime of the type checker, while improving ease of use.

Performance of Type Checking. The implementation of Quotient Haskell includes simple optimisations such as caching of the subtyping relation for quotient types, and erasure of trivial respectability theorems. In practice, we have found that the additional typing features of Quotient Haskell do not usually have a significant impact on time performance. For most practical examples, including those presented in this article, time performance is dominated by the external SMT solver.

9 RELATED WORK

Refinement and quotient types, along with their implementations, have been extensively studied in the literature. This prior work underpins the development of Quotient Haskell, which can be understood as extending a refinement type system with quotient types.

Refinement Types are types equipped with a subtyping predicate from an SMT-decidable logic [Bengtson et al. 2011; Rushby et al. 1998]. As such, refinement types utilise a restricted form of dependency in which bound variables can appear in the body of a predicate. Implementations of refinement type systems have been developed for many popular languages, including ML [Freeman and Pfenning 1991], OCaml [Kawaguchi et al. 2010], and Haskell [Vazou et al. 2013]. Our core language λ_Q is introduced as a conservative extension to a generic underlying refinement type system, and supports the typing extensions of our practical implementation Quotient Haskell. The key idea is to translate the equational laws required by functions defined on quotient types into predicates in the underlying refinement logic. With this translation, we can utilise any suitable solver for the refinement logic to assist in the proof of quotient laws. Moreover, in order to make use of the equations described by quotients, we presented an extension of equality from a mere proposition in the refinement logic to a statement in the type system.

Liquid Haskell is an implementation of a bounded liquid type system [Rondon et al. 2008; Vazou et al. 2015] for Haskell, which adds termination checking to ensure correctness of refinement typing in a lazy setting [Vazou et al. 2014]. As introduced in this article, Quotient Haskell is an extension of the Liquid Haskell type system with quotient typing rules from the core language λ_Q. By developing Quotient Haskell as an extension to Liquid Haskell, quotients and subtypes can be utilised together and the elimination laws for quotients can make use of existing automation and rewriting developed for Liquid Haskell [Grannan et al. 2022; Vazou et al. 2017].

Quotient Types are types that are equipped with a distinguished notion of equality that may differ from the trivial, definitional equality for that type [Hofmann 1995; Li 2015]. Developing a well-behaved theory of quotient types for dependently typed languages has been an ongoing subject in the literature [Abbott et al. 2004; Nogin 2002]. A key difficulty that arises when introducing quotient types to intensional type theories is the preservation of canonicity. In particular, when quotients are added to a type system by means of axiomatic rules, it may be possible to construct closed terms that do not compute to a canonical form by means of the elimination rule for the equality type. This axiomatic approach to quotients is precisely the approach adopted by the type system of λ_Q. However, this issue does not arise in λ_Q, because equality does not constitute a type with its own elimination form but rather a judgement in the type system.

Quotient types can be further generalised to quotient inductive families, quotient inductive-recursive types, or quotient inductive-inductive types [Altenkirch et al. 2018; Altenkirch and Kaposi 2016; Kaposi et al. 2019]. Of particular note is quotient inductive families, whereby an inductive family can be thought of as a generalised algebraic datatype (GADT) that can additionally be indexed by a type, for example lists indexed by their length. A possible extension to λ_Q is the addition of typing rules for generalised algebraic quotient types, which in turn would describe how to extend Quotient Haskell with quotients for GADTs.

Implementations of Quotient Types include the higher-inductive types of Cubical Agda [Vezzosi et al. 2019], the HoTT library of Lean [van Doorn et al. 2017] and the axiomatic quotients in the Lean standard library [de Moura et al. 2015], a quotient types library for Coq [Cohen 2013], various implementations in Isabelle [Kaliszyk and Urban 2011; Paulson 2006; Slotosch 1997], quotient types by means of equivalence relations in NuPRL [Constable et al. 1986], and laws in Miranda [Thompson 1986]. Existing implementations have largely focused on the use of quotient types in proof assistants. However, several practical use cases for quotient types have been highlighted in the literature that may translate to general-purpose functional programming, such as the Boom hierarchy [Meertens 1983] and domain-specific languages with equational laws [Altenkirch and Kaposi 2016]. A crucial drawback of existing implementations for quotient types are the manual proof obligations required by every function that is defined on a quotient type. In practice, these proof obligations can become unnecessarily burdensome, especially in cases where the proofs can be derived in a systematic manner. Quotient Haskell, and the core language λ_Q, were developed to address this drawback by making use of the well-known theory of refinement type systems to introduce quotients whose elimination laws can be handled by a suitable solver for the underlying refinement logic.

