Coinduction Inductively

Mechanizing Coinductive Proofs in Liquid Haskell

Lykourgos Mastorou
National Technical Univ. of Athens
Athens, Greece
IMDEA Software Institute
Madrid, Spain
lykmast@softlab.ntua.gr

Nikolaos Papaspyrou
National Technical Univ. of Athens
Athens, Greece
Google
Munich, Germany
nickie@softlab.ntua.gr

Niki Vazou
IMDEA Software Institute
Madrid, Spain
niki.vazou@imdea.org

Abstract

Liquid Haskell is an inductive verifier that cannot reason about codata. In this work we present two alternative approaches, namely indexed and constructive coinduction, to consistently encode coinductive proofs in Liquid Haskell. The intuition is that indices can be used to enforce the base case in the setting of classical logic and the guardedness check in the constructive proofs. We use our encodings to machine check 10 coinductive proofs, about unary and binary predicates on infinite streams and lists, showcasing how an inductive verifier can be used to check coinductive properties of Haskell code.

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

Keywords: coinduction, refinement types, theorem proving

ACM Reference Format:
Lykourgos Mastorou, Nikolaos Papaspyrou, and Niki Vazou. 2022. Coinduction Inductively: Mechanizing Coinductive Proofs in Liquid Haskell. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium (Haskell '22), September 15–16, 2022, Ljubljana, Slovenia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3546189.3549922

0 Introduction

Consider a rewrite rule for map-fusion on infinite streams:

```
data Stream a = a :> Stream a
smap f (x :> xs) = f x :> smap f xs

{-# RULES "smap-fusion" ∀ f g xs.
      smap f (smap g xs) = smap (f . g) xs #-}
```

This rule will replace the left-hand-side `smap f (smap g xs)` with `smap (f . g) xs`, traversing the infinite stream only once and optimizing your program. But, can we formally prove this rule about Haskell programs?

Formal verification of Haskell programs can be mechanized in numerous tools, including Liquid Haskell [Vazou et al. 2014], Zeno [Sonnex et al. 2012] and HipSpec [Claessen et al. 2013]. Yet, most of these tools implement inductive verification, which is not trustworthy in the presence of infinite codata. Zeno, for example, only considers finite and total values, thus it can prove properties that do not hold in the presence of infinite structures (see §5.1). Even worse, Liquid Haskell can easily prove false in the presence of an infinite stream (see §1.3).

These inconsistencies in formal Haskell verifiers exist because coinduction is not simple. Unlike inductive reasoning that is well-understood, coinductive reasoning can be mechanized using various alternatives. For example, Leino and Moskal [2014] provide a program transformation approach that permits coinductive predicates and proofs to be checked by the inductive and SMT-automated program verifier Dafny. Abel [2010] uses sized types to explicitly reason about finite prefixes of potentially infinite values. Coq is one of the few formal verifiers with a long history of native support for coinduction [Chlipala 2013; Giménez 1996]. Yet, coinductive proof development in Coq is not easy: such proofs are not checked until they are completed, which is too late for Coq's interactive proof development.

In this work we present how coinductive proofs can be encoded and machine checked by an inductive formal verifier without native support for coinduction. The intuition is that, following the core ideas behind both Leino and Moskal [2014] and Abel [2010], by adding an extra index on coinductive predicates, the user can prove coinductive properties by induction on the index. We implement this idea in the Liquid Haskell inductive verifier using two approaches that respectively encode classical (à la Leino and Moskal [2014]) and constructive (à la Abel [2010]) logic proofs. Concretely:

- We start, in §1, by an overview of Liquid Haskell where we present the map fusion property as our "running example", we give an inductive proof for finite data and we discuss why infinite data lead to inconsistencies.

Haskell '22, September 15–16, 2022, Ljubljana, Slovenia
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9438-3/22/09.
https://doi.org/10.1145/3546189.3549922

- In §2, we present the *indexed coinduction* technique in which we index the coinductive predicates and encode coinductive proofs by induction on the index.
- In §3 we present the *constructive coinduction* technique that again uses indices to ensure guardedness in constructive proofs that are encoded in Liquid Haskell using refinements over GADTs.
- In §4 we evaluate the two approaches by presenting our case studies. We prove 10 properties about various unary and binary predicates on both streams and infinite lists. Based on these case studies, we conclude that the two techniques are equally verbose (each requires 281 lines of proof code to prove properties about 44 lines of executable Haskell code) and equally expressive (on the domain of computable predicates).
- Finally, in §5, we compare with related approaches.

The contribution of this work is twofold. First, we present how the user can machine check properties of Haskell code that manipulates infinite data using existing, inductive Haskell verifiers. Second, using our examples, we present how an inductive verifier could be extended to support coinductive reasoning. These extensions could, in the future, be applied both in Liquid Haskell and in GHC's dependent types.

Our code can be found in github.com/lykmast/co-liquid.

1 Liquid Haskell's Inductive Verification

We start with a short introduction on Liquid Haskell's inductive verification. We define a (for now inductive) stream data type (§1.0), which we use to perform "light" (§1.1) and "deep" (§1.2) verification and explain how these existing, inductive verification techniques break (i.e., are inconsistent) in the presence of coinductive data definitions (§1.3).

1.0 Inductive Data

Consider the data type `Stream a` whose elements are either empty streams or the products of prepending elements of type `a`, using the infix `(:>)` constructor:

```
data Stream a = a :> Stream a  | E
```

The standard stream definitions do not contain the empty case. In this section, we treat `Streams` as inductively defined, i.e., they have a base case which is marked in a box that will be removed in the definitions of the next sections, to restore the coinductive, standard stream definition.

Using the refinement types of Liquid Haskell, we define `NEStream`, the type alias of non empty streams.

```
type NEStream a = {s:Stream a | notEmpty s}

notEmpty :: Stream a → Bool
notEmpty E = False
notEmpty _ = True
```

That is, `NEStream` is the type of streams that are refined to satisfy the `notEmpty` predicate.

Note: To use the predicate `notEmpty` in the refinements, in the implementation we had to explicitly mark it as a Liquid Haskell `measure`, using special comment annotation. Here, for simplicity, we do not present such annotations; we only provide the unannotated Liquid Haskell signatures.

1.1 Inductive Light Verification

Liquid Haskell can be used to automate verification about "light" properties on inductive data. As a first example, we can prove that map preserves the stream's length:

```
smap :: (a → b) → x:Stream a
     → {s:Stream b |  slen s == slen x }
smap f E = E
smap f (x :> xs) = f x :> smap f xs

slen :: Stream a -> {i:Int | 0 <= i}
slen E = 0
slen (_ :> xs) = 1 + slen xs
```

Liquid Haskell will happily verify `smap`'s length preservation property. In fact `smap`'s definition serves as a proof that the property holds. Concretely, the refinement type checking judgements [Jhala and Vazou 2020] follow the inductive definition of `smap` to generate logical verification conditions, that directly correspond to proof by induction and are automatically discharged by the underlying STM solver.

