Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Μηχανική κατασκευή συνεπαγωγικών (coinductive) αποδείξεων στη Liquid Haskell

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΛΥΚΟΥΡΓΟΣ ΜΑΣΤΟΡΟΥ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2022

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Μηχανική κατασκευή συνεπαγωγικών (coinductive) αποδείξεων στη Liquid Haskell

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΛΥΚΟΥΡΓΟΣ ΜΑΣΤΟΡΟΥ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 28η Ιουλίου 2022.

....	....	....
Νικόλαος Σ. Παπασπύρου Καθηγητής Ε.Μ.Π.	Κωνσταντίνος Σαγώνας Αν. Καθηγητής Ε.Μ.Π.	Νίκη Βάζου Ερευνήτρια IMDEA

Αθήνα, Ιούλιος 2022

....

Λυκούργος Μαστόρου
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η ορθότητα είναι μια επιθυμητή αλλά όχι τετριμμένη ιδιότητα του λογισμικού. Ένας πολύ αξιόπιστος τρόπος για να διασφαλιστεί η ορθότητα είναι η επαλήθευση λογισμικού. Η Liquid Haskell είναι ένας επαγωγικός επαληθευτής που βασίζεται σε έναν επιλύτη SMT και επεκτείνει το σύστημα τύπων της Haskell χρησιμοποιώντας λογικά κατηγορήματα. Χρησιμοποιώντας τη Liquid Haskell, είναι δυνατό να αποδειχθούν πολλές επιθυμητές ιδιότητες για κώδικα γραμμένο σε Haskell. Η Haskell, λόγω της οκνηρής αποτίμησης, μας επιτρέπει να ορίσουμε αντικείμενα που έχουν πιθανώς άπειρα στοιχεία, όπως άπειρες λίστες. Ωστόσο, η Liquid Haskell δεν είναι σε θέση να επαληθεύσει τέτοιους ορισμούς λόγω έλλειψη τερματισμού τους. Επίσης, για παρόμοιους λόγους, δεν μπορεί να επιχειρηματολογήσει για ιδιότητες τέτοιων ορισμών.

Αυτή η εργασία έχει στόχο να αντιμετωπίσει αυτό το χάσμα μεταξύ των δυνατοτήτων της Haskell και της εκφραστικότητας της Liquid Haskell. Παρουσιάζουμε μια τεχνική που μας βοηθά να επαληθεύσουμε την παραγωγικότητα των συναναδρομικών ορισμών, ακολουθώντας παρόμοια δουλειά σε άλλες γλώσσες. Παρουσιάζουμε επίσης δύο εναλλακτικές προσεγγίσεις, δηλαδή τις συνεπαγωγικές αποδείξεις με τη χρήση δεικτών και τις εποικοδομητικές συνεπαγωγικές αποδείξεις, οι οποίες κωδικοποιούν με συνέπεια συνεπαγωγικές αποδείξεις στη Liquid Haskell. Χρησιμοποιούμε αυτές τις κωδικοποιήσεις για να ελέγξουμε αυτόματα διάφορους ορισμούς και αποδείξεις, επιδεικνύοντας πώς μπορεί να χρησιμοποιηθεί ένας επαγωγικός επαληθευτής για να ελέγχθούν οι συνεπαγωγικές ιδιότητες και η παραγωγικότητα κώδικα Haskell.

Λέξεις κλειδιά

Γλώσσες προγραμματισμού, Προγραμματισμός με αποδείξεις, Πιστοποιημένος κώδικας, Συνεπαγωγικές αποδείξεις, Αυτοματοποιημένες αποδείξεις.

Abstract

Correctness is a desirable, yet not trivial, property of software. One very reliable way to ensure correctness is software verification. Liquid Haskell is an inductive verifier which is based on an SMT solver and extends Haskell's type system using logical predicates. Using Liquid Haskell, it is possible to prove many desired properties for code written in Haskell. Haskell, due to laziness, allows us to define objects that have possibly infinite elements, such as infinite lists. However, Liquid Haskell is not able to verify such definitions due to their lack of termination. Also, for similar reasons, it cannot reason about properties of such definitions.

This thesis aims to address this gap between Haskell's capabilities and Liquid Haskell's expressiveness. We present a technique which aids us to verify the productivity of corecursive definitions, following similar work in other languages. We also present two alternative approaches, namely indexed and constructive coinduction, to consistently encode coinductive proofs in Liquid Haskell. We use these encodings to machine check various definitions and proofs, showcasing how an inductive verifier can be used to check coinductive properties and productivity of Haskell code.

Key words

Programming languages, Programming with proofs, Certified code, Coinduction, Refinement types, Theorem Proving.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω πρωτίστως τον επιβλέποντά μου, Νίκο Παπασπύρου, τόσο για τη συμβολή του στα πλαίσια αυτής της εργασίας όσο και για τις γνώσεις που πήρα για το αντικείμενο των Γλωσσών Προγραμματισμού μέσα από τα μαθήματα που διδάσκει. Επίσης πολλές ευχαριστίες θέλω να δώσω στη Νίκη Βάζου για τη συνεπίβλεψη της εργασίας. Οι υποδείξεις της αλλά και το ενδιαφέρον της αποτέλεσαν μεγάλη βοήθεια όλον αυτόν τον καιρό.

Είμαι ευγνώμων για όλα τα χρόνια στη σχολή τόσο για τις γνώσεις που έλαβα όσο και για την ευρύτερη εμπειρία. Ως εκ τούτου θα ήθελα να ευχαριστήσω το προσωπικό της σχολής για το ενδιαφέρον που έχει για τους φοιτητές και την προαγωγή των γνώσεων και του επιστημονικού πνεύματος στη σχολή. Επίσης θα ήθελα να ευχαριστήσω τους συμφοιτητές μου για την κοινή πορεία και την συμπαράσταση κατά τη διάρκεια των σπουδών.

Τέλος θα ήθελα να ευχαριστήσω την οικογένεια και τους φίλους μου που όλα αυτά τα χρόνια με στηρίζουν με την αγάπη τους.

Λυκούργος Μαστόρου,

Αθήνα, 28η Ιουλίου 2022

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-2-22, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2022.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Περιεχόμενα

Κατάλογος σχημάτων

Σχήματα στο αγγλικό κείμενο

Κεφάλαιο 1

Εισαγωγή

Το λογισμικό είναι άρρηκτα συνδεδεμένο με τη διαδικασία αποσφαλμάτωσης: Συχνά κάνουμε λάθη στην προσπάθεια να κωδικοποιήσουμε τη λογική μας, ή η λογική μας είναι εξ αρχής προβληματική. Κάποιες φορές μπορούμε να αρκεστούμε σε μια αφελή προσέγγιση χωρίς να έχουμε κάποια συστηματική μέθοδο πρόληψης λαθών. Ωστόσο, όταν το λογισμικό γίνεται περίπλοκο παρόμοιες προσεγγίσεις μπορεί να αποβούν καταστροφικές.

Πολλαπλές τεχνικές έχουν αναπτυχθεί ώστε να διασφαλιστεί η ορθότητα του λογισμικού. Η δοκιμή λογισμικού (software testing) είναι ίσως μία από τις πιο δημοφιλείς: αποτελείται από τη συγγραφή κώδικα που εκτελεί μέρος του λογισμικού μας με εισόδους δείγματα και συγκρίνει τις εξόδους με αναμενόμενες τιμές. Όμως με αυτήν την τεχνική δεν έχουμε πραγματικά αποδείξει την ορθότητα του λογισμικού μας, αλλά έχουμε μια πιθανοτική διαβεβαίωση ότι ο κώδικας είναι ορθός, ανάλογα και με το πλήθος και την ποικιλία των σεναρίων ελέγχου, και την πολυπλοκότητα του λογισμικού.

Μια άλλη προσέγγιση στην ορθότητα είναι η χρήση τύπων. Οι τύποι είναι μια αρκετά εδραιωμένη έννοια που μας επιτρέπει να παρέχουμε ένα πλαίσιο στις διάφορες τιμές που χρησιμοποιούνται μέσα σε ένα πρόγραμμα. Ανάλογα με τη φύση του συστήματος τύπων, μας επιτρέπεται να βεβαιώσουμε κάποιες ιδιότητες του κώδικά μας. Υπάρχουν ακόμα και γλώσσες, όπως η Agda [Team22] και η Coq [Barr97], που χρησιμοποιούν τον ισομορφισμό Curry-Howard (ο οποίος σχετίζει προγράμματα με μαθηματικές αποδείξεις) μέσω του συστήματος τύπων τους έτσι ώστε να αποδείξουν ιδιότητες του λογισμικού.

Liquid Haskell Σε αυτόν τον τομέα έχει αναπτυχθεί και το εργαλείο της Liquid Haskell. Η Liquid Haskell είναι ένα εργαλείο το οποίο επεκτείνει το σύστημα τύπων της γλώσσας Haskell με τη χρήση λογικών κατηγορημάτων, τα οποία μας επιτρέπουν να συγκεκριμενοποιήσουμε τους τύπους που δέχονται και επιστρέφουν οι συναρτήσεις. Για παράδειγμα η συνάρτηση `head`

```
data [a] = [] | a : [a]
head :: [a] → a
head (x:_)  = x
```

διαφημίζει, μέσω της υπογραφής τύπου της, ότι δέχεται μία λίστα από στοιχεία τύπου a και επιστρέφει ένα στοιχείο τύπου a. Ο τύπος αυτός ωστόσο δεν είναι ακριβής, καθώς σε μια άδεια λίστα η συνάρτηση αυτή θα δημιουργήσει σφάλμα αφού δεν είναι ορισμένη για άδειες λίστες. Ο ορθός τύπος του ορίσματος της συνάρτησης `head` είναι η μη άδεια λίστα, η οποία μπορεί να εκφραστεί στη Liquid Haskell ως ο τύπος `NonEmpty a` που περιγράφουμε παρακάτω:

```
{-@ measure empty @-}
empty :: [a] → Bool
empty [] = True
empty _  = False

{-@ type NonEmpty a = {v:[a] | not (empty v) @-}
```

Η Liquid Haskell έχει επίσης την ικανότητα να αποδεικνύει επαγωγικές ιδιότητες του κώδικα. Για παράδειγμα μπορεί να αποδειχθεί ότι η συνάρτηση map διατηρεί το μήκος της λίστας που παίρνει σαν όρισμα:

```
map :: (a → b) → x:[a] → {l:[b] | len l = len x}
map f []     = []
map f (x:xs) = f x : map f xs

len :: List a → {v:Int| v >= 0}
len [] = 0
len (_:xs) = 1 + len xs
```

Η Liquid Haskell χρησιμοποιεί το σώμα της map ώστε να αποδείξει την επιθυμητή ιδιότητα. Η αναδρομική κλήση map f xs χρησιμοποιείται ως επαγωγική υπόθεση ενώ η περίπτωση της άδειας λίστας είναι η βάση της επαγωγής.

Σε περιπτώσεις που η απόδειξη περιλαμβάνει πολλαπλούς ορισμούς συναρτήσεων η Liquid Haskell προσφέρει τη δυνατότητα της αναλυτικής συγγραφής μιας απόδειξης, χρησιμοποιώντας τους διασαφηνισμένους τελεστές του σχήματος 1.1. Μια ιδιότητα που αποδεικνύουμε με αυτόν τον τρόπο είναι η ιδιότητα συνένωσης των map (map fusion):

```
mapFusion :: f:(b → c) → g:(a → b) → xs:[a] → {map f (map g xs) = map (f
   . g) xs}
mapFusion f g [] = ()
mapFusion f g (x:xs)
  =   map f (map g (x:xs))
  === map f (g x : map g xs)
  === f (g x) : map f (map g xs)
    ? mapFusion f g xs
  === (f . g) x : map (f . g) xs
  === map (f . g) (x:xs)
  *** QED
```

Εδώ η επαγωγή γίνεται πιο εμφανής λόγω της αναλυτικότητας. Η υπογραφή τύπου δηλώνει το θεώρημα που θέλουμε να αποδείξουμε. Ο τελεστής (===) έχει τέτοιο τύπο ώστε να ελέγχει την ισότητα των διαδοχικών βημάτων ενώ το (?) μπορεί να προσθέτει δεδομένα στην απόδειξη, όπως εδώ η επαγωγική υπόθεση. Τέλος με την έκφραση *** QED ολοκληρώνεται η απόδειξη.