Automation for Quotient Types has been explored in the Isabelle quotient package, with a focus on transferring terms and properties from an underlying type to a corresponding quotient type [Huffman and Kunčar 2013; Kaliszyk and Urban 2011]. In particular, implicit coercion from an underlying type to a quotient type is termed ‘lifting’, while automatically translating properties from one to the other is referred to as ‘transfer’. As a consequence of quotient types being implemented as an internalised package, the automation of lifting and transfer is achieved in Isabelle by non-trivial proof automation tactics. In contrast, Quotient Haskell adds support for quotient inductive types by extending the Liquid Haskell type-checker. Consequently, in Quotient Haskell the lifting property simply arises from the introduction rule for quotient types, and the transferal of proofs follows from the definition of the conservative extension of equality in Section 7. As such, while the automation for quotient types in Isabelle addresses the problems of lifting and transfer, Quotient Haskell instead addresses the problem of automating the proofs of respectability theorems for quotient types.

NuPRL is a proof development system that provides support for both quotient types and proof automation [Constable et al. 1986; Nogin 2002]. In NuPRL, quotient types are constructed by defining a new equality for a type using an equivalence relation, and this construction is provided as an operation on types. However, this approach is known to have a key drawback, namely that quotients of inductive types with a non-finite number of inductive positions often require the axiom of choice to construct their elimination map. In contrast, the quotient inductive types of Quotient Haskell do not have this issue. For example, such types can be used to prove properties of the Cauchy real numbers [Univalent Foundations Program 2013] and the weak delay monad [Chapman et al. 2019] without the need for the axiom of choice. Moreover, in contrast to Quotient Haskell, the quotient operation in NuPRL requires that a user constructs an explicit equivalence relation with proof witnesses for the necessary laws.

10 REFLECTION

Existing systems that support quotient types have primarily focused on the goal of formalising mathematical theories. Quotient Haskell adds a new point to the design space, focusing on the use of quotient inductive types in a general purpose programming language. This is achieved by integrating quotient types into Haskell in a manner that reduces proof obligations arising from their use, thereby allowing users to focus on programming rather than proof. In this section, we reflect on the design, practical use and limitations of the Quotient Haskell system.

Usability of quotient types in a general purpose language was the guiding principle in the design choices of Quotient Haskell. There are three key design choices that were made. First of all, and most importantly, the system is built on top of a refinement type system, in this case Liquid Haskell. This approach allows us to take advantage of existing work and infrastructure on proof automation for refinement types, such as the generation of subtyping constraints and their translation into an SMT-decidable form. Moreover, extending Liquid Haskell with support for quotient types allows for interoperability with subtypes, which provides a more expressive type system while retaining the benefits of proof automation. In practice, many programming use cases involve both quotient types and subtypes, so being able to use them in combination is important.

Secondly, the system supports a particular class of quotient types known as quotient inductive types. This approach allows users to define equational laws alongside an underlying data type without requiring the explicit construction of an equivalence relation. In particular, quotient inductive types can be understood as extending the notion of equality in a manner that implicitly preserves its underlying properties, such as the equivalence relation and function congruence laws. As we have seen in the practical example sections, using quotient inductive types provides a simple and natural approach to defining types with equational laws.

And finally, the system introduces syntax and typing rules for quotient types that allow the reuse of existing data definitions without the need to redefine the constructors. For example, the type of mobiles is defined in Section 2 by quotienting an existing type for trees, with the same constructor names `Leaf` and `Bin` used for both the original and quotiented type. Crucially, this means that any function on mobiles can also be used on trees, without the need to explicitly convert between the two types. In contrast, existing systems that implement quotient inductive types, such as Cubical Agda, require unique data constructors for each type. Consequently, functions on a quotient type cannot be reused on a type that only holds the underlying data without explicit conversion. For example, in the case of mobiles, this would mean introducing new constructors for the mobile type, and using conversion functions when applying a function on mobiles to trees.