Of course, this reasoning does not hold for non inductive data definitions: the length of streams that do not have a base case cannot be defined. That is why we put the `slen` and refinement definitions in box.

Other than functional properties, in the style of length preservation, Liquid Haskell automatically checks that all the defined functions are total: terminating and defined for all cases. For example, the definitions of stream head and tail accessors will only be valid assuming the non empty precondition.

```
shead :: NEStream a → a
shead (x :> _ ) = x

stail :: NEStream a → Stream a
stail (_ :> xs) = xs
```

With this `NEStream` precondition, Liquid Haskell has the obligation to ensure non emptiness each time `shead` or `stail` are used. For example, the `unsafe` function below, generates a refinement type error, since it calls `shead` on its (unconstrained) argument.

```
unsafe xs = shead xs -- Refinement Type Error
safe x xs = shead (smap (+1) (x :> xs))
```

The `safe` definition, on the other hand, is type safe, since Liquid Haskell uses the length preservation specification of `smap` to ensure that its result is not empty.

1.2 Inductive Deep Verification

Deep verification, in the setting of refinement types, is the process of providing explicit proofs to ensure properties that cannot be automatically proved by the SMT automation. Usually, such properties refer to the interaction of more than one function, thus, cannot be proved simply by the function definition.

We want to prove stream map fusion [Kiselyov et al. 2017], that is that the "smap-fusion" rule of §0 is correct. We prove this property using the theorem proving capabilities of Liquid Haskell [Vazou et al. 2018] that encode theorems as refinement type specifications and proofs as inhabitants to these types.

Concretely, the signature below encodes that for every functions f and g and every stream xs, smap f (smap g xs) equals[1] smap (f . g) xs.

```
mapFusion :: f:(b → c) → g:(a → b)
          → xs:Stream a
          → {smap f (smap g xs) = smap (f . g) xs}
```

We use the notation {p} to abbreviate the unit type refined with the predicate p, i.e., {v:()|p}. Thus, mapFusion only returns a unit value.

To prove mapFusion we construct an inhabitant, i.e., we provide a definition of mapFusion's body accepted by Liquid Haskell. The definition below follows the structure of smap: it has two cases and uses an inductive call in the (:>) case.

```
mapFusion f g E = ()
mapFusion f g (x :> xs)
 =   smap f (smap g (x :> xs))
 === smap f (g x :> smap g xs)
 === f (g x) :> smap f (smap g xs)
   ? mapFusion f g xs
 === (f . g) x :> smap (f . g) xs
 === smap (f . g) (x :> xs)
 *** QED
```

The first case is trivial: it is defined to be () and automated by Liquid Haskell's rewriting (concretely the PLE tactic [Vazou et al. 2017]). The inductive case, where the stream is x :> xs starts by the left-hand side and performs equational steps.

Since mapFusion is actually a Haskell function, equational reasoning is encoded by calling a set of Haskell operators that are refined to check equalities between each equational step. These operators are imported by the Liquid Haskell library ProofCombinators and summarized in Figure 0. The operator (===) receives two arguments, checks that they are equal, and returns the first, to accumulate equational-style, checked, proof steps. The operator (?) simply ignores the second argument, that in practice provides helper lemmas that justify equalities, while the *** QED is defined to complete the proof, by turning it into a unit.

```
(===) :: x:a → y:{a | x = y} → {v:a | v = x}
x === _ = x

x ? _ = x       data QED = QED     _ *** QED = ()
```

Figure 0. Proof Combinators of Liquid Haskell.

Back to the proof of mapFusion, we use (===) to expand the definition of smap twice. Next, we notice that the term smap f (smap g xs) is equal to smap (f . g) xs by the inductive hypothesis (here a call to mapFusion f g xs). The proof concludes by folding the definitions of (.) and smap to construct the right-hand side of the theorem.

In short, our proof is by induction! Of course, inductive proofs require a base case to be well formed, which is also required by Liquid Haskell: If the first line of the mapFusion definition (i.e., the proof's base case) is removed, Liquid Haskell will create a totality error that mapFusion is not defined for empty streams. But, when streams do not have the empty case (i.e., when they are coinductive) this error is not generated. Next, let's see what could go wrong if all the boxed code is removed.

1.3 What about Coinduction?

If from the previous example we remove all the boxed code, Liquid Haskell will happily accept our definitions and the proof of mapFusion. This behavior is very well aligned with the partial correctness principle [Flanagan 2006] of refinement types, which states that "if a program terminates, then it satisfies its specifications." Thus, by contraposition, false can be proved by any diverging program. Sadly, the runtime semantics is eager, thus infinite streams are seen as divergent and can easily prove false.

The simplest example that can prove false using infinite streams is falseStream, by recursing over the stream's tail:

```
falseStream :: Stream a → {false}
falseStream (_ :> xs) = falseStream xs
```

This example makes clear that Liquid Haskell's theorem proving capabilities cannot be used to check properties of coinductive data. For example, the definition below would constitute a valid inhabitant of mapFusion's specification.

```
mapFusion' xs = falseStream xs
```

By default, Liquid Haskell does not enforce the construction of valid, coinductive proofs. Variations of the above can be used to prove any property, shaking our confidence in Liquid Haskell itself.

To be fair, in order for Liquid Haskell to accept the definition of Stream, we had to use the "--no-adt" flag. This flag tells Liquid Haskell not to map Haskell data types to SMT

[1] In Liquid Haskell, operator = denotes SMT equality (syntactic equality with the three equality axioms). Haskell users acquainted with GHC RULES can view Liquid Haskell's equality as the same equality used in the RULES.

data types, which would reject non-well-founded types. Still, it would be desirable to prove properties such as `mapFusion`, despite `Stream`'s non-well-foundedness.

Next, we encode in Liquid Haskell two ways to construct proofs of properties on coinductive data that are consistent (i.e., cannot be used to prove false).

2 Indexed Coinduction

In this section we encode indexed coinduction, which lets us consistently prove properties about coinductive predicates. First (§2.0), we index coinductive properties with a natural number, to eliminate inconsistent proofs. Next (§2.1), we define indexed predicates that trivially satisfy base cases. Finally (§2.2), we conclude by noticing that indexed equality bisimulates stream equality.

2.0 Consistent Approach: Indexed Properties

A first attempt to ensure consistent proofs is to require inductive proofs. To do so, we define the type of indexed properties `IProp p`:

```
type-alias IProp p = k:Nat → { p } / [k]
```

This type says that to prove `IProp p` one needs to prove `p`, for all natural numbers `k`, using induction on `k`. The notation `[k]` is used by Liquid Haskell to encode termination metrics, i.e., expressions that provably decrease at each recursive function call, and thus prove termination of the function.

