A Tale of Two Provers

Verifying Monoidal String Matching in Liquid Haskell and Coq

Niki Vazou
University of Maryland

Leonidas Lampropoulos
University of Pennsylvania

Jeff Polakow
Awake Networks

Abstract

We demonstrate for the first time that Liquid Haskell, a refinement type checker for Haskell programs, can be used for arbitrary theorem proving by verifying a parallel, monoidal string matching algorithm implemented in Haskell. We use refinement types to specify correctness properties, Haskell terms to express proofs of these properties, and Liquid Haskell to check the proofs. We evaluate Liquid Haskell as a theorem prover by replicating our 1428 LoC proof in a dependently-typed language (Coq - 1136 LoC). Finally, we compare both proofs, uncovering the relative advantages and disadvantages of the two provers.

CCS Concepts •**Software and its engineering** → *Formal software verification;*

Keywords Liquid Haskell, Coq, Dependent and Refinement Types, Formal Verification, Parallelization, Theorem Proving, Monoid Laws

ACM Reference format:
Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. 2017. A Tale of Two Provers. In *Proceedings of ACM SIGPLAN Conference on Programming Languages, USA, May 2017 (Haskell '17),* 12 pages.
DOI: 10.1145/3122955.3122963

1 Introduction

Liquid Haskell [25] is a verifier for Haskell programs that semi-automatically checks whether the code satisfies logical specifications – expressed as refinement types – using an SMT [1] solver. Traditionally [11], refinement types are syntactically restricted over decidable theories (*e.g.,* linear arithmetic and uninterpreted functions) permitting fully SMT-automated type checking. Liquid Haskell extends traditional refinement type specifications with (undecidable) properties over arbitrary, terminating Haskell functions [26]. While proofs over decidable properties are completely automated via the SMT solver, the user has to manually prove the non-decidable proof obligations. These manual proofs are written as plain Haskell programs; thus, Haskell becomes a theorem prover.

In this paper we present the first non-trivial, 1428 LoC, application of Liquid Haskell as a theorem prover: we prove the correctness of both a monoid for string matching and a monoid morphism from strings to our string matching monoid. This monoid morphism is a string matching function which can be run in parallel over adjacent chunks of an input string, the results of which can be combined, also in parallel, into the final match results. We replicate these correctness proofs in Coq (1136 LoC) and empirically compare the two approaches. Both proofs are available online [28].

The contributions of this paper are outlined as follows.

- We explain how theorems and proofs are encoded and checked in Liquid Haskell by formalizing monoids and proving that lists form a monoid (§ 2). We also use this section to introduce notations and background necessary in the rest of the paper.
- We create the first large application of Liquid Haskell as a theorem prover: a verified parallelizable string matcher. We do this by first formalizing monoid morphisms and showing that such morphisms on "chunkable" input can be correctly parallelized (§ 3) by following the three steps of the MapReduce algorithm:
  1. divide the input in chunks,
  2. apply the morphism in parallel to all chunks, and
  3. recombine in parallel the mapped chunks.

  Our proof assumes the correctness of Haskell's `parallel` library. We then apply these three steps (§ 5) to a sequential string matcher to obtain a correct, parallel (and thus faster) version.
- We evaluate the applicability of Liquid Haskell as a theorem prover by repeating the same proof in the Coq proof assistant. We identify interesting tradeoffs in the verification approaches encouraged by the two tools in two parts: we first draw preliminary conclusions based on the general parallelization steps (§ 4) and then we delve deeper into the comparison, highlighting differences based on the string matching case study (§ 6). Finally (§ 7), we complete the evaluation picture by providing additional quantitative comparisons of the two provers.

2 Haskell as a Theorem Prover

In this section we demonstrate how Haskell can be used as a theorem prover by proving that lists form a monoid. Concretely, we

- *specify monoid laws* as refinement types,
- *prove the laws* using plain Haskell functions, and
- *verify the proofs* using Liquid Haskell.

We start (§ 2.1) by defining a Haskell `List` datatype with the associated monoid elements ϵ and $\diamond$ corresponding to the empty list and concatenation. We then prove the three monoid laws (§ 2.2, § 2.4, and § 2.5) in Liquid Haskell. Finally (§ 2.6), we conclude that lists are indeed monoids.