Αντικείμενα απείρου μήκους Η Haskell μας επιτρέπει ακόμα, μέσω της οκνηρής αποτίμησης, να ορίσουμε άπειρα αντικείμενα. Ένας τύπος δεδομένων που συχνά χρησιμοποιείται για να δείξει αυτήν την ιδιότητα είναι τα Streams (ροές δεδομένων):

```
data Stream a = a :> Stream a

shead (x :> xs) = x
stail (x :> xs) = xs
```

Η δομή του Stream μοιάζει με αυτήν της λίστας με τη διαφορά ότι δεν περιλαμβάνει το άδειο stream. Έτσι ένα stream δεν τερματίζει ποτέ· είναι μια άπειρη σειρά στοιχείων τύπου a.

Παρότι στη Haskell τέτοιοι τύποι χρησιμοποιούνται κατά κόρον, δεν υποστηρίζονται από τη Liquid Haskell όπως θα δούμε στη συνέχεια. Η δουλειά αυτή εργάζεται πάνω σε αυτό το κενό μεταξύ της Haskell και της Liquid Haskell. Συγκεκριμένα κωδικοποιούμε τρόπους με τους οποίους μπορούμε να αποδείξουμε την παραγωγικότητα (μια έννοια δυϊκή του τερματισμού) ενός ορισμού για άπειρα αντικείμενα καθώς και να αποδείξουμε ιδιότητες τέτοιων αντικειμένων.

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

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

Σχήμα 1.1: Αποδεικτικοί τελεστές της Liquid Haskell

Κεφάλαιο 2

Παραγωγικότητα συναναδρομικών ορισμών

Το πρώτο πρόβλημα που συναντούμε όταν προσπαθούμε να ορίσουμε το Stream είναι ότι η Liquid Haskell το απορρίπτει με ένα error που υποδηλώνει ότι αυτός ο τύπος δεν είναι επαγωγικός. Μπορούμε να απενεργοποιήσουμε τον συγκεκριμένο έλεγχο με το flag της Liquid Haskell `"--no-adt"`. Αυτό μας προειδοποιεί και για την μη υποστήριξη τέτοιων ειδών δεδομένων από τη Liquid Haskell.

Στη συνέχεια προχωρούμε σε έναν ορισμό που είναι αρκετά γνωστός στην κοινότητα της Haskell, την ακολουθία των αριθμών Fibonacci:

```
fib = 0 :> 1 :> 1 :> zipWith (+) fib (stail fib)
  where zipWith f (x:>xs) (y:>ys)
          = f x y :> zipWith f xs ys
```

Ορισμοί όπως η `fib` που παράγουν άπειρα αντικείμενα με κλήσεις στον εαυτό τους τις καλούμε *συναναδρομικές* για να τις διαχωρίσουμε από τις αναδρομικές, οι οποίες καλούν τον εαυτό τους με μειούμενα ορίσματα παράγωντας πεπερασμένες εξόδους. Η Liquid Haskell, σύμφωνα και με όσα είπαμε νωρίτερα, θα επισημάνει με ένα σφάλμα ότι η συνάρτηση αυτή δεν τερματίζει. Ωστόσο προσθέτοντας την σήμανση `lazy fib` μπορούμε να απενεργοποιήσουμε τον έλεγχο τερματισμού για τη συγκεκριμένη συνάρτηση.

Αυτή η τακτική όμως δεν έχει την ακρίβεια που θα θέλαμε. Για παράδειγμα φέρνουμε τη συνάρτηση `loop`:

```
loop = loop
```

Η συνάρτηση `loop` γίνεται δεκτή από τη Liquid Haskell με τη σήμανση `lazy loop`. Ωστόσο υπάρχει διαφορά μεταξύ της `fib` και της `loop`.

Η `fib` έχει την ιδιότητα ότι μπορεί να παρατηρηθεί μερικώς: μπορεί να αποτιμηθεί η κεφαλή της, η ουρά της αλλά και οποιοδήποτε πεπερασμένο τμήμα της σε πεπερασμένο χρόνο. Εν αντιθέσει, οποιαδήποτε αποτίμηση της `loop` θα καταλήξει σε έναν ατέρμονα υπολογισμό. Θα ήταν χρήσιμο λοιπόν, εφόσον ασχολούμαστε με δεδομένα απείρου μήκους, να μπορούμε να διαχωρίζουμε ορισμούς που "κολλάνε", όπως η `loop`, από ορισμούς που είναι *παραγωγικοί* όπως η `fib`.

Όπως είπαμε για να ελεγχθεί ο τερματισμός σε συναρτήσεις με λίστες χρησιμοποιείται ως μετρική το μέγεθος της λίστας. Στην προκειμένη περίπτωση κάτι τέτοιο δεν είναι δυνατό: τα streams έχουν πάντα άπειρο μέγεθος και έτσι η μετρική του μεγέθους δεν έχει νόημα.

Υπάρχει όμως η παρεμφερής έννοια του βάθους η οποία μπορεί να μας βοηθήσει. Χρησιμοποιούμε μεταβλητές φυσικών αριθμών που αντιπροσωπεύουν το βάθος, το μέγεθος δηλαδή του προοιμίου ενός stream το οποίο είναι ορισμένο χωρίς να κολλάει.

Για παράδειγμα φέρνουμε τη συνάρτηση `loop2`:

```
loop2 = 1 :> 2 :> stail (stail loop2)
```

```
measure depth :: Stream a → Nat
measure inf  :: Stream a → Bool

type StreamS a S = {v:Stream a | depth v = S}
type StreamG a S = {v:Stream a | depth v >= S || inf v}
type StreamI a I = {v:Stream a | inf v}

type LT I = {j:Nat | j < I}

assume cons :: i:Nat
            → (LT i → a) → (j:LT i → StreamG a j)
            → StreamS a i
cons _ fx fxs = fx 0 :> fxs 0

shead :: j:Nat → {xs:Stream a | depth xs > j || inf xs} → a
shead j (x :> xs) = x

assume stail :: j:Nat
             → {xs:Stream a | depth xs > j || inf xs}
             → {v:StreamS a j | inf xs ==> inf v}
stail i (x :> xs) = xs
```

Σχήμα 2.1: Κατασκευαστές και καταστροφείς του `Stream` με βάθη

Στη συνάρτηση αυτή έχουμε πάλι αυτοαναφορά: η ουρά της ουράς του `loop2` ορίζεται ως ο εαυτός της. Έτσι στο `loop2`, σύμφωνα με τον ορισμό που δώσαμε, μπορεί να δοθεί βάθος ≤ 2. Οι παραγωγικές συναρτήσεις λοιπόν θα είναι αυτές που μπορούμε να τις αναθέσουμε οποιοδήποτε βάθος.

Για να μπορέσουμε να κρατάμε λογαριασμό του βάθους τροποποιούμε τον κατασκευαστή και τους καταστροφείς (το `shead` και το `stail` δηλαδή) του `Stream` με διασαφηνιστικούς τύπους οι οποίοι αναφέρονται στα βάθη των εισόδων εξόδων, όπως φαίνεται στο σχήμα 2.1.

Σημείωση: Οι σημάνσεις `measure` εδώ συστήνουν στη Liquid Haskell τα κατηγορήματα `depth` και `inf`. Η Liquid Haskell αρχικά ξέρει μόνο τις υπογραφές τύπου τους. Όλα τα άλλα δεδομένα για αυτά τα κατηγορήματα τα συλλέγει από τους διασαφηνισμένους τύπους των `cons`, `tail` και `toInf` (σχήμα 2.2). Οι συναρτήσεις αυτές έχουν υπογραφές τύπων που αρχίζουν με την σήμανση **`assume`** η οποία κάνει τη Liquid Haskell να δεχτεί ως αξιώματα αυτούς τους τύπους.

Με αυτό το σύστημα κάθε ορισμός μετατρέπεται και σε επαγωγική απόδειξη παραγωγικότητας. Συγκεκριμένα, το `cons`, για κάθε φυσικό αριθμό `i`, φτιάχνει ένα stream βάθους `i`. Αυτό το πραγματοποιεί χρησιμοποιώντας ένα `Stream` που έχει βάθος `>= j` για ουρά, για οποιοδήποτε βάθος `j < i` και ένα απλό στοιχείο για κεφαλή. Όπως θα δούμε αργότερα η `cons` παρέχει τα βάθη `j` στις συναρτήσεις που παίρνει σαν ορίσματα οι οποίες μπορούν να γραφτούν σαν λάμδα εκφράσεις. Η `shead` και η `stail` χρησιμοποιούν το βάθος `j`, το οποίο είναι μικρότερο του βάθους του `xs`, ως μάρτυρα ότι το `xs` έχει βάθος `> 0` και άρα μπορεί να γίνει ασφαλής πρόσβαση στην κεφαλή και στην ουρά του. Επιπλέον η `stail` χρησιμοποιεί το `j` ώστε να δηλώσει ότι το βάθος της ουράς είναι μικρότερο του βάθους του αρχικού stream.

Με τη χρήση αυτών των συναρτήσεων μπορούμε να ξαναγράψουμε τον ορισμό της `fib` ώστε να αποδεικνύεται η παραγωγικότητά της:

```
zipWith :: i:Nat
        → (a → b → c)
        → StreamG a i → StreamG b i
        → StreamG c i
zipWith i f xs ys = cons i
                    (\j → f (shead j xs) (shead j ys))
                    (\j → zipWith j f (stail j xs) (stail j ys))

fib :: i:Nat → StreamG Int i
fib i = cons i (const 0)
             $ \j → cons j (const 1)
             $ \k → zipWith k (fib k) (stail k (fib j))

const x _ = x
```

Η `zipWith` παίρνει δυο streams που είναι ορισμένα σε βάθος `i` και επιστρέφει ένα με το ίδιο βάθος `i`. Αυτό αποδεικνύεται επαγωγικά από τις υπογραφές τύπου των `cons`, `shead` και `stail`. Ο έλεγχος τερματισμού επιβεβαιώνει ότι η επαγωγή μας είναι σωστή και σε αυτό βοηθούν οι μεταβλητές βαθών που μας επιτρέπουν να καλέσουμε την `zipWith` με μειούμενο όρισμα (`j < i`).

Η `fib` αφού ορίσει τα δύο πρώτα στοιχεία με `cons` κάνει διαθέσιμες τις μεταβλητές βαθών `j` και `k`. Αυτό μας αφήνει με την υποχρέωση να δείξουμε πως το υπόλοιπο stream έχει βάθος `>= k`, ώστε η `fib i` να έχει βάθος `i`. Το πρώτο όρισμα της `zipWith` έχει βάθος `k` λόγω της επαγωγικής υπόθεσης, η οποία είναι έγκυρη αφού `k < i`. Στο δεύτερο όρισμα της `zipWith` βλέπουμε πως καλείται η `stail k` πάνω στη `fib j`. Η `fib j` έχει βάθος `j` λόγω της επαγωγικής υπόθεσης και άρα η κλήση στην `stail` είναι έγκυρη αφού `j > k`. Έτσι αποδεικνύεται πως και η έκφραση `stail k (fib j)` έχει βάθος `k` και άρα, λόγω της `zipWith` η όλη έκφραση έχει βάθος `k`.

Αν προσπαθήσουμε να γράψουμε την `loop2` με αυτό το σύστημα καταλήγουμε στο εξής:

```
loop2 :: i:Nat → StreamG
loop2 i = cons i (const 1) $ \j → cons j (const 2)
                           $ \k → stail ?? (stail k (loop2 j))
```

Στην κλήση της `loop2` αναγκαζόμαστε να δώσουμε όρισμα `j` (και όχι `i`) ώστε να ικανοποιείται ο έλεγχος τερματισμού (και να είναι ορθή η επαγωγή). Το πρώτο `stail` αναγκαστικά παίρνει όρισμα `k` για να ισχύει `k < j`. Στη δεύτερη `stail` δεν έχουμε κάποια μεταβλητή βάθους η οποία να είναι `< k` οπότε δεν μας επιτρέπεται να ολοκληρώσουμε την απόδειξη παραγωγικότητας της `loop2` – που είναι ακριβώς το αποτέλεσμα που επιθυμούσαμε.