Note: Even though Liquid Haskell permits type aliases, it does not permit them being accompanied by termination metrics. In our implementation, `type-alias` annotations are manually inlined by the user.

Wrapped in `IProp`, the false predicate cannot be proved anymore, since in the base case, for `k=0`, there is not enough evidence to show `false`, as no recursive call is allowed.

```
falseStream :: Stream a → IProp false
falseStream _         0 = () -- ERROR
falseStream (_ :> xs) i = falseStream xs (i-1)
```

Yet, this is exactly the case for correct stream properties. Wrapped in `IProp` the `mapFusion` sketches as follows:

```
mapFusion :: f:(b → c) → g:(a → b) → s:Stream a
    → IProp (smap f (smap g s) = smap (f . g) s)
mapFusion _ _ _         0 = ()  -- ERROR
mapFusion f g (_ :> xs) i = ... -- OK
```

Even though Liquid Haskell can easily verify the inductive case, there is no way to prove the base case of the, now correct, theorem.

From this failing first attempt we conclude that the indexed technique can be used only to prove properties that trivially hold for the base case.

2.1 Precise Approach: Indexed Predicates

Our goal is to define coinductive predicates, indexed with a natural number `k`, that trivially hold when `k=0`. Having set this goal, we define `eqK` to be indexed stream equality.

```
eqK :: Eq a ⇒ Stream a → Stream a → Int → Bool
eqK _ _ 0             = True
eqK (x:>xs) (y:>ys) k = x == y && eqK xs ys (k-1)
```

Concretely, `eqK xs ys k` checks if the first `k` elements of the streams `xs` and `ys` are equal. Indexed equality on `k=0` is trivially true, since the zero first elements of the stream are always equal. So, indexed equality can be proved via indexed coinduction.

Next, we encode and prove map-fusion as a coinductive indexed proposition.

Indexed Coinductive Propositions. We encode coinductive propositions using the type alias `CProp p`, that is similar to `IProp` except the index `k` is now further applied to the indexed property `p`. (In §4 we discuss how indexed properties can be derived in general.)

```
type-alias CProp p = k:Nat → {p k} / [k]
```

Using `CProp`, we define the map-fusion property as the specification of `mapFusionIdx` that equates all the elements of the streams `smap f (smap g xs)` and `smap (f . g) xs`.

```
mapFusionIdx :: f:(b → c) → g:(a → b)
             → s:Stream a →
  CProp {eqK (smap f (smap g s)) (smap (f . g) s)}
```

The proof can only go by induction on the index `k`, as indicated by the termination metric `/ [k]`. The base case is easy and goes by unfolding the definition of `eqK` which is always true at the index `0`.

```
mapFusionIdx f g xs 0
  = eqK (smap f (smap g xs)) (smap (f . g) xs) 0
  *** QED
```

The inductive case also starts easily. Concretely, it starts by exactly following the equational reasoning steps of the theorem proved in §1.2:

```
mapFusionIdx f g (x :> xs) k
 =   smap f (smap g (x :> xs))
 === smap f (g x :> smap g xs)
 === f (g x) :> smap f (smap g xs)
   ? mapFusionIdx f g xs (k-1)
 === (f . g) x :> smap (f . g) xs -- ERROR
 === smap (f . g) (x :> xs)
 *** QED
```

However, we are stuck again: Liquid Haskell is not convinced that the inductive call `mapFusionIdx f g xs (k-1)` can prove `smap f (smap g xs) = smap (f . g) xs`. And it has every right not to be convinced, since the inductive call provides evidence for the indexed equality `eqK`, not `(=)`.

To proceed with the proof, we need to define a new, coinductive proof operator, similar to the `(===)` of Figure 0, that will let us: (1) *check* that the proof step is correct and (2) *conclude* that our final proof is correct. We define the proof combinator `(=#=)`, which has a precondition that checks and a postcondition that concludes indexed equalities:

```
(=#=) :: Eq a
  ⇒ x:Stream a
  → k:{Nat | 0 < k }
  → y:{Stream a | eqK (stail x) (stail y) (k-1)
                && shead x == shead y}
  → {v:Stream a | eqK x y k && v == x}
```

That is, `(=#=) x k y` checks that `x` and `y` have equal heads and indexed equal tails to conclude that they are indexed equal. Its definition is not assumed, but proved just by expanding the definition of indexed equality. Note, that the operator returns its first argument, giving us the ability to chain indexed equality proof steps. Also, note that the order of the arguments is strange: the index `k` appears between the two stream arguments. We chose this order on purpose; we further define a function application operator `(#)`, similar to `($)` but with the proper precedence, that lets us write `x =#= k # y` instead of `(=#=) x k y`.

```
f # x = f x
```

Let us now conclude the proof of `mapFusionIdx`:

```
mapFusionIdx f g (x :> xs) k
 =   smap f (smap g (x :> xs))
 === smap f (g x :> smap g xs)
 === f (g x) :> smap f (smap g xs)
   ? mapFusionIdx f g xs (k-1)
 =#= k #
     (f . g) x :> smap (f . g) xs
 === smap (f . g) (x :> xs)
 *** QED
```

This proof is now not only accepted, but it is consistent (as proof by induction on `Nat`) and, most importantly, it looks a lot like the inductive proof.

2.2 Take Lemma: Did we Prove Equality?

Even though our proof looks much like the original inductive proof, the theorem's statement has diverged. Instead of proving equality between streams, in §2.1 we prove indexed equality. Here, we explain how these two forms of the theorem's statement connect.

Bird and Wadler [1988] formulate and prove the *take lemma*, which states that two streams are equal *if and only if* their first k "taken" elements are equal, forall k. Namely:

$$x = y \iff \forall k.\, \mathsf{take}\ k\ x\ = \mathsf{take}\ k\ y$$

We axiomatize the right-to-left direction of this lemma in Liquid Haskell as follows:

```
assume takeLemma :: x:Stream a → y:Stream a
       → (k:Nat → {take k x = take k y})
       → {x = y}
```

In our mechanization, streams do not have a base case, thus `take` converts streams to Haskell's lists, returning an empty list on zero:

```
take :: Nat → Stream a → [a]
take 0 _         = []
take i (x :> xs) = x : take (i-1) xs
```

By induction on k, we can prove that our indexed equality predicate behaves like the `take` equality:

```
eqKLemma :: x:Stream a → y:Stream a → k:Nat
         → {eqK x y k ⇔ take k x = take k y}
```

We combine the two lemmas above to derive stream equality from our indexed equality:

```
approx :: x:Stream a → y:Stream a
       → CProp {eqK x y} → {x = y}
approx x y p =
  takeLemma x y (\k → p k ? eqKLemma x y k)
```

The proof calls the `takeLemma` with an argument that combines the `eqK x y k` premise and `eqKLemma`, for each `k`.