Applications were the central motivation for the development of Quotient Haskell. In addition to the applications presented in Sections 2, 3 and 4 we have explored a range of further examples, including modular arithmetic, coordinate systems, efficient data structures, and domain specific languages with equational laws. We briefly describe two of these examples below, and plan to focus on additional applications of quotient types as part of our further work.

Polar coordinates are typically represented by a pair of numbers corresponding to a magnitude and an angle. For example, in Haskell we might choose to define `type Polar = (Double, Int)`, where the angle is given to one degree of accuracy. However, this definition allows multiple representations of the same point in the space. In particular, the point represented by `(r, a)` can also be represented by `(-r, -a)`, and by any pair constructed by adding or subtracting a multiple of 360 degrees. This problem can be avoided by bounding the two polar components using subtyping, by defining `type Magnitude = {r : Double | r >= 0}` and `type Angle = {a : Int | a >= 0 && a < 360}`. However, this representation brings its own problem, namely that operations on angles may need to normalise their results. To avoid this issue, we can represent polar coordinates as a quotient type, which can be achieved in Quotient Haskell as follows:

```
data Polar
  =  (Magnitude, Int)
  |/ turn :: r:Magnitude -> a:Int -> (r, a) == (r, a `mod` 360)
```

In this definition, the `turn` equality constructor captures the property that the full rotation of a point around the origin does not change that point. To illustrate the utility of representing polar coordinates as a quotient type, consider the following rotation operation:

```
rotate :: Int -> Polar -> Polar
rotate x (r, a) = (r, a + x)
```

This definition is well-typed in Quotient Haskell, because the `turn` equality constructor is respected as a consequence of eliminating into the `Polar` type. In contrast, the definition would be ill-typed if polar coordinates were instead represented by the type `(Magnitude, Angle)` of bounded magnitudes and angles. Making the definition type correct would require that the `rotate` function normalises the resulting angle, which impacts on both simplicity and efficiency.

Another application of quotient types, and in particular quotient inductive types, is the representation of domain-specific languages with equational laws. To demonstrate this, we consider how Quotient Haskell can be used to add η-expansion to the (untyped) lambda calculus. In particular, we use a de Bruijn representation where variables are natural numbers given by `type Nat = {n : Int | n >= 0}`, and define terms of the lambda calculus as follows:

```
data Expr = Var Nat | Lam Nat Expr | App Expr Expr
```

To state the η-expansion law, we require a predicate `isNotFree :: Nat -> Expr -> Bool`, which asserts that a given variable is not free in a given term. This predicate can easily be defined by induction on the terms of `Expr`. Using these two definitions, we can then define the following quotient type in which η-expansion is explicitly given by an equality constructor:

```
data LambdaExpr
  =  Expr
  |/ eta :: f:LambdaExpr -> v:Int -> {isNotFree v f} -> Lam v (App f v) == f
```

A key operation for the lambda calculus is β-reduction, which for the underlying type `Expr` can take the form of a function `reduce :: Expr -> Expr`. A correct definition of `reduce` should have the property `reduce (Lam v e) == Lam v (reduce e)`. Indeed, this property typically forms the defining equation for `reduce` in the `Lam` case. As such, if we assume a correct definition, the type of `reduce` can be refined to `LambdaExpr -> LambdaExpr` in Quotient Haskell. Notably, this refinement is only possible by eliminating into `LambdaExpr` and not `Expr`. To construct a well-typed function from `LambdaExpr` to `Expr` would require η-expansion to be explicitly applied.

Limitations are an important practical consideration for users of Quotient Haskell. Below we consider a number of limitations and their practical consequences.

First of all, without function extensionality, i.e. the principle that two functions are equal if they are equal for all arguments, it is not possible to prove some equalities that might otherwise be expected to hold. In practice, this issue can arise when the type checker attempts to resolve an equality that involve higher-order functions. For example, in Liquid Haskell the equation `map (1+) xs == map (+1) xs` cannot be shown without function extensionality. Naively extending Liquid Haskell by adding function extensionality as an axiom is known to be inconsistent, however, solutions to this problem have been explored [Vazou and Greenberg 2022]. At present, Quotient Haskell does not support reasoning with function extensionality when checking respectability theorems for quotient types. As we have seen, the lack of this feature does not preclude interesting and useful examples of quotient types, but we plan to consider in future work how function extensionality can be added to Quotient Haskell without compromising consistency.