Ολοκληρωμένοι ορισμοί Για να εκφράσουμε την παραγωγικότητα ενός ορισμού, ότι δηλαδή ο ορισμός έχει άπειρο βάθος, μπορούμε να χρησιμοποιήσουμε το κατηγόρημα `inf` (σχήμα 2.1). Το κατηγόρημα αυτό αφορά ολοκληρωμένους ορισμούς που έχουν ήδη απόδειξη παραγωγικότητας. Σε τέτοιους ορισμούς μπορούμε να κάνουμε όσες προσβάσεις θέλουμε χωρίς να μας περιορίζουν οι μεταβλητές βαθών χρησιμοποιώντας τις συναρτήσεις του σχήματος 2.2. Για παράδειγμα μπορούμε να πάρουμε το πέμπτο στοιχείο της ακολουθίας Fibonacci χρησιμοποιώντας την έκφραση `ihead (itail (itail (itail (itail (toInf fib)))))`.

```
assume toInf :: (i:Nat → StreamG a i) → StreamI a
toInf f = f 0

ihead :: StreamI a → a
ihead xs = shead 0 xs

itail :: StreamI a → StreamI a
itail xs = stail 0 xs
```

Σχήμα 2.2: Συναρτήσεις για πλήρως ορισμένα streams

Κεφάλαιο 3

Αποδείξεις συνεπαγωγικών ιδιοτήτων

Στο μέρος 1 περιγράψαμε την ιδιότητα της συνένωσης των map (map fusion) η οποία ως ορισμένη για λίστες αποδεικνύεται επαγωγικά. Ορίζουμε το ίδιο θεώρημα για streams ως εξής:

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

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

Υπενθυμίζουμε ότι για να γίνει δεκτός ο τύπος `Stream` από την Liquid Haskell πρέπει να απενεργοποιήσουμε κάποιους ελέγχους με το flag `"--no-adt"`. Προσπαθούμε αρχικά να δομήσουμε την απόδειξη ακολουθώντας την αντίστοιχη απόδειξη για λίστες:

```
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
```

Αυτός ο κώδικας γίνεται δεκτός από τη Liquid Haskell. Ωστόσο η απόδειξη αυτή δεν είναι επαγωγική, αφού ούτε βάση υπάρχει, ούτε η επαγωγική υπόθεση έχει μειούμενο όρισμα. Πράγματι η Liquid Haskell έχει γίνει ασυνεπής λόγω της χρήσης του `"--no-adt"`, όπως φαίνεται από το παρακάτω:

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

Η συνάρτηση `falseStream` αποδεικνύει το `false` με λανθασμένη επαγωγή που όμως επιτρέπει η Liquid Haskell.

Συνεπαγωγικές αποδείξεις με δείκτες Βλέπουμε ότι το να αποδείξουμε μια συνεπαγωγική (coinductive) απόδειξη, μια απόδειξη δηλαδή η οποία αφορά αντικείμενα όπως τα stream, είναι κάτι που δεν υποστηρίζεται από τη Liquid Haskell. Αφού η ισότητα των streams (ως συνεπαγωγική ιδιότητα) δεν αποδεικνύεται με ευθύ τρόπο προσπαθούμε να αποδείξουμε την ισότητα των πρώτων `k` στοιχείων, η οποία αποτελεί επαγωγική ιδιότητα. Η ισότητα αυτή μπορεί να εκφραστεί ως εξής:

```
eqK :: Eq a ⇒ Stream a → Stream a → k:Nat → Bool
```

```
(=#=) :: 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}
```

Σχήμα 3.1: Τελεστής απόδειξης για ισότητα των πρώτων k στοιχείων

```
eqK _ _ 0 = True
eqK (x:>xs) (y:>ys) k = x == y && eqK xs ys (k-1)
```

Η απόδειξη του θεωρήματός μας με αυτήν την ιδιότητα έχει ως εξής:

```
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
```

Ο τελεστής =#= είναι ορισμένος στο σχήμα 3.1 και χρησιμεύει ώστε να μπορέσουμε να προωθήσουμε την ισότητα των προοιμίων των streams, ενώ το # απλά μας βοηθάει με τις προτεραιότητες τελεστών ώστε να μην χρειάζονται παρενθέσεις.

Η ιδιότητα που μόλις αποδείξαμε είναι, όπως είπαμε, επαγωγική και άρα δεν έχει το πρόβλημα ασυνέπειας που περιγράψαμε προηγουμένως. Επιπρόσθετα φαίνεται ότι αυτή η απόδειξη είναι ισοδύναμη με την απλή ισότητα των streams, αφού αν δυο streams είναι ίσα σε οποιοδήποτε μήκος προοιμίου θα είναι και ίσα. Η πρόταση αυτή υπάρχει στη βιβλιογραφία ως take lemma [Bird88] και μπορούμε να την κωδικοποιήσουμε σε Liquid Haskell:

```
assume takeLemmaEq :: x:Stream a → y:Stream a
                   → (k:Nat → {eqK x y k}) → {x = y}
```

Με τη χρήση αυτής της πρότασης μπορούμε να ολοκληρώσουμε την αρχική μας απόδειξη:

```
mapFusion f g xs =
  takeLemmaEq (smap f (smap g xs)) (smap (f . g) xs) (mapFusionIdx f g xs)
```

Κατασκευαστικές συνεπαγωγικές αποδείξεις Άλλος τρόπος που μπορούμε να προσεγγίσουμε τις συνεπαγωγικές αποδείξεις είναι να ορίσουμε κατασκευαστικά συνεπαγωγικά κατηγορήματα τύπου Coq [Chli13] χρησιμοποιώντας data propositions [Bork22] της Liquid Haskell σε συνδυασμό με διασαφηνισμένα GADTs [Peyt06]. Η ισότητα για streams μπορεί να εκφραστεί με αυτόν τον τρόπο ως:

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

Αυτός ο τύπος έχει έναν κατασκευαστή (έναν τρόπο δηλαδή να κατασκευάσουμε μια ισότητα), ο οποίος με ένα στοιχείο x για κεφαλή, δύο streams xs και ys και μια απόδειξη της πρότασης `xs = ys`, κατασκευάζει μια απόδειξη της πρότασης `x :> xs = x :> ys`.

Αυτή η τεχνική όμως πάλι δεν αρκεί. Η Coq στα συνεπαγωγικά κατηγορήματά της επιβάλλει την ιδιότητα "guardedness", η οποία είναι υποσύνολο της παραγωγικότητας που περιγράψαμε στο κεφάλαιο 2. Χωρίς αυτόν τον έλεγχο καταλήγουμε πάλι σε ασυνεπές σύστημα, όπως φαίνεται από το παρακάτω:

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

Η λύση λοιπόν που θα χρησιμοποιήσουμε είναι να προσθέσουμε τον έλεγχο παραγωγικότητας του κεφαλαίου 2 στην ισότητα των streams:

```
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))
```

Με τη χρήση του `EqC` μπορούμε να αποδείξουμε εκ νέου την ιδιότητά μας:

```
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
```

Στην οποία το `EqRefl` είναι ο σκελετός της απόδειξης και τα `lhs` και `rhs` επεκτείνουν τη δεξιά και την αριστερή μεριά της ισότητας αντίστοιχα, ώστε να μπορεί να εφαρμοστεί η επαγωγική υπόθεση.

Τέλος, για να μετατρέψουμε την απόδειξη αυτή σε απόδειξη ισότητας, μπορούμε να χρησιμοποιήσουμε ένα αξίωμα σαν αυτό της προηγούμενης μεθόδου:

```
assume eqLemma :: x:Stream a → y:Stream a
               → (i:Nat → Prop (EqC i x y)) → {x = y}

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

Κεφάλαιο 4

Συμπεράσματα

Χρησιμοποιήσαμε τη Liquid Haskell για να υποστηρίξουμε συνεπαγωγικά χαρακτηριστικά δηλαδή να αποδείξουμε την παραγωγικότητα των διάφορων συναναδρομικών ορισμών και συνεπαγωγικές ισότητες.

Πετύχαμε τον έλεγχο παραγωγικότητας αλλάζοντας τους κατασκευαστές και καταστροφείς συνεπαγωγικών αντικειμένων για να παρακολουθείτε το βάθος του αντικείμενο και χρησιμοποίησε αυτή την υποδομή για να ορίσει και να παρέχει την παραγωγικότητα από διάφορα αντικείμενα.

Κωδικοποιήσαμε συνεπαγωγικές αποδείξεις στον επαγωγικό επαληθευτή χρησιμοποιώντας δύο προσεγγίσεις. Στην προσέγγιση με δείκτες, προσθέτουμε στην ισότητα έναν φυσικό αριθμό k και η απόδειξη γίνεται με επαγωγή στο k. Στην κατασκευαστική προσέγγιση, η ισότητα κωδικοποιείται ως εκλεπτυσμένο GADT το οποίο ελέγχεται για την παραγωγικότητά του. Χρησιμοποιώντας οποιαδήποτε από αυτές τις προσεγγίσεις, ένας προγραμματιστής Haskell μπορεί να έλεγξει συνεπαγωγικές ιδιότητες κώδικα Haskell στη Liquid Haskell.

Όλες οι μέθοδοι που περιγράψαμε μπορούν να γενικευτούν και για άλλους απείρους τύπους πέρα από streams, όπως δένδρα ή πιθανώς άπειρες λίστες. Επιπλέον μπορούν να αποδειχθούν και άλλες ιδιότητες πέρα από την ισότητα, π.χ. η λεξικογραφική σύγκριση. Παραδείγματα όλων αυτών υπάρχουν στη διεύθυνση github.com/lykmast/co-liquid. Επίσης, ένα μεγάλο μέρος της παρούσας διπλωματικής έχει γίνει δεκτό για παρουσίαση στο φετινό Haskell Symposium [Mast22].

Κείμενο στα αγγλικά

Chapter 1

Introduction

Software, for anyone that has attempted to produce it, is closely coupled with the process of debugging: We often make mistakes while encoding our logic, or our logic is faulty to begin with. Sometimes we can get away with approaching software naively, without a strategy to catch errors. However, when software becomes complex or has critical functionality, such naiveness may be catastrophic.

Multiple techniques have been developed for ensuring that software is correct. Testing is perhaps the most widely used: it comprises of writing code that runs part of our software with sample inputs and compares the output to expected values. Though, as Dijkstra famously said, “Program testing can be used to show the presence of bugs, but never to show their absence!” With testing we don’t actually prove correctness, but we get a probabilistic assurance that our software is correct, depending on the number and diversity of test-cases, and the complexity of our software.

Another approach to correctness is types. Types are a very traditional concept that allows us to provide context for values that are used throughout a program. Depending on the nature of the type system it can allow us to assert certain properties of our code. There are even languages like Agda [Team22] and Coq [Barr97] that use the Curry-Howard isomorphism (which relates computer programs to mathematical proofs), through their type system to formally prove software properties.

1.1 Liquid Haskell

Haskell’s type system is famous for its expressiveness. With *algebraic datatypes* we can define types with sums and products. Its usefulness can become apparent when defining the types for an interpreter:

```
data Expr = Add Int Int
          | Mul Int Int
          | Lt  Int Int
          | Eq  Int Int
          | Or  Bool Bool
          | Not Bool
          ...
```

We also have the ability to inductively define types such as lists:

```
data [a] = a : [a] | []
```

The power of this type system is famously put into words with the saying ”If it compiles, it is correct!”. However this view overestimates the expressiveness of Haskell’s type system.

There are ample properties that we cannot express in this type system and, hence, cannot ensure at compile time. Consider the following:

```
tail :: [a] → [a]
tail (x : xs) = xs
```

This function is happily accepted by Haskell's type system. In fact, `tail` is part of *Prelude*, Haskell's standard library. What happens then if we call `tail` on an empty list? We get a runtime error! The problem is that the type of `tail` is incorrect, as the function does not work for all lists, but only for non-empty ones.

Liquid Haskell [Vazo14] is a tool that extends the type system of Haskell by adding logical predicates to types, in order to cover this possibility of incorrectness. The logical predicates are translated to logical constraints which are passed to an SMT solver. The SMT solver then informs us whether our constraints are satisfiable or not.

Indeed, Liquid Haskell will not accept the above definition of `tail` because it is non-total in its argument (since it only covers the non-empty case of a list). In order to correct `tail` we can define a type that describes a non-empty list, using a predicate on lists that expresses the property of being empty.