By calling `approx` we are able to replace indexed with stream equality in our map fusion theorem:

```
mapFusion :: f:(b → c) → g:(a → b)
          → s:Stream a →
          → {smap f (smap g s) == smap (f . g) s}
mapFusion f g s
  = approx (smap f (smap g s))
           (smap (f . g) s) (mapFusionIdx f g s)
```

In short, we mechanized indexed coinduction by (1) defining a related property indexed by a natural number k and (2) proving the related property, by induction on k. The benefit of this technique is that the proof is simple and can use inductive techniques, in the style of equational reasoning. The great drawback though is that for consistency, the developer needs to make sure that induction happens on the index and not on a substream, as sketched below.

```
thm (x <: xs) i
  = ... thm _ (i-1)  -- good inductive hypothesis
  = ... thm xs _     -- potentially inconsistent!
```

In all our examples, we used Liquid Haskell's termination metrics to ensure inductive calls occur on smaller indices, yet, in more advanced proofs this requirement could be missed. Next, we present an alternative mechanization of coinductive proofs that does not have user-imposed requirements.

3 Constructive Coinduction

Constructive coinduction is our second mechanization technique, where proofs are constructed using Haskell's (refined) GADTs [Peyton Jones et al. 2006; Xi et al. 2003]. First (§3.0) we define `EqC`, the GADT that constructs observational equality on streams. Next (§3.1), we use `EqC` to prove our running theorem. Finally (§3.2), via the take lemma, we prove that `EqC` approximates stream equality.

3.0 Constructive Equality

As a first (failing) attempt to define constructive stream equality, we define Coq's textbook [Chlipala 2013] coinductive stream equality, using Liquid Haskell's data propositions [Borkowski et al. 2022] and a refined GADT:

```
data EqC1 a where
  EqRefl1 :: x:a → xs:Stream a → ys:Stream a
          → Prop (EqC1 xs ys)
          → Prop (EqC1 (x :> xs) (x :> ys))
```

The `EqC1` data type has one constructor, that given a head `x`, two steams, `xs` and `ys`, and a proof of the proposition that `xs` is equal to `ys`, constructs a proof of the proposition that `x :> xs` is equal to `x :> ys`.

Liquid Haskell's built-in `Prop` type constructor encodes propositions; given an expression `e`, it denotes a proposition that `e` holds. It is defined as follows:

```
type Prop e = {v:a | e = prop v}
measure prop :: a → b
```

where `prop` is an *uninterpreted function* in the logic. So, any expression of type `Prop e` is a witness that proves `e`.

The `EqC1` data constructor, that is used as an argument to `Prop`, is defined below:

```
data Proposition a = EqC1 (Stream a) (Stream a)
```

The statement `w : Prop (EqC1 xs ys)` states that `w` witnesses that the proposition `EqC1 xs ys` holds. Since the only way to construct such a term is via the `EqRefl1` construction, `w : Prop (EqC1 xs ys)` witnesses observational equality of `xs` and `ys`.

The problem: no guardedness condition. Even though `EqC1` seemingly encodes observational equality, due to the lack of a base case, as in §1.3, we can trivially prove false.

```
falseProp :: xs:Stream a → ys:Stream a
          → Prop (EqC1 xs ys) → {false}
falseProp _ _ (EqRefl1 a xs ys p)
  = falseProp xs ys p
```

Remember, that the definition of `EqC1` follows Coq's textbook stream equality definition. But in Coq, this equality is defined as `CoInductive`, which comes with the *guardedness condition* check. This check ensures that recursive calls *produce* values, i.e., dually to recursive calls of inductive data, recursive calls on codata should be guarded by data constructors. Such a condition is not enforced by (Liquid) Haskell and is violated by the `falseProp` definition. Thus, our first attempt to define constructive stream equality is not consistent.

Indices to the rescue. Next, we encode the guardedness condition using indices, following Agda's sized types approach [Abel 2010]. The indexed constructive stream equality is defined as follows:

```
data EqC a where
  EqRefl :: i:Nat → x:a
         → xs:Stream a → ys:Stream a
         → (j:{Nat | j < i} → Prop (EqC j xs ys))
         → Prop (EqC i (x :> xs) (x :> ys))

data Proposition a = EqC Int (Stream a) (Stream a)
```

That is, to construct an equality for the index `i` one can use the equality on tails for some index `j` strictly smaller than `i`. With this guard, the previous `falseProp` cannot be encoded:

```
falseProp :: i:Nat  → xs:Stream a → ys:Stream a
          → Prop (EqC i xs ys) → {false}
falseProp 0 _ _ _ = () -- REFINEMENT TYPE ERROR
falseProp i _ _ (EqRefl _ x xs ys p)
  = falseProp (i-1) xs ys (p (i-1))
```

The recursive call is easy: `p` of type `j:{Nat | j < i} → Prop (EqC j xs ys)` can be called with `i-1`. That call, combined with the requirement that `j` is a `Nat` requires that `i` is greater than `0`. Thus we are left with the `i=0` base case, from which it is impossible to prove false. Unsurprisingly, this reasoning is similar to §2.0. Indexing permits coinductive reasoning using inductive verification.

3.1 Proof by Constructive Coinduction

Next, we use constructive coinduction to prove the map fusion theorem.

```
mapFusionC :: f:(b → c) → g:(a → b)
           → s:Stream a → i:Nat
           → Prop (EqC i (smap f (smap g s))
                         (smap (f . g) s))
mapFusionC f g (x :> xs) i =
  EqRefl i ((f . g) x) (smap f (smap g xs))
           (smap (f . g) xs) (mapFusionC f g xs)
  ? lhs ? rhs
 where
  lhs =   ((f . g) x) :> (smap f (smap g xs))
      === (f (g x)) :> (smap f (smap g xs))
      === smap f (g x :> smap g xs)
      === smap f (smap g (x :> xs))
      *** QED
  rhs =   ((f . g) x) :> (smap (f . g) xs)
      === smap (f. g) (x :> xs)
      *** QED
```

The only way to construct a term of the required type is by the data constructor `EqRefl`. Calling this with the inductive hypothesis in the definition of `MapFusionC` above gives us a witness that `EqC i ((f . g) x :> smap f (smap g xs)) ((f . g) x :> smap (f . g) xs)`. In both sides, we need to push the head `(f . g) x` inside the `smap` and persuade Liquid Haskell that this push proves the theorem. This is exactly what `? lhs` and `? rhs` serve for: they provide the missing steps using equational reasoning. With this, the proof completes without any unguarded recursive calls!

3.2 Again, Did we Prove Equality?

Finally, as in §2.2, we use the take lemma to show that constructive equality approximates stream equality and use this approximation in our map fusion theorem.