Secondly, quotients of Generalised Algebraic Data Types (GADTs) are not currently supported in Quotient Haskell. GADTs allow inductive types to be indexed by other types, which enhances the expressivity of the type system by allowing data constructors to make use of additional type information. For example, a GADT of kind `Expr :: Type -> Type` that represents a simple form of well-typed expressions can be defined as follows:

```
data Expr a where
  Val :: a -> Expr a
  Add :: Expr Int -> Expr Int -> Expr Int
  Eq :: Expr Int -> Expr Int -> Expr Bool
  If :: Expr Bool -> Expr a -> Expr a -> Expr a
```

In particular, the type of each constructor ensures that it can only be applied to arguments of suitable types, ensuring that expressions are always well-formed. Support for quotients of GADTs would allow equations to be introduced between expressions indexed by the same type. In the case of `Expr`, this would allow equations such as the following to be added:

```
  ifTrue :: t:Expr a -> f:Expr a -> If (Val True) t f == t
  commute :: m:Expr Int -> n:Expr Int -> Add m n == Add n m
```

Such equations would then need to be respected by functions defined on expressions, such as a well-typed evaluation function `eval :: Expr a -> a`. Adding support for GADTs to Quotient Haskell would enhance the space of examples that can be considered.

Thirdly, while the underlying logic of both Liquid Haskell and Quotient Haskell is SMT-decidable, the type checker may still be unable to automatically prove some statements. This can occur for two key reasons: generated constraints may not be solvable in reasonable time, and theorems that require inductive reasoning cannot always be solved. The first of these problems can require users to make design choices to minimise the complexity of generated constraints. In Quotient Haskell, this can mean designing equality constructors with fewer quantified variables and preconditions.

Further work on automatically optimising and minimising generated SMT constraints can assist in reducing the design burden for users. Meanwhile, the second problem can be observed in Liquid Haskell by considering associativity of list concatenation, which cannot be automatically proved. This issue is similarly present in Quotient Haskell, and while proof by logical evaluation (PLE) can assist in some cases, there is not yet a precise classification of which inductive proofs can be automated. In practice, this means that generated respectability theorems that require inductive reasoning to prove will often require manual proofs. Quotient Haskell provides such a manual proof mechanism for respectability theorems, as demonstrated in Section 2.

And finally, at present Quotient Haskell has a limited error reporting system. In the case that a respectability theorem cannot be proven by the type checker and is not manually provided, a user is given a simple error message describing which equality constructor was not respected, followed by a generic Liquid Haskell error detailing the constraints. Improvements to the error reporting system of Quotient Haskell are a subject for future development work.

11 CONCLUSION AND FURTHER WORK

In this article, we presented a core language that supports practical programming with quotient types. This is achieved by extending an established core language that supports liquid types with a class of quotients whose proof obligations can be automatically discharged by the type checker. In particular, this class has the property that the equational laws that functions on quotients must satisfy can be translated into a collection of constraints that can be decided by an SMT solver. Furthermore, we showed how the equations constructed by quotients can be exploited by the type system, which is an essential aspect of having proper support for quotient types. More specifically, we extended the notion of equality in the refinement logic of the underlying liquid type system by adding substitution rules corresponding to each quotient.

The above ideas are realised in practice by Quotient Haskell, a proof-of-concept extension of Liquid Haskell with quotient types. We presented a range of examples demonstrating the use of quotients for practical programming, including mobiles (commutative binary trees), the Boom hierarchy (lists, trees, bags and sets), and the rational numbers.

There are many interesting topics for further work. First of all, at present Quotient Haskell requires explicit typing declarations when using quotient types, and cannot infer them from untyped expressions. As such, a possible improvement is to extend the constraint generation phase to include the possibility of typing judgements that include quotients, and consequently to allow quotient types to be automatically inferred. Secondly, we could generalise the range of types that can be quotiented by including additional features of (Liquid) Haskell, such as GADTs and refinement polymorphism. Moreover, we could also generalise the subquotient relationship given in Section 6 to relate a wider range of quotients, and hence improve the reusability of functions defined on quotient types. And finally, it is important to consider possible improvements to the practical aspects of the system, such as error messages and IDE support.