```
{-@ measure empty @-}
empty :: [a] → Bool
empty [] = True
empty _  = False

{-@ type NonEmpty a = {v:[a] | not (empty v) @-}
```

Note: The `measure` annotation tells Liquid Haskell to reflect the definition of the corresponding function in logic so that it can be used as a predicate in `NonEmpty`. Liquid Haskell annotations are enclosed between `{-@ @-}`, which we mostly omit in the rest of this work, along with measure annotations, in order to avoid cluttering.

Using `NonEmpty` we can refine the definition of `tail` so that we won't get a runtime error:

```
{-@ tail :: NonEmpty → [a] @-}
tail (x : xs) = xs

unsafe = tail []     -- Refinement Type Error
safe   = tail (1:[])
```

`tail` now is total and Liquid Haskell will prevent us from calling it on an empty list, by throwing an error at compiler time.

1.1.1 Verification of properties in Liquid Haskell

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 list's length:

```
map :: (a → b) → x:[a] → {l:[b] | len l = len x}
map f []     = []
map f (x:xs) = f x : map f xs

len :: List a → {v:Int| v >= 0}
len [] = 0
len (_:xs) = 1 + len xs
```

Liquid Haskell verifies that `map` preserves the length of its argument using the definition of `map` as a proof. In the empty (`[]`) case the property holds trivially, since the result is also `[]` which has a length of 0. In the cons case, using the definition of `len` the argument has length `1 + len xs` while the result has length `1 + len (map f xs)` which can be proven equal by the inductive hypothesis (`len xs = len (map f xs)`).

Note: termination Liquid Haskell automatically checks that our functions are terminating i.e., for all arguments they do not lead to infinite computation. It does so by ensuring that, when a function recurses, it does so on a *decreasing* argument that also has a minimal value that it decreases towards. For arguments like natural numbers this tactic is straightforward. In the `map` function the decreasing argument is the list itself. What actually decreases in each recursive step is the size of the list, which, in the empty case, takes its minimal value of zero. Indeed, we can observe that, in the recursive case, `map` recurses on the tail of its list argument, and so will eventually lead to the base case by recursing on the empty list.

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.

Such an example is the famous map fusion property:

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

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

The rule defined above replaces the left-hand side `map f (map g xs)` with `map (f . g) xs`, traversing the list only once which optimizes the correspondent program. It is useful to be able to prove such properties before asserting a rule to the compiler.

To prove such a property we need to employ the theorem proving capabilities of Liquid Haskell [Vazo18] that encode theorems as refinement type specifications and proofs as inhabitants to these types.

We start by encoding our theorem with the following signature:

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

The notation `{ p }` is an abbreviation of the unit type with the predicate `p`, i.e., `{v:()| p}`. Thus, `mapFusion` only returns a unit value. This is because we don't need anything from a proof at runtime, we solely need it to prove the desired property at compile time.

To actually prove `mapFusion` we need to construct an inhabitant of the previously defined signature, i.e., a definition of `mapFusion`'s body that is accepted by Liquid Haskell:

```
mapFusion f g [] = ()
mapFusion f g (x:xs)
  =   map f (map g (x:xs))
  === map f (g x : map g xs)
  === f (g x) : map f (map g xs)
    ? mapFusion f g xs
  === (f . g) x : map (f . g) xs
  === map (f . g) (x:xs)
  *** QED
```

In the empty case the proof is trivial; we need only define the result as `()` and the rest is automated by Liquid Haskell's rewriting [Vazo17]. The non-empty case starts by the left-hand side and performing equational steps arrives at the right-hand side.

This equational reasoning is encoded as a Haskell function using a set of Haskell operators that are refined to check equalities at each equational step. These operators are defined in the Liquid Haskell

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

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

Figure 1.1: Proof Combinators of Liquid Haskell

library `ProofCombinators` and are summarized in Figure 1.1. Operator `(===)` takes two arguments, checks their equality and returns the first. In that way, it accumulates proof steps and propagates the equality from the left-hand (resp. right-hand) side to the right (resp. left) one. Operator `(?)` ignores its second argument and returns the first. It is solely used to invoke facts (such as other lemmas or the inductive hypothesis), which Liquid Haskell takes into account parallel to equational reasoning. Finally `*** QED` simply completes the proof by turning the result into a unit.

Specifically, in the `mapFusion` proof, we start by expanding twice the definition of `map` using `===` steps. We notice that we have arrived at the expression `f (g x) : map f (map g xs)`. The sub-expression `map f (map g xs)` is eligible for the `mapFusion` property and so we can invoke the inductive hypothesis on the tail of the list with `? mapFusion f g xs` to transform it to `map (f . g) xs`. Finally, by folding the definitions of `(.)` and `map`, we arrive at the desired result `map (f . g) (x : xs)` and complete the proof with `*** QED`.

The validity of this proof depends on the following: For one, the refinement checks of `(===)` ensure that each step is congruent to the previous one. Secondly, the proof is checked to be total, which ensures that we truly prove the property for all possible arguments of this type. Lastly, the proof is checked to be terminating, as we previously described. That means that when invoking the inductive hypothesis we need to do so on a term that decreases to a base case, making our induction well-formed.

1.2 Corecursive definitions

Laziness is one of the most distinctive features of Haskell. It describes the evaluation model of Haskell, where an expression is only evaluated when its result is needed. This feature allows us to define and use objects with infinite size, such as streams, which are defined as follows:

```
data Stream = a :> Stream a
```

We can observe that streams are very similar to lists; they basically describe infinite lists. This allows us to easily adapt many functions that are defined for lists to work on streams.

Streams are defined using a technique called *corecursion*. Corecursion, like recursion, describes self-calling algorithms which, rather than destructing data until they reach a base case (as is done in a recursive setting), build up from a base case producing data.

For example we can define `srepeat`, which takes an argument and returns a stream that consists of this argument repeated infinite times:

```
srepeat :: a → Stream a
srepeat x  = x :> srepeat x
```

A very popular example of an infinite list definition that illustrates the usefulness and elegance of laziness is the definition of the infinite list containing all the Fibonacci numbers in order:

```
{-@ lazy fib @-}
fib = 0 :> 1 :> zipWith (+) fib (tail fib)
```

These definitions yield, as we described, infinite objects. If we were to evaluate them (e.g by printing their result), we would get an infinitely running program. Of course such a program can sometimes be useful. Servers for one can be thought of as programs that never terminate, always waiting for requests and returning answers. We can also interact with such definitions by observing a finite part of them, which is possible because of laziness! If we try to evaluate the first `n` elements of the Fibonacci list (using `stake` which we define below) the `fib` object will only be evaluated at a depth of `n` elements; the rest will remain unevaluated, unless we specifically ask for it through some computation.

As we previously described, Liquid Haskell demands that functions are terminating and therefore our elegant `fib` definition would normally be rejected. In order to get it accepted we have to explicitly mark it as a non-terminating function, which is the purpose of the annotation `lazy fib`.

However, there are definitions and expressions that are infinite in some sense but are undesirable. Consider, for example, the function `loop`:

```
loop = loop
```

This function is perfectly valid Haskell code, but it describes a divergent computation. `loop` is not useful in most normal scenarios. Trying to evaluate will yield an infinite computation that does not produce any output, in contrast to `fib`, the usefulness of which we described previously. However, if we added a `lazy loop` annotation, Liquid Haskell would naively accept the definition of `loop` just as it accepts the definition of `fib`! There is no way to account for well-behavedness of infinite definitions.

When dealing with laziness, we can attain a part of a possibly infinite result in finite time, since we can refer to and compute part of an infinite object without triggering an infinite computation. Specifically for streams we can define `stake`:

```
stake :: Nat → Stream a → [a]
stake 0 _         = []
stake n (x :> xs) = x: stake n xs
```

We can then express the well-behavedness of a stream definition if for any `n` we can compute `stake n` of this stream in finite steps. This kind of well-behavedness is called *productivity*. Below we define `srepeat` which, despite being infinite it is well-behaved in this sense:

```
srepeat :: Stream a
srepeat x = x :> srepeat x
```

In contrast to `srepeat`, the function `loop`, which we defined previously, is not productive. The problem with it is that not only is it infinite, but also there is not a part of its result which we can attain in finite time: `stake n loop` for any `n` yields an infinite computation.

In chapter 2 we describe how we can encode productivity in Liquid Haskell and write corecursive definitions so that those that are well-defined (e.g., `srepeat`) are accepted and ill-defined ones (e.g., `loop`) are rejected by Liquid Haskell. As a motivation we add here the definition of the Fibonacci sequence as a stream as encoded in chapter 2:

```
zipWith :: i:Nat
        → (a → b → c)
        → StreamG a i → StreamG b i
        → StreamG c i
zipWith i f xs ys = cons i
                     (\j → f (shead j xs) (shead j ys))
                     (\j → zipWith j f (stail j xs) (stail j ys))

fib :: i:Nat → StreamG Int i
```

```
fib i = cons i (const 0)
            $ \j → cons j (const 1)
            $ \k → zipWith k (fib k) (stail k (fib j))

const x _ = x
```

1.3 Coinduction

Since we are dealing with infinite definitions it is useful to start venturing into the world of *coinduction*, which is a notion dual to structural induction.

Structural induction deals with well-founded datatypes, meaning types that have base cases (such as lists). Such types contain objects that can be constructed with finite steps by starting from the base cases and using the other available rules.

Coinduction on the other hand deals with datatypes which need not be well-founded. Coinductive objects cannot be constructed in the inductive sense, since base cases are not available. We can only define a coinductive object if we have such an object to begin with. If we observe the definitions of `fib` and `repeat` previously, we can see that both use self-references in order to complete their definitions.

Coinduction is better understood as *observation* or *destruction* instead of construction: Objects are defined by how they can be observed, or, equally, by the parts to which they can be destructed. A (non-empty) list for example can be decomposed to its `head` (the first element) and its `tail` which is also a coinductive object that can be decomposed. In fact in Agda we can define coinductive objects by using co-patterns [Abel13] which makes this definition by observation explicit.

While lists are very relevant and well-known in the Haskell world, coinduction is better illustrated when acting on streams as we mentioned in section 1.2:

```
data Stream = a :> Stream a
```

We can easily see that streams are non-well-founded and therefore they can only be interpreted coinductively. The similarity of streams to lists allows us to easily adapt many functions that are defined for lists to work on streams. For example we can define `smap` following the previous `map` definition:

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

Because of non-termination we need to annotate `smap` as lazy in order for Liquid Haskell to accept it. In fact we also have to pass a special flag in order for Liquid Haskell and the underlying SMT solver to allow the definition of `Stream` in the first place. The `"--no-adt"` flag that we use for this purpose, tells Liquid Haskell not to map Haskell data types to SMT data types, which would reject non-well-founded types.

Now if we also adapt the `mapFusion` proof for streams we arrive at:

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

This proof definition is accepted by Liquid Haskell – provided we have already added the "--no-adt" flag. The proof is accepted without being terminating. This makes us question the validity of the proof. Indeed, Liquid Haskell becomes inconsistent with this setup!

The problem is that by not having the restriction of termination we can easily prove any property, even false ones, by simply invoking the inductive hypothesis. Such an example is `falseStream` defined below:

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

In chapter 3 we show how we can in fact reason about coinductive properties in Liquid Haskell using two techniques: the indexed approach (section 3.1) and the constructive approach (section 3.2). For example, with the constructive approach we can obtain the `mapFusion` proof as follows:

```
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
```

1.4 Contribution

To sum up, in this work we encode coinductive techniques in Liquid Haskell. We first present a technique for modifying an inductive verifier to ensure productivity of corecursive definitions (chapter 2). Second, we present how an inductive verifier could be extended to support coinductive reasoning (chapter 3). These extensions could, in the future, be applied both in Liquid Haskell and in GHC's dependent types. Apart from the specific methods used, we also highlight the gap that an inductive verifier has to cover in order to verify coinductive properties. We finally (chapter 4) provide a number of examples in order to illustrate the use of the techniques described.

Our code can be found in its entirety in github.com/lykmast/co-liquid. Also, a large part of this work has been accepted for presentation at this year's Haskell Symposium [Mast22].

Chapter 2

Productivity of Corecursive Definitions

In chapter 1 we discussed how and why Liquid Haskell ensures termination of definitions. We also described *corecursive definitions* which can be non-terminating.

In this chapter we encode corecursive definitions in Haskell and use Liquid Haskell to typecheck their well-behavedness – despite non-termination:

- In section 2.1 we make a first attempt at expressing productivity using a syntactic check.
- In section 2.2 we encode in Liquid Haskell a better approach using depths.

2.1 Syntactic guardedness

Observing the difference between `srepeat` and `loop` we can start formulating rules about how a corecursive definition should look like. We see that `stake` pattern-matches on a stream cons. In contrast to `loop`, `srepeat` is expressed as a cons. However this is not a sufficient condition for a productive corecursive function, as exemplified by `badCons`:

```
badCons = shead badCons :> badCons
```

The problem here is that in order to calculate the first element of `badCons` we first need to calculate `shead badCons`. This tautology leads to an infinite computation. We can generalize this problem to a bad self-call: `shead badCons` is obviously problematic in this position.

A rule we can come up with to exclude `badCons` is that, besides using the constructor `(:)`, a valid definition needs to directly self-call (so `head badCons` is not accepted in any position). Both `loop` and `badCons` are now rejected while `srepeat` is accepted. This rule is the *guardedness condition* which is widely used (e.g. Coq [Bert06]) and it is a sufficient condition of productive corecursive definitions.

Unfortunately, there is a large class of definitions that are productive but not syntactically guarded. Let's come back to our Fibonacci function from chapter 1. To define `fib` we first need `zipWith`:

```
zipWith :: (a → b → c) → Stream a → Stream b → Stream c
zipWith f (x :> xs) (y :> ys) = f x y :> zipWith f xs ys
```

We can easily see that `zipWith` is guarded: the only self-call is directly applied inside the `(:)` constructor. Now to define `fib`:

```
fib = 0 : 1 : zipWith (+) fib (stail fib)
```

It is obvious that this definition is not guarded in the sense we described above, since the self-call of `fib` is inside `zipWith`. If we implemented this guardedness check we would not be able to get `fib` to typecheck. However we strongly suspect that `fib` is productive – for one, because of its popularity in the Haskell community. If we want to be able to accept `fib` we need a more complex system that will be able to keep track of productivity during function composition.

2.2 Productivity with depths

In order to implement a better productivity check we should explore what can go wrong in a definition like `fib` in terms of productivity. To that end we define `zipWith'`:

```
zipWith' f (x :> _ :> xs) (y :> _ :> ys) = f x y :> zipWith f xs ys
```

The definition of `zipWith'` is similar to `zipWith`, but its result is different since it discards every other element of the initial streams. We now define `fib'` that is identical to `fib` except for the fact that it uses `zipWith'` instead of `zipWith`:

```
fib' =   0 :> 1 :> zipWith' (+) fib' (stail fib')
     === 0 :> 1 :> 0 + 1 :> zipWith' (+) (stail (stail fib'))
                                         (stail (stail (stail fib')))
     === 0 :> 1 :> 1 :> 1 + shead (stail (stail (stail fib'))) :> ...
```

We can see that `fib'` is not productive: the term `shead (stail (stail (stail fib')))` here cannot be computed because its computation depends on itself. We can reframe this by saying that `fib'` attempts to define its fourth element with a dependency on the fourth element, which will of course get stuck since it is not yet calculated. Only the first three elements should be present in the expression that calculates the fourth one, since they are the only ones that have already been calculated.

What we need is a system that allows us to express which elements of the stream have already been calculated and so can be used in an expression that calculates another element. To that end we use a natural number that keeps track of the number of elements from the start of the stream that can be accessed without getting stuck on a non-computable element. We call this number the *depth* of the stream. In order to actually keep track of the depth of a stream we need to alter the definitions of `cons`, `shead` and `stail` as viewed in fig. 2.1. The actual value of a depth variable in the runtime does not matter, because depths are only relevant to Liquid Haskell, so we instantiate every depth with 0.

Depths are different than sizes: Size is used to count the number of elements that a structure has. This metric is useful for structures like lists that are terminating, but for streams it is nonsensical since streams have always infinite size. Depth, in contrast to size, measures the number of elements from the start of the stream that can be accessed without getting stuck on a non-computable element like `fib'`. According to this definition `fib'` can be given a maximum depth of 3, since the fourth element cannot be computed.

Productivity for a stream can be expressed by proving that this stream can be given any depth, i.e., any of its elements can be accessed and computed. To actually prove this, we use the altered infrastructure of fig. 2.1 to enable a stream definition to also serve as a proof of its productivity.

Namely, `cons` signature dictates that a stream can be accessed at depth `i`, if for every `j<i` we can produce a stream of depth `>= j` and a single element. Of course this follows from the meaning we gave to depth: if a stream's first `j` elements can be calculated and we provide a definition for another element that will serve as the head of the stream, then we can obtain a stream of which the first `i > j` elements can be safely accessed.

Respectively, if a stream has depth `> j` we can produce a stream with depth `>= j` using `stail`. `shead` works in the same way but does not assert a depth property for its result as it is a single element. In `shead` and `stail` the `j` argument serves also as a witness that we can access the head and tail of the stream: The reasoning is that since the stream has a depth `i` with the property `i > j`, then `i` is `>= 1` (since all depths are natural numbers) and both head and tail can be safely computed.

Note that in these depth signatures we use inequalities, such as `i>j` or `>=j`, instead of specifying the exact depth of the stream. We do this because it gives greater flexibility to definitions. Basically it allows us to use strong induction inside the productivity proofs as we highlight later.

Finally we use the measure `inf` to signify a stream that can be accessed at infinite depth i.e., a stream which has a productive definition. This is expressed in `toInf` (section 2.2) which allows us

```
measure depth :: Stream a → Nat
measure inf  :: Stream a → Bool

type StreamS a S = {v:Stream a | depth v = S}
type StreamG a S = {v:Stream a | depth v >= S || inf v}
type StreamI a I = {v:Stream a | inf v}

type LT I = {j:Nat | j < I}

assume cons :: i:Nat
            → (LT i → a) → (j:LT i → StreamG a j)
            → StreamS a i
cons _ fx fxs = fx 0 :> fxs 0

shead :: j:Nat → {xs:Stream a | depth xs > j || inf xs} → a
shead j (x :> xs) = x

assume stail :: j:Nat
             → {xs:Stream a | depth xs > j || inf xs}
             → {v:StreamS a j | inf xs ==> inf v}
stail i (x :> xs) = xs
```

Figure 2.1: Infrastructure of `Stream`

to say that a stream is considered to have infinite depth when it can be given any depth `i` which is equivalent to having a proof of productivity. We also use `inf` in the signatures of fig. 2.1 to express that any access is allowed to streams that are already proven productive. Note that inside its own definition a stream is not yet proven productive and so cannot be `inf`.

A note on measures: In Liquid Haskell we can use the measure annotation to introduce uninterpreted functions. Liquid Haskell initially knows nothing about these functions but their type signature. We can introduce facts about these functions to Liquid Haskell by using them in axiomatized signatures. For example, `depth` is such a function that express the depth of a stream as a natural number. Liquid Haskell initially knows only that `depth` is a function that accepts a stream and returns a natural numbers. Additional facts about `depth` are introduced to Liquid Haskell through the assumed signatures of `cons` and `stail` as described above.

With the help of this infrastructure we can now write `fib` and other interesting corecursive functions in a way that allows Liquid Haskell to verify that they are productive. These definitions differ from the original (i.e., the ones that we would define in plain Haskell) because in order to keep track of depths we have to use lambda expressions to instantiate the depth with the property that we want (e.g. smaller than the depth of a stream).

```
zipWith :: i:Nat
        → (a → b → c)
        → StreamG a i → StreamG b i
        → StreamG c i
zipWith i f xs ys = cons i
                    (\j → f (shead j xs) (shead j ys))
                    (\j → zipWith j f (stail j xs) (stail j ys))
```

```
fib :: i:Nat → StreamG a i
fib i = cons i (const 0)
            $ \j → cons j (const 1)
            $ \k → zipWith k (fib k) (stail k (fib j))

const x _ = x
```

Explaining the depth-annotated versions. The signature of `zipWith` expresses the fact that `zipWith` takes as arguments two streams defined at depth `i` or higher and returns a stream defined at the same depth `i`. In other words if both streams can be accessed at a specific depth, so can the result of an element-wise operation on them. This property is proved in the body of `zipWith` and checked by Liquid Haskell through the type signatures of `shead`, `stail` and `cons`.

More concretely, `cons` produces a stream of depth `i`, since it is given `i` as its depth argument, provided that the second and third argument produce a single element and a stream of depth `>= j` respectively for every `j < i`. The head of the stream is produced by operating on the heads of the two argument streams, which we are able to do using `j` as a witness on the fact that we are allowed to access them. The tail of the stream can be proven to have depth `j` by applying the inductive hypothesis on `j` and the tails of the two streams – which we are again allowed to access because of `j`.

Productivity proofs are inductive. We remind that in order for the proof to be inductive we also need Liquid Haskell's termination check which in these case is satisfied since `zipWith` recurses on `j < i`. Proofs of productivity can be viewed as – and indeed are – inductive proofs. For `i = 0` there are no `j < i` so the proof is trivial, as is the fact that any stream has depth at least 0. Then for every other `i` we show that the stream has depth `i` by using strong induction, i.e., using the inductive hypothesis on any `j < i`.

Moving on to `fib`, we define with constants the first two elements, which leaves us to prove that the tail of the tail of `fib i` has depth `k` for every `k < j < i`. To prove this we first invoke the inductive hypothesis on `k` and `j` –which proves that `fib j` and `fib k` have depths `>= j` and `>= k` respectively. This allows us to invoke `stail k (fib j)` to prove that this expression has depth `>= k` and finally using `zipWith k` we prove that the whole tail-of-tail expression has depth `k`.

We can also disallow `fib'`. If we try to add signatures through our newly defined system to `zipWith'` and `fib'` we discover that we don't have a way to do so without Liquid Haskell throwing an error:

```
zipWith' :: i:Nat
         → (a → b → c)
         → StreamI a → StreamI a
         → StreamG a i
zipWith' i f xs ys = cons i
                       (\_ → f (ihead xs) (ihead ys))
                       (\j → zipWith j f (itail (itail xs)
                                         (itail (itail ys))
```

In order to get `zipWith'` to typecheck we need to demand from the arguments to be of infinite depth i.e., fully defined, because otherwise we will not able to access two depths below `i` with only one nesting of `cons`. Because of this infinite depth we can use the corresponding destructors which do not need a depth witness (section 2.2).
The signature of `zipWith'` prevents us from defining `fib'` using `zipWith'` since we can only invoke `zipWith'` with full stream definitions. Even if we try to use `toInf` inside the definition we will get an error.

```
assume toInf :: (i:Nat → StreamG a i) → StreamI a
toInf f = f 0

ihead :: StreamI a → a
ihead xs = shead 0 xs

itail :: StreamI a → StreamI a
itail xs = stail 0 xs
```

Figure 2.2: Functions for fully defined streams

```
fib' :: i:Nat → StreamG Int i
fib' i = cons i (const 0)
              $ \j → cons j (const 1)
              $ \k → zipWith' k (toInf fib') (itail (toInf fib'))
              -- !!ERROR!!
```

The problem here is that `toInf` attempts to invoke `fib'` for all natural numbers. This is not a valid inductive hypothesis invocation since `toInf` also needs depths that are greater than `i`. Liquid Haskell points this out by throwing a termination error on the self-call of `fib'`.

Generalization. Productivity can also be applied to other coinductive types than streams (e.g. possibly infinite tree structures). The technique we use here can easily be applied to express the productivity of such types: we just have to apply the depth refinements that we applied to `shead`, `stail` and `cons` to the corresponding destructors and constructors of these types. In chapter 4 we include examples of such corecursive definitions. A more formal definition of constructors and destructors that use depths can be found in [Abel10, Abel16].

Chapter 3

Coinductive Proofs

In chapter 1 we described how coinductive properties can not be seamlessly proven in Liquid Haskell: The well-foundedness check disallows us even defining coinductive types and deactivating it introduces inconsistency (e.g., `falseStream`).

In this chapter we present two methods we used for adapting Liquid Haskell's infrastructure to prove coinductive properties:

- In section 3.1 we present the *indexed coinduction* technique in which we index the coinductive predicates and encode coinductive proofs by induction on the index.
- In section 3.2 we present the constructive coinduction technique that again uses indices, but to ensure guardedness in constructive proofs that are encoded in Liquid Haskell using refinements over GADTs.

3.1 Indexed Coinduction

In this section we encode indexed coinduction, that let us consistently prove properties about coinductive predicates. First (§3.1.1), we index coinductive properties with a natural number, to eliminate inconsistent proofs. Next (§3.1.2), we define indexed predicates that trivially satisfy base cases. Finally (§3.1.3), we conclude by noticing that indexed equality bisimulates stream equality.

3.1.1 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` 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.

3.1.2 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`.