Concretely, we start by proving that for each index `i`, constructive equality between the streams `x` and `y` implies that the `i` prefixes of the streams are equal.

```
eqCLemma :: x:Stream a → y:Stream a
         → i:Nat → (Prop (EqC i x y))
         → {take i x = take i y}
eqCLemma _ _ 0 _ = ()
eqCLemma _ _ i (EqRefl _ _ xs ys p)
  = eqCLemma xs ys (i-1) (p (i-1))
```

The proof goes by induction on `i`: the base case is automatically proved by Liquid Haskell's PLE and the inductive case is easy, calling the tail equality `p` for the previous index.

Note that the proof of `eqCLemma` requires inverting the constructive `EqC` proof. In theory, to prove the lemma given the `EqC i x y` witness, we need to know that this equality was only derived by the tail equality and not via any other way. That is, if the `EqC` data type had other constructors, the proof would have to pattern match on all of them. In practice, this proof and the requirement of inversion are the reasons why the definition of `EqC` had to be a GADT, instead of a function assumption.

By combining the `eqCLemma` above with the `takeLemma` of §2.2, we prove that constructive equality approximates stream equality:

```
approx :: x:Stream a → y:Stream a
       → (i:Nat → Prop (EqC i x y)) → {x = y}
approx x y p
  = takeLemma x y (\i → eqKLemma x y i (p i))
```

Finally, this approximation theorem can be used to convert constructive to stream equality in our map fusion theorem.

```
mapFusion :: f:(b → c) → g:(a → b)
          → xs:Stream a
          → {smap f (smap g xs) = smap (f . g) xs}
mapFusion f g xs =
  approx (smap f (smap g xs)) (smap (f . g) xs)
         (mapFusionC f g xs)
```

In short, we mechanized constructive coinduction by (1) encoding the coinductive predicate as an indexed data proposition and (2) proving a coinductive property by constructing a witness for the coinductive predicate. Consistency of the constructive proofs relies on the guardedness check, that we implemented using indices. One way to add native support for coinductive reasoning in Liquid Haskell would be to extend it with guardedness checks, like Coq.

4 Evaluation

We used both the indexed (§2) and the constructive (§3) techniques to prove 10 properties on infinite streams and lists that involve equality as well as more complicated coinductive predicates, for example, lexicographic ordering. Here, we present the properties we proved (§4.0) and use them to compare the two techniques (§4.1).

4.0 Case Studies

Table 0 summarizes our 10 case studies. Most of our examples are taken from the literature [Leino and Moskal 2014; Roşu and Lucanu 2009] and cover a wide variety of properties.

4.0.0 Equal Streams.

The first 4 properties prove equality on streams. Property 0 was detailed in §2 and §3. Using exactly the same predicates (`eqK` and `EqC`) and axiom (`takeLemma`), we prove three more properties:

Property 1: Merge even and odd elements. One very popular example of a coinductive proof concerns the following functions on streams:

```
merge :: Stream a → Stream a → Stream a
merge (x :> xs) ys = x :> merge ys xs

evens, odds :: Stream a → Stream a
odds  (x :> xs) = x :> odds (stail xs)
evens xs        = odds (stail xs)
```

It is easy to see that, for any stream, merging its odd and even elements will reconstruct the initial stream. This is expressed in Liquid Haskell as follows:

```
mergeEvenOdd :: xs:Stream a
             → {merge (odds xs) (evens xs) = xs}
```

Properties 2-3: Thue-Morse sequence. These two properties are inspired by Roşu and Lucanu [2009] and deal with morse signals, represented as infinite streams of Booleans. We included them because they are somewhat more complex proofs since we have to invoke the coinductive hypothesis at a deeper level, after unfolding the streams twice. The definition of the properties is shown in Figure 1. First, we define the stream `morse` that encodes the Thue-Morse sequence, i.e., an infinite sequence obtained by starting with `False` and successively appending the Boolean complement of the sequence obtained thus far. Then, we define the function `f` that takes as input a stream and replaces each of its values

Table 0. Quantitative Summary of Coinductive Proofs.

Note: **Predicate** is the predicate used to express the proved **Property**. **Exec.** is the lines of executable Haskell code, i.e., functions that return neither unit nor propositions and are shared by the two techniques. **Proof** is the lines of Haskell function defined to inhabit proofs. **Annot.** is the Liquid Haskell annotations.

				Indexed		**Constructive**	
	Property	Predicate	Exec.	Proof	Annot.	Proof	Annot.
Streams	0. `mapFusion`	`equal`	2	21	14	21	14
	1. `mergeEvenOdd`	`equal`	6	19	14	17	14
	2. `morseFix`	`equal`	8	44	15	34	14
	3. `fNotCommute`	`equal`	6	41	14	28	13
	4. `trivialAll`	`trivial`	2	13	4	12	10
	5. `mergeSelfDup`	`dup`	3	17	5	16	12
	6. `squareNNeg`	`nneg`	4	13	4	10	11
	7. `belowSquare`	`below`	5	29	13	22	14
Lists	8. `mapInfinite`	`infinite`	5	15	7	18	11
	9. `mapFusion`	`equal`	3	26	7	25	13
	`Total`		44	214	67	182	99

```
morse :: Stream Bool
morse = False :> True
  :> merge (stail morse) (smap not (stail morse))

f :: Stream Bool → Stream Bool
f xs = shead xs :> not (shead xs) :> f (stail xs)

not True  = False
not False = True

-- Morse Property
morseFix :: {f morse = morse}

-- f Property
fNotCommute :: s:Stream Bool
            → {f (smap not s) = smap not (f s)}
```

Figure 1. Properties 2 and 3 on Morse signals.

`x` with `x`, followed by `x`'s negation. Property 2, `morseFix`, proves that `f` is the fixpoint of the `morse` sequence. Property 3, `fNotCommute`, proves that `f` and `(smap not)` commute.

4.0.1 Unary Predicates on Streams.

While equality is the most frequently used predicate, we used our techniques to prove other copredicates. The next three properties reason about unary predicates on streams.

Property 4: Trivial streams. The most trivial coinductive unary predicate on streams, is the one that traverses the infinite stream and "returns" some Boolean.

```
trivial :: Stream a → Bool
trivial (x :> xs) = trivial xs

trivialAll :: s:Stream a → {trivial s}
```

The property we proved is `trivialAll` and states that all streams satisfy trivial.

Following the equality proofs, for each new predicate we introduce we need to define an indexed version, a constructive version, and an axiom that connects the indexed with the original predicate.

The indexed predicate is defined as below:

```
trivialK :: Stream a → Nat → Bool
trivialK _         0 = True
trivialK (x :> xs) k = trivialK xs (k-1)

trivialAllK :: s:_ → k:Nat → {trivialK s k}
```

Importantly, for `k=0` the predicate should be true, while for bigger `k`s it simply recurses. We proved, by induction on `k`, that `trivialK` holds for all indices and streams.