ACKNOWLEDGEMENTS

We would like to thank Patrick Bahr and the anonymous reviewers for many useful comments and suggestions that significantly improved the article.

REFERENCES

Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. 2004. Constructing Polymorphic Programs with Quotient Types. In *Proceedings of the International Conference on Mathematics of Program Construction.*

Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. 2018. Quotient Inductive-Inductive Types. In *Proceedings of the Conference on Foundations of Software Science and Computation Structures.*

Thorsten Altenkirch and Ambrus Kaposi. 2016. Type Theory in Type Theory Using Quotient Inductive Types. In *Proceedings of the Symposium on Principles of Programming Languages*.

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D Gordon, and Sergio Maffeis. 2011. Refinement Types for Secure Implementations. *ACM Transactions on Programming Languages and Systems* 33, 2 (2011).

James Chapman, Tarmo Uustalu, and Niccolò Veltri. 2019. Quotienting the Delay Monad by Weak Bisimilarity. *Mathematical Structures in Computer Science* 29, 1 (2019), 67–92.

Cyril Cohen. 2013. Pragmatic Quotient Types in Coq. In *Proceedings of the International Conference on Interactive Theorem Proving*.

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. 1986. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., USA.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Proceedings of the International Conference on Automated Deduction*.

Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the Conference on Programming Language Design and Implementation*.

Zachary Grannan, Niki Vazou, Eva Darulova, and Alexander J Summers. 2022. REST: Integrating Term Rewriting with Program Verification (Extended Version). (2022). arXiv preprint arXiv:2202.05872.

Brandon Hewer. 2023. Quotient Haskell. https://github.com/brandonhewer/QuotientHaskell/tree/develop.

Martin Hofmann. 1995. A Simple Model for Quotient Types. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*.

Brian Huffman and Ondřej Kunčar. 2013. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In *Proceedings of the International Certified Programs and Proofs*.

Cezary Kaliszyk and Christian Urban. 2011. Quotients Revisited for Isabelle/HOL. In *Proceedings of the Symposium on Applied Computing*.

Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. 2019. Constructing Quotient Inductive-Inductive Types. *Proceedings of the ACM on Programming Languages* 3, POPL (2019).

Ming Kawaguchi, Patrick M Rondon, and Ranjit Jhala. 2010. Dsolve: Safety Verification via Liquid Types. In *Proceedings of the International Conference on Computer Aided Verification*.

Jan Willem Klop, Vincent van Oostrom, and Roel de Vrijer. 2008. Lambda Calculus with Patterns. *Theoretical Computer Science* 398 (2008).

Nuo Li. 2015. *Quotient Types in Type Theory*. Ph.D. Dissertation. University of Nottingham.

Lambert Meertens. 1983. Algorithmics: Towards Programming as a Mathematical Activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*.

Aleksey Nogin. 2002. Quotient types: A Modular Approach. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 263–280.

Lawrence C Paulson. 2006. Defining Functions on Equivalence Classes. *ACM Transactions on Computational Logic* 7, 4 (2006).

Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of Conference on Programming Language Design and Implementation*.

John Rushby, Sam Owre, and Natarajan Shankar. 1998. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering* 24, 9 (1998).

Oscar Slotosch. 1997. Higher Order Quotients and their Implementation in Isabelle HOL. In *Proceedings of the International Conference on Theorem Proving in Higher Order Logics*.

Simon Thompson. 1986. Laws in Miranda. In *Proceedings of the ACM Conference on LISP and Functional Programming*.

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

Floris van Doorn, Jakob von Raumer, and Ulrik Buchholtz. 2017. Homotopy Type Theory in Lean. In *Proceedings of the International Conference on Interactive Theorem Proving*.

Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover*. Ph.D. Dissertation. University of California San Diego.

Niki Vazou, Alexander Bakst, and Ranjit Jhala. 2015. Bounded Refinement Types. In *Proceedings of the International Conference on Functional Programming*.

Niki Vazou and Michael Greenberg. 2022. How to Safely Use Extensionality in Liquid Haskell. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*. 13–26.

Niki Vazou, Patrick M Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *Proceedings of the European Symposium on Programming*.

Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the International Conference on Functional Programming*.

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G Scott, Ryan R Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *Proceedings of the ACM on Programming Languages* 2, POPL (2017).

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019).

Received 2023-07-11; accepted 2023-11-07