```
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 §1.1:

```
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 1.1, 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.

3.1.3 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 §3.1.2 we prove indexed equality. Here, we explain how these two forms of the theorem's statement connect.

[Bird88] 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:

$$x = y \quad \Leftrightarrow \quad \forall k.\ \texttt{take}\ k\ x\ = \texttt{take}\ k\ y$$

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.

3.2 Constructive Coinduction

Constructive coinduction is our second mechanization technique, where proofs are constructed using Haskell's (refined) GADTs [Xi03, Peyt06]. First (§3.2.1) we define `EqC`, the GADT that constructs observational equality on streams. Next (§3.2.2), we use `EqC` to prove our running theorem. Finally (§3.2.3), via the take lemma, we prove that `EqC` approximates stream equality.

3.2.1 Constructive Equality

As a first (failing) attempt to define constructive stream equality, we define Coq's textbook [Chli13] coinductive stream equality, using Liquid Haskell's data propositions [Bork22] 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 [Abel10]. 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 §3.1.1. Indexing permits coinductive reasoning using inductive verification.

3.2.2 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!

3.2.3 Again, Did we Prove Equality?

Finally, as in §3.1.3, 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 §3.1.3, 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.

Chapter 4

Examples

In this chapter we present some code examples of the techniques described in this work.

- In section 4.1 we present examples of corecursive definitions annotated with depths in order to verify their productivity.
- In section 4.2 we present examples of coinductive proofs using the indexed (§3.1) and constructive (§3.2) techniques.

4.1 Corecursive definition examples

Example 1: Merge Evens Odds In section 4.2 we prove the property `mergeEvenOdd` which uses the functions `merge`, `evens` and `odds`. Here we prove the productivity of those functions:

```
{-@ odds :: i:Nat → StreamI _ → StreamG _ i @-}
odds :: Int → Stream a → Stream a
odds i xs = cons i (const $ ihead xs) (\j → odds j $ itail . itail $ xs)

{-@ evens :: i:Nat → StreamI _ → StreamG _ i @-}
evens :: Int → Stream a → Stream a
evens i = odds i . itail


{-@ merge :: i:Nat → StreamG _ i → StreamG _ i → StreamG _ i @-}
merge :: Int → Stream a → Stream a → Stream a
merge i xs ys = cons i (\j → shead j xs) (\j → merge j ys (stail j xs))
```

The functions `odds` and `evens` take a fully defined stream as an argument. This is because they consume twice the elements that they produce and therefore cannot be part of a self-calling definition that can be proven productive with our technique. This allows us to use the destructors `ihead` and `itail` and makes productivity trivial to prove.

`merge` on the other hand takes two arguments that, for some `i` have depth `i` (meaning they are defined at least for depth `i`) and returns a stream that claims to have depth `i`. The self-call to `merge j` satisfies both the termination checker and the signature of `cons` (since `j < i` and `merge j` has depth `j`). We cannot easily give `merge` a more accurate type (which expresses that it returns two elements after one access to each stream) using this technique. However, as in the case of `odds` and `evens`, these annotations are expressive enough for most sensible definitions.

Notice that all of `merge`, `evens` and `odds` can be also proven productive with syntactic guardedness, since the self call is nested under `cons`.

Example 2: Paperfolds Another interesting corecursive definition that we encountered in [Clou15] is `paperfolds`, which represents the regular paper-folding sequence (A014577) as a stream. The original definition of `paperfolds` is:

```
paperfolds = merge toggle paperfolds

toggle = True :> False :> toggle
```

Below we rewrite `paperfolds` and `toggle` with depths in order to prove their productivity:

```
{-@ toggle :: i:Nat → StreamG _ i @-}
toggle :: Num a ⇒ Int → Stream a
toggle i = cons i (const 0) $ \j →
           cons j (const 1) toggle


{-@ paperfolds :: i:Nat → StreamG _ i @-}
paperfolds :: Num a ⇒  Int → Stream a
paperfolds i = cons i (\j → ihead (toInf toggle)) $
                      \j → merge j (paperfolds j) (itail (toInf toggle))
```

In the code above, one unfolding of merge was necessary to prove termination of `paperfolds`: without unfolding `merge` we would have to self-call `paperfolds i` which would not pass the termination check. We also see that `toggle` can be used as a complete definition (using `toInf`), since `paperfolds` is not part of its definition. Finally, while `toggle` could be proven productive with syntactic guardedness, `paperfolds` cannot, as its self-call is inside `merge` rather than being directly nested in `cons`.

Example 3: Mixed recursive and corecursive calls In corecursive definitions there can be branches that are non-productive. This is allowed as long as we can prove that a productive branch will be always reached. A simple example of such a definition is `fivesUp` (found in [Lein14]), which produces a stream with all multiples of five greater than its argument:

```
{-@ fivesUp :: i:Nat → n:_
            → StreamG {v:_ | v >= n} i / [i, fivesUpTerm n]
@-}
fivesUp :: Int -‘> Int → Stream Int
fivesUp i n | n ‘mod‘ 5 == 0
            = cons i (const n) $ \j → fivesUp j (n+1)
            | otherwise = fivesUp i (n+1)

{-@ inline fivesUpTerm @-}
fivesUpTerm :: Int → Int
fivesUpTerm n = 4 - ((n-1) ‘mod‘ 5)
```

The first branch is productive as it has a valid self-call inside `cons` with `j < i` (guarded by coinductive constructor). The second branch is not directly productive, but we can determine that it will eventually return to the `n ‘mod‘ 5 == 0‘` branch, as `fivesUpTerm n` is decreasing for the `otherwise` branch. In order to satisfy the termination checker we have added a lexicographic termination measure (`[i, fivesUpTerm n]`) which expresses that when the branch is not productive (i.e., `i` does not decrease), it is eventually escaped because of another termination metric (i.e., `fivesUpTerm n`).

Example 4: Breadth first labeled infinite tree Jones and Gibbons [Jone93] have described a functional, linear-time, breadth-first tree labeling algorithm. Abel and Pientka [Abel16] modified it for infinite trees and, using copatterns and sized types, prove its productivity.

```
bf :: Stream a → Tree a
```

```
bf vs = t where (t, vss) = bfs (vs :> vss)

bfs :: Stream (Stream a) → (Tree a, Stream (Stream a))
bfs ((v :> vs) :> vss) = (node v l r, vs :> vss'')
      where (l, vss')  = bfs vss
            (r, vss'') = bfs vss'
```

The function `bf` takes as an input a stream of labels and produces the infinite binary tree that is labeled by the stream in a breadth-first order. `bfs` takes a stream of streams of labels and produces a pair of a tree and a stream of streams. The role of `bfs` is to help define `bf` in a cyclical way. In `bf` it is apparent that the input `v:>vss` of `bfs` is partly constructed by its output.

The algorithm is fairly complex and its productivity is not evident by a simple read-through. For that reason, and also to illustrate the use of the technique in more complex datatypes, we add here the proof of its productivity following [Abel16].

We first define the `Tree` datatype and along with the depth-tracking constructor and destructors:

```
data Tree a = Node {_label :: a, _left :: Tree a, _right :: Tree a}

{-@ measure tdepth :: Tree a → Nat @-}
{-@ measure tinf  :: Tree a → Bool @-}

{-@ label :: j:Nat → {t:_ | tdepth t > j} → _ @-}
label :: Int → Tree a → a
label _ = _label

{-@ assume left :: j:Nat → {t:_ | tdepth t > j || tinf t}
                → {l:_| tdepth l = j && (tinf t ==> tinf l)}
@-}
left  :: Int → Tree a → Tree a
left _ = _left

{-@ assume right :: j:Nat → {t:_ | tdepth t > j || tinf t}
                 → {r:_| tdepth r = j && (tinf t ==> tinf r)}
@-}
right :: Int → Tree a → Tree a
right _ = _right

{-@ assume node :: i:Nat → ({j:Nat|j<i} → _)
                → ({j:Nat|j<i} → TreeG _ j)
                → ({j:Nat|j<i} → TreeG _ j)
                → {v:_ | tdepth v = i}
@-}
node :: Int → (Int → a) → (Int → Tree a) → (Int → Tree a) → Tree a
node i flb fl fr = Node (flb 0) (fl 0) (fr 0)
```

We also need to define our stream of streams and a `Result` datatype which will take the place of the tuple that carries the result of `bfs`. We mostly need it to relate the depth of the resulting tree with the depth of the resulting stream of streams.

```
type SS a = Stream (Stream a)
{-@ type SS a S = StreamG (StreamI a) S @-}
```

```
data Result a = Res {_tree:: Tree a, _rest:: SS a}

{-@ measure rdepth :: Result a → Nat @-}
{-@ measure rinf  :: Result a → Bool @-}

{-@ type ResultI a I = {r:Result a | rdepth r = I} @-}

{-@ assume  res :: i:Nat → TreeG _ i
                → SS _ i → ResultI _ i @-}
res :: Int → Tree a → SS a → Result a
res _ t ss = Res t ss


{-@ assume tree :: r:_ → TreeG _ {rdepth r} @-}
tree = _tree
{-@ assume rest :: r:_ → SS   _ {rdepth r} @-}
rest = _rest
```

Now we can proceed to define `bfs`:

```
{-@ bfs :: i:Nat → SS _ i → ResultI _ i @-}
bfs i ss = res i (node i v (\j → tree $ p1 j)
                         $  \j → tree $ p2 j)
               $  cons i vs $ \j → rest (p2 j)

           where p1  = \j → bfs j (vss j)
                 p2  = \j → bfs j $ rest $ p1 j
                 vss = \j → stail j ss
                 v   = \j → ihead (shead j ss)
                 vs  = \j → itail (shead j ss)
```

Finally we can define `bf` as follows:

```
{-@ bf :: i:Nat → StreamI _ → TreeG _ i @-}
bf i = tree . bfp i
  where {-@ bfp :: i:Nat → StreamI a → ResultI a i @-}
        bfp i vs = bfs i $ cons i (const vs) $ \j → rest (bfp j vs)
```

In [Abel16] the above definition of `bfp` is not possible. Sizes are only available when copatterns are being used, so `bfp` does not have a `j` available because of `cons`. We do not share that problem since we have a different annotation of `cons` that provides us with a `j<i`. However we proceed to translate the two implementations of `bfp` that Abel and Pientka provide. Note that in order to define `bfp` with copatterns there has to be one unfolding of `bfs` which makes the code a little bulkier:

```
{-@ bfp' :: i:Nat → StreamI _ → ResultI _ i @-}
bfp' :: Int → Stream a → Result a
bfp' i vs = res i (node i  (\j → label j $ t j)
                           (\j → left  j $ t j)
                           (\j → right j $ t j))

                $ cons i (\j → shead    j $ r j)
                       $  \j → stail    j $ r j
  where p j = bfs (j+1) $ cons (j+1)
```

```
                              (const vs)
                              (\_ → rest $ bfp' j vs)
        t j = tree $ p j
        r j = rest $ p j
```

In order to embelish `bfp`, the authors also propose the use of `fixR` to hide away the unfolding:

```
{-@ fixR :: i:Nat
         → (j:Nat → ResultI _ j → ResultI _ {j+1})
         → ResultI _ i
@-}
fixR :: Int → (Int → Result a → Result a) → Result a
fixR i f =  res i (node i  (\j → label j $ t j)
                           (\j → left  j $ t j)
                           (\j → right j $ t j))

                $ cons i (\j → shead    j $ r j)
                       $  \j → stail    j $ r j
  where p j = f j (fixR j f)
        t j = tree $ p j
        r j = rest $ p j