2.1 Reflection of Lists into Logic

To begin with, we define a standard recursive `List` datatype.

```
data L [length] a
  = N | C {head :: a, tail :: L a}
```

The `length` annotation in the definition teaches Liquid Haskell to use `length` to check the termination of functions recursive on lists. The `length` function is defined as a standard Haskell function.

```
length          :: L a → {v:Integer | 0 ≤ v}
length N         = 0
length (C x xs) = 1 + length xs
```

The refinement type specifies that `length` returns a natural number, that is, `length` returns a Haskell `Integer` value `v` that is moreover refined to satisfy the constraint `0 ≤ v`. To check the validity of this specification, Liquid Haskell encodes Haskell's `Integer` as a logical integer[1] and via standard refinement type constraint generation [9, 27], generates two proof obligations. For the `N` case it checks that the body $v = 0$ is a natural number.

$$v = 0 \Rightarrow 0 \leq v \tag{1}$$

For the `C` case Liquid Haskell binds the recursive call to a fresh variable v_r = `length` *xs* and checks that assuming the specification for v_r, *i.e.,* assuming that v_r is a natural number, the body $v = 1 + v_r$ is also non negative.

$$0 \leq v_r \Rightarrow v = 1 + v_r \Rightarrow 0 \leq v \tag{2}$$

Liquid Haskell decides the validity of both these proof obligations automatically using an SMT solver.

We define the two monoid operators on Lists: an identity element ϵ (the empty list) and an associative operator (◇) (list append).

```
ϵ :: L a      (◇)           :: L a → L a → L a
ϵ = N         N ◇ ys        = ys
              (C x xs) ◇ ys = C x (xs ◇ ys)
```

Our goal is to specify and prove the monoid laws on the above operators using Liquid Haskell. However, to preserve the decidability of SMT-automated type checking, Liquid Haskell does not automatically lift arbitrary Haskell functions in the refinement logic. Instead, it enforces a clear separation between Haskell functions and their interpretation into the SMT logic, allowing only the refinement specification of the function, *i.e.,* a decidable abstraction of the Haskell function, to flow into the SMT logic. For example, the validity check of both the linear arithmetic statements (1) and (2) is automatically decided by the SMT, since the recursive call `length xs` is, by default, interpreted in the logic as a value v_r that only satisfies the `length` specification of being a natural number.

Liquid Haskell lifts Haskell functions into the logic using the **`measure`** and **`reflect`** annotations, that preserve SMT decidability.

- The **`measure`** `f` annotation [27] lifts into the logic the Haskell function `f`, if `f` is syntactically defined on precisely one Algebraic Data Type (ADT). Due to this syntactic restriction the measure `f` is automatically unfolded into the SMT logic (*i.e.,* imitating automatic type level computations).
- The **`reflect`** `f` annotation [26] lifts the arbitrary, terminating Haskell function `f` into the logic but, for decidable type checking, `f` is not automatically unfolded in the logic. Instead, as we shall describe, type level unfolding of the reflected function `f` is manually performed via respective value level computations.

Since `length` is defined on exactly one ADT (*i.e.,* the List) it is lifted in the refinement logic as a **`measure`**

```
measure length
```

With the above measure annotation, Liquid Haskell interprets `length` into the logic by automatically strengthening the types of the `List` data constructors. For example, the type of `C` is *automatically* strengthened to

```
C :: x:a → xs:L a
  → {v:L a | length v = length xs + 1 }
```

where `length` is an uninterpreted function in the logic.

We lift the monoid operators ϵ and (◇) in the logic via reflection.

```
reflect ϵ, (◇)
```

The **`reflect`** annotations lift (◇) and (ϵ) into the logic by *automatically* strengthening the types of the functions' specifications.

```
(ϵ) :: {v:L a | v = ϵ ∧ v = N}

(◇) :: xs:L a → ys:L a
    → {v:L a | v = xs ◇ ys
      ∧ v = if isN xs then ys
         else C (head xs) (tail xs ◇ ys)}}
```

Here, the (◇) and (ϵ) appearing in the refinements are uninterpreted functions and `isN`, `head`, and `tail` are automatically generated measures. To preserve predictable type checking, Liquid Haskell will not attempt to unfold the reflected functions into the logic [14]. But after reflection, at each Haskell function call the function definition is unfolded exactly once into the logic, allowing Liquid Haskell to prove properties about Haskell functions.

2.2 Left Identity

In Liquid Haskell, we express theorems as refined type specifications and proofs as their Haskell inhabitants. We construct proofs using the combinators from the built-in library `ProofCombinators`[2] that are summarized in Figure 1. A **`Proof`** is a unit type that when refined is used to specify theorems. A `trivial` proof is the unit value. For example, `trivial :: {v:`**`Proof`**` | 1 + 2 = 3}` trivially proves the theorem `1 + 2 = 3` using the SMT solver. The expression `p *** `**`QED`** casts any expression `p` into a **`Proof`**. The equality assertion `x ==. y` states that `x` and `y` are equal and returns the first argument for use in the rest of the proof. We extend the equality assertion to receive an optional third proof argument. For instance, `x ==. y lemma` proves `x = y` using the proof term `lemma`. To avoid parenthesizing the optional proof argument in the common case where `lemma` is an application and not a variable, we follow the same approach as Haskell's dollar ($) and define the ∴ operator with appropriate precedence (thus, we can write `x ==. y ∴ lemma`). Finally, `x∧.y` combines two proofs `x` and `y` into one by inserting the argument proofs into the logical environment.