For the constructive approach, we defined the `Trivial` proposition as follows:

```
data Trivial a where
  TRefl :: i:Nat → x:a → xs:Stream a
        → (j:{Nat | j < i} → Prop (Trivial j xs))
        → Prop (Trivial i (x :> xs))

trivialAllC :: s:_ → i:Nat → Prop (Trivial i s)
```

The `Trivial` GADT has one constructor that, like `EqC` in §3, for each natural number `i` and stream `x :> xs`, returns a property that `x :> xs` is trivial on `i`, given a property that `xs` is trivial for all `j` smaller than `i`. Using the constructive technique, we proved in `trivialAllC` that each stream `s` has the trivial property.

To prove `trivialAll` from either `trivialK` or `trivialC`, we used an axiom that similar to the take lemma, connects the indexed with the original predicates:

```
assume trivialLemma :: s:Stream a
                    → (k:Nat → {trivialK s k})
                    → {trivial s}
```

Using `trivialLemma`, we reached the `trivialAll` proof twice.

Property 5: Duplicate streams. The second unary predicate we defined is `dup` that checks that each stream element has an equal element next to it. This property was added because it observes more than one elements of the stream in each unfolding.

```
dup (x₁ :> x₂ :> xs) = x₁ == x₂ && dup xs
mergeSelfDup :: xs:_ → {dup (merge xs xs)}
```

We proved, using definitions similar to the trivial predicate, that merging a stream with itself always satisfies the `dup` predicate.

Property 6: Non negative streams. Our final unary stream predicate is `nneg` and checks that a stream of integers consists only of non negative numbers:

```
nneg :: Stream Int → Bool
nneg (x :> xs) = 0 <= x && nneg xs
```

The property we proved states that the "square" of a stream, i.e., the result of pointwise multiplication of the stream with itself, is a non negative stream.

```
mult :: Stream Int → Stream Int → Stream Int
mult (a :> as) (b :> bs) = a * b :> mult as bs

squareNNeg :: s:_ → {nneg (mult s s)}
```

This property shows that our techniques can be used to reason about streams of non polymorphic values, here integers.

4.0.2 Binary Predicates: Lexicographic Ordering.

In order to challenge the expressiveness of our techniques, we used them to check lexicographic comparison for streams. The original predicate `below x y` is true only when `x` is lexicographically below `y`:

```
below :: Ord a ⇒ Stream a → Stream a → Bool
below (x :> xs) (y :> ys) =
  x <= y && (x == y `implies` below xs ys)
  where implies x y = not x || y
```

The indexed version of `below` is quite straightforward, it simply guards the recursive call:

```
belowK :: Ord a ⇒ Stream a → Stream a → Nat → Bool
belowK k (x :> xs) (y :> ys) =
  x <= y && (x == y `implies` belowK (k-1) xs ys)
```

The constructive version of `below` is more interesting. In order to avoid reasoning about constructive Booleans (since `below` is using conjunction and implication) we interpreted `below` using two different cases:

```
data BelowC a where
  Bel0 :: Ord a
    ⇒ i:Nat → x:a → xs:Stream a → ys:Stream a
    → ({j:Nat | j < i} → Prop (BelowC j xs ys))
    → Prop (BelowC i (x :> xs) (x :> ys))
  Bel1 :: Ord a
    ⇒ i:Nat → x:a → {y:a| x < y}
    → xs:Stream a → ys:Stream a
    → Prop (BelowC i (x :> xs) (y :> ys))
```

The first case `Bel0` compares streams of same heads and requires that the tail of the first is below the tail of the second. The second case `Bel1` decides `below`, simply by looking at the heads. We can show that the constructive and original predicates indeed encode the same predicate.

Property 7: Below square. We used the two encodings of `below` to prove our final property on streams: each stream is always below its "square":

```
belowSquare :: s:Stream Int → {below s (mult s s)}
```

4.0.3 Coinduction on Lists.

Haskell's lists are also often treated as codata (e.g., Prelude's notable `repeat` returns an infinite list). We used our two approaches to prove two coinductive properties on lists.

Because Liquid Haskell comes with various inductive predicates on built-in Haskell's lists, we did not use Haskell's lists but defined our own data type:

```
data L a = a :| L a | Nil
```

We defined two coinductive predicates on this list, a unary which ensures infinity and a binary which checks equality.

Property 8: Map infinite lists. The check of infinity is the most interesting property on lists, coming from streams, since it relies on returning `False` in the base case:

```
infinite :: L a → Bool
infinite (_ :| xs) = infinite xs
infinite Nil       = False
```

We used the `infinite` predicate to ensure than `map` preserves infinity:

```
mapInfinite :: f:(a → b) → xs:{L a | infinite xs}
            → {infinite (map f xs)}

map :: (a → b) → L a → L b
map _ Nil = Nil
map f (x :| xs) = f x :| map f xs
```

The proving techniques remain the same on lists: we defined the indexed and constructive predicates and an axiom that reconstructs the original predicate.

The indexed `infinite` predicate is defined as follows:

```
infiniteK :: L a → Nat → Bool
infiniteK _         0 = True
infiniteK Nil       _ = False
infiniteK (_ :| xs) k = infiniteK xs (k-1)
```

As with streams, the k=0 case should be True. Note that with lists, unary predicates have one more case, for Nil. Because of this, our proofs, that usually follow the structure of the predicates, also have one extra case, which is usually trivial.

The constructive predicate has only one case:

```
data InfiniteC a where
  Inf :: i:Nat → x:a → xs:L a
      → (j:{Nat | j < i} → Prop (InfiniteC j xs))
      → Prop (InfiniteC i (x :| xs))
```

The list x :| xs is infinite when xs is also infinite, while there is no constructor to ensure an empty list is infinite. Of course, this is a consequence of the meaning of the predicate, while for most predicates (e.g., dup or nneg) the constructive property requires more than one constructors.

In both techniques, the list proofs are similar to the ones on streams. To reconstruct the original from the indexed predicate, similar to streams, we assume the lemma below:

```
infLemma :: xs:L a → (k:Nat → {infiniteK xs k})
                   → {infinite xs}
```

Property 9: List map fusion. Our last property proves map fusion on infinite lists:

```
mapFusion :: f:(b → c) → g:(a → b) → xs:L a
          → {map f (map g xs) = map (f . g) xs}
```

The indexed predicate for list equality has now four cases:

```
eqK :: Eq a ⇒ L a → L a →  k: Nat → Bool
eqK _        _        0 = True
eqK Nil      Nil      k = True
eqK (a:|as) (b:|bs) k = a == b && eqK as bs (k-1)
eqK _        _        _ = False
```

The first three cases are expected, while the last returns false when comparing an empty to a non empty list.