{-@ bfp'' :: i:Nat → StreamI _ → ResultI _ i @-}
bfp'' :: Int → Stream a → Result a
bfp'' i vs = fixR i f
  where f j r = bfs (j+1) $ cons (j+1)
                                 (const vs)
                                 (\_ → rest r)
```

4.2 Examples of coinductive proofs

4.2.1 Equal Streams

The first 4 properties prove equality on streams. Property 0 was detailed in §3.1 and §3.2. 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}
```

- Indexed proof of mergeEvenOdd:

```
{-@ mergeEvenOddK :: xs:_ → k: Nat
                  → {eqK k (merge (odds xs) (evens xs)) xs}
@-}
mergeEvenOddK s 0
  =   eqK 0 (merge (odds s) (evens s)) s
  *** QED
mergeEvenOddK xxs@(x :> xs) k
  =
  merge (odds xxs) (evens xxs)
  === merge (x :> odds (stail xs))
            ((odds . stail) xxs)
  === merge (x :> (odds . stail) xs) (odds xs)
  === x :> merge (odds xs) (evens xs)
    ? mergeEvenOddK xs (k-1)
  =#= k # x :> xs
  *** QED

  mergeEvenOdd xs = approx (merge (odds xs) (evens xs)) xs (
 mergeEvenOddK xs)
```

- Constructive proof of mergeEvenOdd:

```
{-@ mergeEvenOddI :: xs:Stream a
                  → i:Nat
                  → Prop (Bisimilar i  (merge (odds xs) (evens xs)) xs)
@-}
mergeEvenOddI xxs@(x :> xs) i
  = Bisim i x (merge (odds xs) (evens xs)) xs (mergeEvenOddI xs)
     ? ( merge (odds xxs) (evens xxs)
    === merge (x :> odds (stail xs)) (odds (stail xxs))
    === x :> merge (odds (stail xxs)) (odds (stail xs))
    === x :> merge (odds xs) (evens xs)
    *** QED)

  mergeEvenOdd xs = eqCLemma (merge (odds xs) (evens xs)) xs (
 mergeEvenOddI xs)
```

Properties 2-3: Thue-Morse sequence These two properties are inspired by [Rosu09] 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 4.1. 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 `ff` that takes as input a stream and replaces each of its values `x` with `x`, followed by `x`'s negation. Property 2, `morseFix`, proves that `f` is the fixpoint of the `morse`

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

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

not True  = False
not False = True

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

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

Figure 4.1: Properties 2 and 3 on Morse signals.

sequence. In order to prove it we use the `morseMerge` property which proves that `f xs` is equal to `merge xs (smap not xs)`, from which we can obtain `morseFix` by:

```
  {-@ morseFix :: {ff morse = morse} @-}
morseFix
  =   ff morse
  === shead morse :> not (shead morse) :> ff (stail morse)
    ? morseMerge (stail morse)
  === False :> True :> merge (stail morse) (smap not (stail morse))
  === morse
  *** QED
```

Property 3, `fNotCommute`, proves that `f` and `(smap not)` commute.

- Indexed proofs of `morseMerge` and `fNotCommute`:

```
{-@ morseMergeK :: xs:_ → k:Nat
                   → {eqK k (ff xs) (merge xs (smap not xs))}
@-}
morseMergeK xs 0 = eqK 0 (ff xs) (merge xs (smap not xs)) *** QED
morseMergeK xxs@(x :> xs) 1
  =   ff xxs
  === x :> not x :> ff xs
    ? (eqK 0 (not x :> ff xs) (not x :> merge xs (smap not xs)) *** QED)
  =#= 1 #
      x :> not x :> merge xs (smap not xs)
  === x :> merge (not x :> smap not xs) xs
  === merge xxs (smap not xxs)
  *** QED
morseMergeK xxs@(x :> xs) k
  =   ff xxs
  === x :> (
```

```
                not x :> ff xs
              ? morseMergeK xs (k-2)
            =#= k-1 #
                not x :> merge xs (smap not xs)
            )
  =#= k #
      x :> not x :> merge xs (smap not xs)
  === x :> merge (not x :> smap not xs) xs
  === merge xxs (smap not xxs)
  *** QED

{-@ morseMerge :: xs:Stream Bool → {ff xs = merge xs (smap not xs)} @-}
morseMerge xs = approx (ff xs) (merge xs (smap not xs)) (morseMergeK xs)

{-@ fNotCommuteK :: xs:_ → k:Nat → {eqK k (smap not (ff xs))
                                            (ff (smap not xs))}
@-}
fNotCommuteK xs 0 = eqK 0 (smap not (ff xs)) (ff (smap not xs)) *** QED
fNotCommuteK xxs@(x :> xs) 1
  =   smap not (ff xxs)
  === smap not (x :> not x :> ff xs)
  === not x :> smap not (not x :> ff xs)
    ? (eqK 0 (smap not (not x :> ff xs))
             (not (not x) :> ff (smap not xs))
       *** QED
      )
  =#= 1 #
      not x :> not (not x) :> ff (smap not xs)
  === ff (not x :> smap not xs)
  === ff (smap not xxs)
  *** QED
fNotCommuteK xxs@(x :> xs) k | k > 1
  =   smap not (ff xxs)
  === smap not (x :> not x :> ff xs)
  === not x :> smap not (not x :> ff xs)
  === not x :> (   not (not x) :> smap not (ff xs)
               ?   fNotCommuteK xs (k-2)
               =#= k-1 #
                    not (not x) :> ff (smap not xs)
               )
  =#= k #
      not x :> not (not x) :> ff (smap not xs)
  === ff (not x :> smap not xs)
  === ff (smap not xxs)
  *** QED

 fNotCommute xs = eqCLemma (smap not (ff xs)) (ff (smap not xs)) (
  fNotCommuteI xs)
```

- Constructive proofs of `morseFix` and `fNotCommute`:

```
{-@
morseMergeI :: xs:_ → i:Nat
              → Prop (Bisimilar i (ff xs) (merge xs (smap not xs)))
@-}
morseMergeI xxs@(x :> xs) i
  =       Bisim i x (stail (ff xxs)) (stail (merge xxs (smap not xxs)))
  $ \j → Bisim j (not x) (ff xs) (merge xs (smap not xs))
                (morseMergeI xs) ? expandL ? expandR
  where
    expandL
       =  stail (merge xxs (smap not xxs))
      === stail (x :> merge (smap not xxs) xs)
      === merge (not x :> smap not xs) xs
      === not x :> merge xs (smap not xs)
      *** QED
    expandR
       =  stail (ff xxs)
      === stail (x :> not x :> ff xs)
      === not x :> ff xs
      *** QED

{-@ morseMerge :: xs:Stream Bool → {ff xs = merge xs (smap not xs)} @-}
morseMerge xs = eqCLemma (ff xs) (merge xs (smap not xs)) (morseMergeI
  xs)

{-@
fNotCommuteI :: xs:_ → i:Nat
              → Prop (Bisimilar i (smap not (ff xs)) (ff (smap not xs)))
@-}
fNotCommuteI xxs@(x :> xs) i
  =       Bisim i (not x) tlLhs tlRhs
  $ \j → Bisim j (not (not x)) (smap not (ff xs))
                  (ff (smap not xs)) (fNotCommuteI xs)
  where
    lhs
      =  smap not (ff xxs)
     === smap not (x :> not x :> ff xs)
     === not x :> smap not (not x :> ff xs)
     === not x :> not (not x) :> smap not (ff xs)
    rhs
      =  ff (smap not xxs)
     === ff (not x :> smap not xs)
     === not x :> not (not x) :> ff (smap not xs)
    tlRhs
      =  stail rhs
     === not (not x) :> ff (smap not xs)
    tlLhs
      =  stail lhs
     === not (not x) :> smap not (ff xs)
```

```
fNotCommute xs = eqCLemma (smap not (ff xs)) (ff (smap not xs)) (
 fNotCommuteI xs)
```

4.2.2 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 §3.2, 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}

assume trivialLemmaC :: s:Stream a
                      → (k:Nat → Prop (Trivial k s))
                      → {trivial s}
```

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

- Indexed proof of `trivialAll`:

```
trivialAll xs = trivialLemmaC xs (trivialAllK xs)
where {-@ trivialAllK :: s:_ → k: Nat → {trueStreamK k s} @-}
      trivialAllK s 0 = trueStreamK 0 s *** QED
      trivialAllK (s :> ss) k
        =   trueStreamK k (s:>ss)
        === trueStreamK (k-1) ss
        ? trivialAllK ss (k-1)
        *** QED
```

- Constructive proof of `trivialAll`:

```
trivialAll xs = trivialLemmaC xs (trivialAllI xs)
  where {-@ trivialAllI :: xs:_ → i:Nat → Prop (TrueStream i xs) @-}
        trivialAllI (x :> xs) i = TrueS i x xs (trivialAllI xs)
```

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.

- Indexed proof of `mergeSelfDup`:

```
{-@ mergeSelfDupK :: xs:_ → k:Nat → {dupK k (merge xs xs)} @-}
mergeSelfDupK xs 0 = dupK 0 (merge xs xs) *** QED
mergeSelfDupK xxs@(x :> xs) k =
  dupK k (    merge xxs xxs
         === x :> merge xxs xs
         === x :> x :> merge xs xs
         )
  === dupK (k-1) (merge xs xs)
    ? mergeSelfDupK xs (k-1)
  *** QED
```

```
mergeSelfDup xs = dupLemma (merge xs xs) (mergeSelfDupK xs)
```

- Constructive proof of `mergeSelfDup`:

```
{-@ mergeSelfDupI :: xs:_ → i:Nat → Prop (Dup i (merge xs xs)) @-}
mergeSelfDupI xxs@(x :> xs) i =
  MkDup i x (merge xs xs) (mergeSelfDupI xs)
  ? ( merge xxs xxs
   === x :> merge xxs xs
   === x :> x :> merge xs xs
   *** QED)

mergeSelfDup xs = dupLemmaC (merge xs xs) (mergeSelfDupI xs)
```

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.

- Indexed proof of `squareNNeg`:

```
{-@ squareNNegK :: xs:_ → k:Nat → {nnegK k (mult xs xs)} @-}
squareNNegK xs 0 = nnegK 0 (mult xs xs) *** QED
squareNNegK xxs@(x:>xs) k
  =   nnegK k (mult xxs xxs)
  === nnegK k (x * x :> mult xs xs)
  === (x * x >= 0 && nnegK (k-1) (mult xs xs))
    ? squareNNegK xs (k-1)
  *** QED

squareNNeg xs = nnegLemma (mult xs xs) (squareNNegK xs)
```

- Constructive proof of `squareNNeg`:

```
{-@ squareNNegI :: xs:_ → i:Nat
                        → Prop (NNeg i (mult xs xs)) @-}
```

```
squareNNegI xxs@(x :> xs) i
  =  NNegC i (x*x) (mult xs xs) (squareNNegI xs)
  ?  (mult xxs xxs === x*x :> mult xs xs *** QED)

squareNNeg xs = nnegLemmaC (mult xs xs) (squareNNegI xs)
```

4.2.3 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)
```

Like in the case of equality we can define a proof combinator for `belowK` in order to embellish our proof:

```
{-@ (<=#) :: x:Stream Int → k:{Nat | 0 < k}
          → y:{Stream Int | (belowK (k-1) (stail x) (stail y)
                               && shead x == shead y)
                             || shead x < shead y}
          → {v:Stream Int | belowK k x y && v = y } @-}
```

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.