Armed with these combinators, left identity is expressed as a refinement type signature that takes as input a list `x:L a` and returns a **`Proof`** (*i.e.,* unit) type refined with the property `ϵ ◇ x = x`.

```
idLeft_List :: x:L a → { ϵ ◇ x = x }
idLeft_List x = ϵ ◇ x ==. N ◇ x ==. x *** QED
```

We write `{ϵ ◇ x = x}` as a simplification for `{v:`**`Proof`**` | ϵ ◇ x = x}` since the binder `v` is irrelevant. We begin from the left hand side `ϵ ◇ x`, which is equal to `N ◇ x` by calling ϵ thus unfolding the equality `empty = N` into the logic. Next, the call `N ◇ x` unfolds the definition of (◇) on `N` and `x`, which is equal to `x`, concluding our proof. Finally, we use the operator `p *** `**`QED`** which casts `p` into a proof term. In short, the proof of left identity, proceeds by unfolding the definitions of ϵ and (◇) on the empty list.

[1] It is possible to encode bounded Int in Liquid Haskell (an example of such an encoding can be found in Arithmetic Overflows) but this encoding would require extra in-bound checking proof obligations for all Int operators leading to imprecise verification.

[2] The `ProofCombinators` library comes with Liquid Haskell and is defined in https://github.com/ucsd-progsys/liquidhaskell/blob/develop/include/Language/Haskell/Liquid/ProofCombinators.hs.

```
type Proof   = ()                       trivial   :: Proof                  (==.) :: x:a -> y:{a|x = y} -> {v:a|v = x}
data QED   =  QED                       trivial   = ()                      x ==. _ = x

(***)   :: a -> QED -> Proof            (∴) :: (Proof -> a) -> Proof -> a   (∧.) :: Proof -> Proof -> Proof
_ *** _    = ()                         thm ∴ lemma = thm lemma             _ ∧. _ = ()
```

Figure 1. Operators and Types defined in ProofCombinators Library.

2.3 PLE: Proof by Logical Evaluation

To automate trivial proofs, Liquid Haskell uses PLE (Proof by Logical Evaluation) a terminating but incomplete heuristic, inspired by [14], that automatically unfolds reflected functions in proof terms. PLE evaluates (*i.e.,* unfolds) a reflected function call if it can be statically decided what branch the evaluation takes, *e.g.,* `N ◇ ys` is unfolded to `ys` while `xs ◇ ys` is not unfolded when the structure of `xs` cannot be statically decided. Unlike SMT's axiom instantiation heuristics (*e.g.,* E-matching [6, 18]) that make verification unstable [14], PLE is always terminating and is enabled on a per-function basis. For instance, the annotation

```
automatic-instances idLeft_List
```

activates PLE in the `idLeft_List` function. When PLE is used to complete a proof, it could be unpredictable whether proof synthesis succeeds, yet the verification of the rest of the program is not affected. Thus, global verification stability is preserved.

PLE is used to simplify the left identity proof by automatically unfolding ϵ to `N` and then `N ◇ x` to `x`. (We use the cornered one line frame to denote Liquid Haskell proofs that use PLE via the `automatic-instances` annotation.)

```
idLeft_List :: x:L a → { ϵ ◇ x = x }
idLeft_List _ = trivial
```

That is the proof proceeds, trivially, by logical evaluation of `ϵ ◇ x`.

2.4 Right Identity

Right identity is proved by structural induction. We encode inductive proofs by case splitting on the base and inductive case, and by enforcing the inductive hypothesis via a recursive call.

```
idRight_List :: x:L a → { x ◇ ϵ = x }
idRight_List N = N ◇ ϵ ==. N *** QED
idRight_List (C x xs)
   =   (C x xs) ◇ ϵ
   ==. C x (xs ◇ ϵ)
   ==. C x xs ∴ idRight_List xs
   *** QED
```

The recursive call `idRight_List xs` is provided as a third optional argument in the `(==.)` operator to justify the equality `xs ◇ ϵ = xs`, while the operator `(∴)` is merely a function application with the appropriate precedence. Since Haskell is pure, to ensure well formedness of proof terms one merely needs to check that such terms are not partial. Liquid Haskell verifies that all the proof terms are well formed via termination and totality checking since (1) the inductive hypothesis is only applying to smaller terms and (2) all cases are covered.