As with the infinite predicate, the false cases simply do not appear in the constructive predicate, which for equality has two constructors: one that equates empty lists and the coinductive that compares two non empty lists.

```
data EqC a where
  EqNil :: i:Nat
        → Prop (EqC i Nil Nil)
  EqCos :: i:Nat → x:a → xs:L a → ys:L a
        → (j:{Nat | j < i} → Prop (EqC j xs ys))
        → Prop (EqC i (x :| xs) (x :| ys))
```

The proofs are unsurprising, while, as in stream equality, we used the take lemma to retrieve SMT equalities.

Note on more complex data types. Even though we only evaluated our techniques on streams and lists, we are confident that they apply to more complex data types. Essentially the requirement to apply our techniques to some codata is the ability to assume the "take lemma". Hutton and Gibbons [2001] explain how the approximation lemma, which is a simplification of the take lemma, can be generalized to any data type μF, where F is a locally continuous functor, ensuring that the generalized approximation lemma, and thus our techniques, do apply to e.g., infinite tree-like data types.

4.1 Comparison of the Two Techniques

Based on our case studies and experience, we compare the two techniques (indexed and constructive) on three axes: (1) code size, (2) expressiveness, and (3) cognitive effort.

Code size. Table 0 presents the lines of code required for our proofs. The **Exec** column contains the executable Haskell functions (e.g., merge, smap) as well as the original versions of the predicates (e.g., below). The **Indexed** and **Constructive** columns can be used to compare the code required for each approach, where **Annot.** refers to the Liquid Haskell specific annotations while **Proof** is the Haskell proof terms that inhabit them. The sum of annotations and proofs is 281 lines of code for both approaches. The fact that this number is exactly the same in both approaches is a coincidence; it was however expected that code size would be similar, since each approach has a different (but similar in size) overhead. In the constructive approach the definition of the GADT takes many lines, especially because they are defined twice: the Liquid Haskell refined definitions also require unrefined Haskell GADT. On the other hand, the indexed approach has the overhead of encoding proof combinators (e.g., operator (=#=) of §2.1) for some of the predicates. The size of the proofs of the properties is very similar in both approaches.

Expressiveness. Our examples only involve "computable" predicates, i.e., predicates that can be expressed as Haskell Boolean functions. On this domain, we observe that the expressiveness of the two approaches is the same, since we did not run into a coinductive predicate that can be encoded using one technique but not the other. The indexed approach lets you conduct the proofs by folding and unfolding the Haskell indexed predicate, while the constructive approach goes by case splitting and applying the data constructor of the GADT. The expressiveness advantage of the constructive approach will show on reasoning about non computable predicates, in the style of Kleene closures [Winskel 1993], but we leave such predicates as future work.

Cognitive effort. The constructive approach is more expressive, yet our educated claim is that it requires more cognitive effort. Data propositions is a novel Liquid Haskell feature that encodes Coq-style inductive predicates using GADTs. We conjecture that Haskell programmers are not very familiar with this style of constructive programming. Yet, once the user hits the maximum of the constructive learning curve, our constructive technique is cleaner: in most situations the constructors are the only place we have indices. The term expansion and the coinductive hypothesis are usually index-free!

5 Related Work

Here we present the three mechanized verifiers that influenced our work (§5.0) and summarize how existing verifiers for Haskell programs treat coinduction (§5.1). We refer the reader to Jacobs and Rutten [1997] for a foundational tutorial on coinduction and to Gibbons and Hutton [2005] for (paper and pencil) proofs on Haskell corecursive programs.

5.0 Mechanized Coinduction

Coq has, for some time now, support for coinduction [Bertot 2006]. The proving technique in §3 is partly inspired from the Coq's textbook [Chlipala 2013] bisimilarity relation for infinite streams, where in place of syntactic guardedness we use natural numbers to keep track of productivity. Both Coq's and our guardedness conditions are similarly strict and require data to produce, thus rejecting functions and properties that are not proved to be always productive, such as `filter` properties [Rusu and Nowak 2022]. The disadvantage of Coq's coinductive mechanization, compared to our technique, is that the proof is checked after `QED`, which means that the user interaction is lost. In our Liquid Haskell encoding, we have no user interaction, but we do have localized errors. The approach of §2 preserves local errors (and thus better user experience), while §3, as in Coq, has no proof steps and only returns a general failing error.

Mini Agda's coinduction [Abel 2010] is quite similar to Coq's in the encoding of bisimilarity. A key difference is that Mini Agda uses sizes to encode guardedness — a feature that we leverage in §3 in order to encode bisimilarity in Liquid Haskell. In actual proofs, this difference is not significant since the invocation of the coinductive hypothesis is immediate. However, sizes prove useful when dealing with more complex definitions, e.g., various coinductive functions. More on the expressiveness of sizes as a measure of productivity can be found in [Abel 2010; Abel and Pientka 2016].

Dafny's approach of coinduction [Leino and Moskal 2014] greatly inspired our indexed approach (§2). Coinductive predicates are syntactically checked to ensure monotonicity, which is important for proving soundness. Indexed proofs are formed by proving the indexed version of the predicate for all indexes. Finally, coinductive proofs are obtained by using the correspondent axiom, like we do in §4.0.1 with `trivialLemma`. Of course, Dafny provides an automated program transformation that introduces indices, while in our case the transformation is manually performed by the user.

In [Leino and Moskal 2014] we can also find a proof of soundness, which connects indexed proofs and predicates to coinductive ones. It uses the Kleene fix-point theorem [Winskel 1993], after proving Scott-continuity for predicates. An important takeaway is "positivity", which is a restriction on the form of predicates that can be approximated using the indexed method.

5.1 Haskell Verifiers

Many Haskell verifiers target only total Haskell programs, which permits using well known and automated inductive verification techniques but allows them to prove properties that do not hold in the presence of infinite data. Consider for example, the standard Haskell encoding of natural numbers: `data Nat = Z | S Nat`. Zeno [Sonnex et al. 2012] assumes all values are total and, in Theorem 10 of its test suite, automatically proves that `∀ m:Nat. m - m = Z`, which does not hold when `m` is infinite, because the left-hand-side will not terminate. Liquid Haskell can also prove the same property and also can prove false (§1.3) in the presence of infinite data. The soundness of inductive reasoning is preserved by rejecting non-wellfounded data definitions. With the wellfoundedness check active, users can employ the well understood principle of induction to reason about their programs, but are not able to define coinductive types and reason about their properties as we did here.

HERMIT [Farmer et al. 2012] and HALO [Vytiniotis et al. 2013] are two Haskell verifiers that do reason about infinite data. HERMIT performs equational reasoning by rewriting the GHC core language, guided by user specified scripts. This approach is far from ours where the proofs are Haskell programs while SMT solvers are used to automate reasoning. HALO is a prototype contract checker that translates Haskell programs to first-order SMT logic, using denotational semantics, and validates them against user-provided contracts. HALO reasons about laziness and infinite data and explicitly encodes Haskell's bottom in SMT logic. Unfortunately, this encoding renders HALO's SMT queries outside of decidable logics which makes verification using HALO unpredictable. On the contrary, Liquid Haskell prioritizes SMT-predictable verification, so it shamefully disregards bottoms, which, currently, makes coinductive reasoning possible only with explicit user encodings, like the ones we presented.