We also encode the lemmas that we need to go from the indexed or constructive predicate to the intended coinductive:

```
assume belowLemma  :: xs:_ → ys:_ → (k:Nat → {belowK k xs ys})
                   → {below xs ys}
assume belowLemmaC :: xs:_ → ys:_ → (k:Nat → Prop (BelowC k xs ys))
                   → {below xs ys}
```

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)}
```

- Indexed proof of `belowSquare`:

```
{-@ belowSquareK :: a: Stream Int → k: Nat → {belowK k a (mult a a)}
 @-}
belowSquareK as 0
  =   belowK 0 as (mult as as)
  *** QED
belowSquareK (a :> as) k
   =  a :> as
   ?  belowSquareK as (k-1)
  <=# k #
      a * a :> mult as as
  === mult (a :> as) (a :> as)
  *** QED

belowSquare xs = belowLemma xs (mult xs xs) (belowSquareK xs)
```

- Constructive proof of `belowSquare`:

```
{-@ belowSquareI :: xs:_ → i:Nat → Prop (Below i xs (mult xs xs)) @
-}
belowSquareI xxs@(x :> xs) i
  | x == x*x
  = Bel0 i x xs (mult xs xs) (belowSquareI xs)
  | x < x*x
  = Bel1 i x (x*x) xs (mult xs xs) ? expand
  where
    expand =  mult xxs xxs
          === x*x :> mult xs xs
          *** QED

belowSquare xs = belowLemmaC xs (mult xs xs) (belowSquareI xs)
```

4.2.4 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 or constructive predicate, similar to streams, we assume the lemma below:

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

assume infLemmaC :: xs:L a → (k:Nat → Prop (Infinite k xs))
                 → {infinite xs}
```

- Indexed proof of `mapInfinite`:

```
mapInfiniteK :: (a → b) → List a → Int → Proof
mapInfiniteK f xs 0 = isInfiniteK 0 (map f xs) *** QED
mapInfiniteK f xs@Nil k = infinite xs *** QED
mapInfiniteK f xxs@(x :| xs) k
  =   isInfiniteK k (map f xxs)
  === isInfiniteK k (f x :| map f xs)
    ? (infinite xxs === infinite xs *** QED)
    ? mapInfiniteK f xs (k-1)
  *** QED

mapInfinite f xs = infLemma (map f xs) (mapInfiniteK f xs)
```

- Constructive proof of `mapInfinite`:

```
{-@ mapInfiniteS :: f:_ → {xs:_| infinite xs} → i:Nat
                      → Prop (Infinite i (map f xs))@-}
mapInfiniteS f xs@Nil _ =
  absurd (xs ? (infinite xs === False *** QED))
mapInfiniteS f xxs@(x :| xs) i =
  Inf i (f x) (map f xs) (mapInfiniteS f (xs ?infTail)) ? expand
  where expand =  map f xxs
              === f x :| map f xs
              *** QED
        infTail = infinite xxs
               === infinite xs
               *** QED

mapInfinite f xs = infLemmaC (map f xs) (mapInfiniteS f 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.

```
eqLemma  :: xs:_ → ys:_ → (k:Nat → {eqK k xs ys}) → {xs = ys}

eqLemmaC :: xs:_ → ys:_ → (k:Nat → Prop (Bisim k xs ys)) → {xs = ys}
```

- Indexed proof of `mapFusion`:

```
{-@ mapFusionK :: f:_ → g:_ → xs:_ → k:Nat
                → {eqK k (map (f . g) xs) (map f (map g xs))} @-}
mapFusionK :: (Eq a, Eq b, Eq c)
           ⇒ (b → c) → (a → b) → List a → Int → Proof
mapFusionK f g xs 0
  =   eqK 0 (map (f.g) xs) (map f (map g xs))
  *** QED
mapFusionK f g xs@Nil k | k > 0
  =   eqK k (map (f.g) xs) (map f (map g xs))
  === eqK k xs xs
  *** QED
mapFusionK f g xxs@(x :| xs) k | k > 0
  =   map (f.g) xxs
  === (f.g) x :| map (f.g) xs
  === (f.g) x :| map (f.g) xs
    ? mapFusionK f g xs (k-1)
  =#= k #
      f (g x) :| map f (map g xs)
  === map f (g x :| map g xs)
  === map f (map g xxs)
  *** QED

mapFusion f g xs =
  eqLemma (map (f . g) xs) (map f (map g xs)) (mapFusionK f g xs)
```

- Constructive proof of `mapFusion`:

```
{-@ mapFusionS :: f:_ → g:_ → xs:_ → i:Nat
                → Prop (Bisimilar i (map f (map g xs))
                                    (map (f . g) xs)) @-}
mapFusionS f g xs@Nil i = BisimNil i ? (map f (map g Nil) === Nil ***
QED)
                                     ? (map (f . g) Nil   === Nil ***
QED)
mapFusionS f g xxs@(x :| xs) i =
  Bisim i ((f . g) x) (map f (map g xs)) (map (f . g) xs)
```

```
      (mapFusionS f g xs) ? expandL ? expandR
  where expandL
         =  map f (map g xxs)
        === map f (g x :| map g xs)
        === (f (g x)) :| map f (map g xs)
        *** QED
      expandR
         =  map (f . g) xxs
        === (f . g) x :| map (f . g) xs
        *** QED

mapFusion f g xs =
  eqLemmaC (map f (map g xs)) (map (f . g) xs) (mapFusionS f g xs)
```

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". Graham Hutton and Jeremy Gibbons [Hutt01] 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.

Chapter 5

Related Work

Here we present the three mechanized verifiers that influenced our work (§5.1) and summarize how existing verifiers for Haskell programs treat coinduction (§5.2). We refer the reader to [Jaco97] for a foundational tutorial on coinduction and to [Gibb05] for (paper and pencil) proofs on Haskell corecursive programs.

5.1 Mechanized Coinduction

Coq has, for some time now, support for coinduction [Bert06]. The proving technique in §3.2 is partly inspired from Coq's textbook [Chli13] bisimilarity relation for infinite streams, where in place of syntactic guardedness we use natural numbers to keep track of productivity. The disadvantage of Coq's coinductive mechanization 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 §3.1 preserves local errors (and thus better user experience), while §3.2, as in Coq, has no proof steps and only returns a general failing error. As we also described, Coq has a guardedness condition which allows the definition of corecursive functions, but definitions like `fib` are not accepted.

Agda's coinduction [Abel10, Abel16] is quite similar to Coq's in the encoding of bisimilarity. A key difference is that Agda uses sizes (instead of syntactic guardedness) to encode productivity, a feature that we leverage to encode productivity in §2 and in §3.2 in order to construct our own bisimilarity relation in Liquid Haskell. In actual proofs, this difference is not significant since the invocation of the coinductive hypothesis is immediate. However, using sizes to encode guardedness in proves gives as a unified approach to both coinductive proofs and corecursive definitions.

Dafny's approach of coinduction [Lein14] greatly inspired our indexed approach (§3.1). 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. Of course, Dafny provides an automated program transformation that introduces indices, while in our case the transformation is manually performed by the user.

In [Lein14] we can also find a proof of soundness, which connects indexed proofs and predicates to coinductive ones. It uses the Kleene fix-point theorem [Wins93], 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.

Corecursive definitions in Dafny also use syntactic guardedness which has the limitations we have already discussed.

5.2 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 [Sonn12] 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 well-foundedness 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 [Farm12] and HALO [Vyti13] 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 here.

Hs-to-coq [Spec18] 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 coinductive properties we describe can be verified in Coq, via hs-to-coq.

Dependent Types for Haskell is a work initiated by Eisenberg [Eise16] 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.

5.3 Conclusion

We used Liquid Haskell to support coinductive features; namely to prove the productivity of various corecursive definitions and to prove coinductive properties.

We achieved productivity checking by altering the constructors and destructors of coinductive objects to keep track of the depth of the object and used this infrastructure to define and prove the productivity of various objects.

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 proposed encodings, currently manually provided by the user. Regarding productivity proofs, future work can be applied to either automating the process by trying to infer depth annotations where possible, or to binding the proofs with the original definitions so as to hide away the complexity of depths. In the coinductive proof setting we see two potential directions for such automation. First, we could follow Dafny's approach [Lein14] 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 [Reyn17]) support for codata to reason about coinductive properties using SMT's decision procedures.

Βιβλιογραφία

[Abel10] Andreas Abel, "MiniAgda: Integrating Sized and Dependent Types", *Electronic Proceedings in Theoretical Computer Science*, vol. 43, pp. 14–28, December 2010.

[Abel13] Andreas Abel, Brigitte Pientka, David Thibodeau and Anton Setzer, "Copatterns: Programming Infinite Structures by Observations", in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, p. 27–38, New York, NY, USA, 2013, Association for Computing Machinery.

[Abel16] Andreas Abel and Brigitte Pientka, "Well-founded Recursion with Copatterns and Sized Types", *Journal of Functional Programming*, 2016.

[Barr97] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent-vigouroux, Christine Paulin-Mohring, Amokrane Saïbi and Benjamin Werner, *The Coq Proof Assistant Reference Manual : Version 6.1*, June 1997.

[Bert06] Yves Bertot, "CoInduction in Coq", *CoRR*, 2006.

[Bird88] Richard Bird and Philip Wadler, *Introduction to Functional Programming*, Prentice Hall International, 1988.

[Bork22] Michael Borkowski, Niki Vazou and Ranjit Jhala, "Mechanizing Refinement Types", in *CoRR*, 2022.

[Chli13] Adam Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*, MIT Press, 2013.

[Clou15] Ranald Clouston, Aleš Bizjak, Hans Grathwohl and Lars Birkedal, "Programming and Reasoning with Guarded Recursion for Coinductive Types", in Andrew Pitts, editor, *Foundations of Software Science and Computation Structures*, vol. 9034, pp. 407–421, Berlin, Heidelberg, January 2015, Springer.

[Eise16] Richard A. Eisenberg, *Dependent Types in Haskell: Theory and Practice*, Ph.D. thesis, University of Pennsylvania, 2016.

[Farm12] Andrew Farmer, Andy Gill, Ed Komp and Neil Sculthorpe, "The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs", in *Proceedings of the 2012 Haskell Symposium*, Haskell '12, p. 1–12, 2012.

[Gibb05] Jeremy Gibbons and Graham Hutton, "Proof Methods for Corecursive Programs", *Fundamenta Informaticae*, 2005.

[Hutt01] Graham Hutton and Jeremy Gibbons, "The Generic Approximation Lemma", *Information Processing Letters*, 2001.

[Jaco97] Bart Jacobs and Jan J. M. M. Rutten, "A Tutorial on (Co)Algebras and (Co)Induction", *Bulletin of The European Association for Theoretical Computer Science*, vol. 62, pp. 62–222, 1997.

[Jone93] Geraint Jones and Jeremy Gibbons, "Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips", Technical report, Dept of Computer Science, University of Auckland, 1993.

[Lein14] K. Rustan M. Leino and Michał Moskal, "Co-induction Simply", in Cliff Jones, Pekka Pihlajasaari and Jun Sun, editors, *Formal Methods*, pp. 382–398, Springer International, 2014.

[Mast22] Lykourgos Mastorou, Niki Vazou and Nikolaos Papaspyrou, "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*, Haskell '22, 2022.

[Peyt06] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich and Geoffrey Washburn, "Simple Unification-Based Type Inference for GADTs", in *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, p. 50–61, 2006.

[Reyn17] Andrew Reynolds and Jasmin Christian Blanchette, "A Decision Procedure for (Co)datatypes in SMT Solvers", in *Journal of Automated Reasoning*, 2017.

[Rosu09] Grigore Roşu and Dorel Lucanu, "Circular Coinduction: A Proof Theoretical Foundation", in Alexander Kurz, Marina Lenisa and Andrzej Tarlecki, editors, *Algebra and Coalgebra in Computer Science*, pp. 127–144, Berlin, Heidelberg, 2009, Springer.

[Sonn12] William Sonnex, Sophia Drossopoulou and Susan Eisenbach, "Zeno: An automated prover for properties of recursive data structures", in Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 407–421, Berlin, Heidelberg, 2012, Springer.

[Spec18] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah and Stephanie Weirich, "Total Haskell is Reasonable Coq", in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, p. 14–27, New York, NY, USA, 2018, Association for Computing Machinery.

[Team22] Agda Team, "The Agda Wiki", 2022.

[Vazo14] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis and Simon Peyton-Jones, "Refinement Types for Haskell", in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, p. 269–282, 2014.

[Vazo17] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler and Ranjit Jhala, "Refinement Reflection: Complete Verification with SMT", *Proc. ACM Program. Lang.*, vol. 2, no. POPL, December 2017.

[Vazo18] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn and Graham Hutton, "Theorem Proving for All: Equational Reasoning in Liquid Haskell (Functional Pearl)", in *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, p. 132–144, 2018.

[Vyti13] Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen and Dan Rosén, "HALO: Haskell to Logic through Denotational Semantics", in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, p. 431–442, 2013.

[Wins93] Glynn Winskel, *The Formal Semantics of Programming Languages: An Introduction*, MIT Press, 1993.

[Xi03] Hongwei Xi, Chiyan Chen and Gang Chen, "Guarded Recursive Datatype Constructors", in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, p. 224–235, 2003.