Hs-to-coq [Spector-Zabusky et al. 2018] converts Haskell code to Coq, which users can verify for functional correctness. Hs-to-coq has been used to verify real Haskell code (e.g., the containers library) and permits coinductive reasoning. Concretely, the user can annotate data types as coinductive and functions as corecursive and then use Coq's `CoInductive` principle to prove coinductive properties. Thus, the properties of § 4 can be verified, in Coq, via hs-to-coq.

Dependent Types for Haskell is a work initiated by Eisenberg [2016] and is currently under active design in GHC (see ghc-proposal#378). Interestingly, the dependent Haskell proposal promises neither a termination nor a guardedness check. We conjecture that in the presence of codata, the lack of a guardedness check could lead to inconsistencies, similar to §1.3, and we believe that the lessons presented in this work can be used by the GHC's dependent types proposal.

6 Conclusion

We used the Liquid Haskell inductive verifier to prove 10 properties on infinite data by coinduction. We encoded coinduction in the inductive verifier using two approaches. In the indexed approach, the predicate is indexed by a natural number k and the proof is by induction on k. In the constructive approach, the predicate is encoded as a refined GADT which is guarded using indexing. Using either of these approaches, a Haskell programmer can machine check coinductive properties of their Haskell code in Liquid Haskell.

As an important contribution, with this experiment we concretely identify two alternative extensions required for Liquid Haskell (or even GHC's dependent types) to natively support coinductive reasoning: indexed predicate transformation (in the classical logic setting; like in Dafny), or implementation of a guardedness check (in the constructive setting; like in Coq).

In the future, we can design and implement automation to realize the two proposed encodings, currently manually provided by the user. We see two potential directions for such automation. First, we could follow Dafny's approach [Leino and Moskal 2014] to mechanically transform copredicates and cofunctions by inserting an index that will, also mechanically, be used to ensure the guardedness and positivity requirements. A second direction would be to use SMT's (concretely CVC4's [Reynolds and Blanchette 2017]) support for codata to reason about coinductive properties using SMT's decision procedures.

Acknowledgments

This work is partially founded by the Horizon Europe ERC Starting Grant CRETE (GA: 101039196), the US Office of Naval Research HACKCRYPT (Ref. N00014-19-1-2292), the Atracción de Talento grant (Ref. 2019-T2/TIC-13455), and the Juan de la Cierva grant (IJC2019-041599-I).

References

Andreas Abel. 2010. MiniAgda: Integrating Sized and Dependent Types. *Partiality and Recursion in Interactive Theorem Provers.* https://doi.org/10.4204/EPTCS.43.2

Andreas Abel and Brigitte Pientka. 2016. Well-founded Recursion with Copatterns and Sized Types. *Journal of Functional Programming.* https://doi.org/10.1017/S0956796816000022

Yves Bertot. 2006. CoInduction in Coq. *CoRR.* https://arxiv.org/abs/cs/0603119

Richard Bird and Philip Wadler. 1988. *Introduction to Functional Programming.* Prentice Hall International. https://www.research.ed.ac.uk/en/publications/an-introduction-to-functional-programming

Michael Borkowski, Niki Vazou, and Ranjit Jhala. 2022. Mechanizing Refinement Types. In *CoRR.* https://arxiv.org/abs/2207.05617

Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press. http://adam.chlipala.net/cpdt/

Koen Claessen, Moa Johansson, Dan Rosen, and Nick Smallbone. 2013. HipSpec: Automating Inductive Proofs of Program Properties. *ATx/WInG.* https://doi.org/10.29007/3qwr

Richard A. Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice.* Ph. D. Dissertation. University of Pennsylvania. https://arxiv.org/abs/1610.07978

Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe. 2012. The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. *Haskell.* https://doi.org/10.1145/2364506.2364508

Cormac Flanagan. 2006. Hybrid Type Checking. *POPL.* https://doi.org/10.1145/1111320.1111059

Jeremy Gibbons and Graham Hutton. 2005. Proof Methods for Corecursive Programs. *Fundamenta Informaticae.* https://doi.org/10.5555/1227189.1227192

Eduardo Giménez. 1996. An Application of Co-Inductive Types in Coq: Verification of the Alternating Bit Protocol. *Types for Proofs and Programs.* https://doi.org/10.1007/3-540-61780-9_67

Graham Hutton and Jeremy Gibbons. 2001. The Generic Approximation Lemma. *Inform. Process. Lett.* https://doi.org/10.1016/S0020-0190(00)00220-9

Bart Jacobs and Jan J. M. M. Rutten. 1997. A Tutorial on (Co)Algebras and (Co)Induction. *Bulletin of The European Association for Theoretical Computer Science* 62 (1997), 62–222.

Ranjit Jhala and Niki Vazou. 2020. Refinement Types: A Tutorial. *Foundations and Trends in Programming Languages.* http://dx.doi.org/10.1561/2500000032

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. *POPL.* https://doi.org/10.1145/3009837.3009880

K. Rustan M. Leino and Michał Moskal. 2014. Co-induction Simply. *International Symposium on Formal Methods.* https://doi.org/10.1007/978-3-319-06410-9_27

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple Unification-Based Type Inference for GADTs. *ICFP.* https://doi.org/10.1145/1160074.1159811

Andrew Reynolds and Jasmin Christian Blanchette. 2017. A Decision Procedure for (Co)datatypes in SMT Solvers. In *Journal of Automated Reasoning.* https://doi.org/10.1007/s10817-016-9372-6

Grigore Roşu and Dorel Lucanu. 2009. Circular Coinduction: A Proof Theoretical Foundation. *Algebra and Coalgebra in Computer Science.* https://doi.org/10.1007/978-3-642-03741-2_10

Vlad Rusu and David Nowak. 2022. Defining Corecursive Functions in Coq Using Approximations. *ECOOP.* https://doi.org/10.4230/LIPIcs.ECOOP.2022.12

William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. 2012. Zeno: An automated prover for properties of recursive data structures. *TACAS.* https://doi.org/10.1007/978-3-642-28756-5_28

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is Reasonable Coq. *Certified Programs and Proofs.* https://doi.org/10.1145/3167092

Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. 2018. Theorem Proving for All: Equational Reasoning in Liquid Haskell (Functional Pearl). *Haskell.* https://doi.org/10.1145/3242744.3242756

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. *ICFP.* https://doi.org/10.1145/2628136.2628161

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *POPL.* https://doi.org/10.1145/3158141

Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen, and Dan Rosén. 2013. HALO: Haskell to Logic through Denotational Semantics. *POPL.* https://doi.org/10.1145/2429069.2429121

Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. *POPL.* https://doi.org/10.1145/604131.604150