We use the PLE heuristic to automatically generate all function unfoldings and simplify the right identity proof.

```
idRight_List :: x:L a → { x ◇ ϵ = x }
idRight_List N        = trivial
idRight_List (C _ xs) = idRight_List xs
```

PLE performs symbolic unfolding but not case splitting, that is the cases should be explicitly split by the user. For instance, in the `C` branch the term `C x xs ◇ ϵ` automatically unfolds to `C x (xs ◇ ϵ)`. Then the SMT will use the inductive hypothesis and congruence to conclude the proof.

2.5 Associativity

Similarly, we prove associativity using structural induction.

```
assoc_List :: x:L a → y:L a → z:L a
           → {x ◇ (y ◇ z) = (x ◇ y) ◇ z}
assoc_List N _ _       = trivial
assoc_List (C _ x) y z = assoc_List x y z
```

As with the left identity, the proof proceeds by (1) function unfolding (or rewriting in paper and pencil proof terms), (2) case splitting (or case analysis), and (3) recursion (or induction).

2.6 Lists are a Monoid

Finally, we formally define monoids as structures that satisfy the monoid laws of associativity and identity and conclude that `L a` is indeed a monoid.

Definition 2.1 (Monoid)**.** The triple (`m`, ϵ, ◇) is a monoid (with identity element ϵ and associative operator ◇), if the following functions are defined.

```
idLeft_m  :: x:m → {ϵ ◇ x = x}
idRight_m :: x:m → {x ◇ ϵ = x}
assoc_m   :: x:m → y:m → z:m
          → {x ◇ (y ◇ z) = (x ◇ y) ◇ z}
```

Note that for each monoid law we use the subscript $_m$ to denote a different proof term for different monoids. Ideally, we would like to define proof terms as extra methods in the monoid class, but since Liquid Haskell does not yet support theorem proving on class methods in our implementation we need to redefine each monoid method as a Haskell function for each monoid.

Corollary 2.2. *(*`L a`*, ϵ, ◇) is a monoid.*

3 Verified Parallelization of Morphisms

A monoid morphism is a function between two monoids which preserves the monoidal structure. We call a monoid morphism *chunkable* if its domain can be split into pieces. To parallelize a chunkable morphism `f` we:

§ 3.1 chunk up the input in chunks of size `i` (`chunk i`),
§ 3.2 apply `f` in parallel to all chunks (`pmap f`), and

§ 3.3 recombine the chunks, in parallel `j` at a time, back to a single value (`pmconcat j`).

In this section we implement and verify in Liquid Haskell the correctness of the transformation

```
f = pmconcat j . pmap f . chunk i
```

We rely on the correctness of Haskell's parallelization primitive (`withStrategy`) that is *assumed* to be correct.

3.1 Lists are Chunkable Monoids

Definition 3.1 (Chunkable Monoids)**.** We define a monoid (`m`, ϵ, $\diamond$) to be chunkable if for every natural number `i` and monoid `x`, the functions take$_m$ `i x` and drop$_m$ `i x` are defined in such a way that take$_m$ `i x` $\diamond$ drop$_m$ `i x` exactly reconstructs `x`.

```
length_m :: m → Nat
drop_m   :: i:Nat → x:{m | i ≤ length_m x}
         → {v:m | length_m v = length_m x - i}
take_m   :: i:Nat → x:{m | i ≤ length_m x}
         → {v:m | length_m v = i}
take_drop_spec_m :: i:Nat → x:m
         → {x = take_m i x ◇ drop_m i x}
```

The functional methods of chunkable monoids are `take` and `drop`, while the `length` method is required to give the pre- and post-condition on the other operations. The proof term `take_drop_spec` specifies the reconstruction property.

Next, we use the take$_m$ and drop$_m$ methods for each chunkable monoid (`m`, ϵ, $\diamond$) to define a chunk$_m$ `i x` function that splits `x` in chunks of size `i`.

```
type Pos = {v:Integer | 0 < v}

chunk_m :: i:Pos → x:m
        → {v:L m | chunk_spec_m i x v}
        /  [length_m x]
chunk_m i x
  | length_m x ≤ i
  = C x N
  | otherwise
  = take_m i x `C` chunk_m i (drop_m i x)
```

To prove termination of chunk$_m$ Liquid Haskell checks that the user-defined termination metric (written `/ [length`$_m$ `x]`) decreases at the recursive call. The check succeeds as drop$_m$ `i x` is specified to return a monoid smaller than `x`. We specify the `length` of the chunked result using the specification function chunk_spec$_m$.

```
chunk_spec_m i x v
  | length_m x ≤ i = length v == 1
  | i == 1         = length v == length_m x
  | otherwise      = length v <  length_m x
```

The specifications of both take$_m$ and drop$_m$ are used to automatically verify the length$_m$ constraints imposed by chunk_spec$_m$.

Finally, we prove that the `Lists` defined in § 2 are chunkable monoids.

```
take_List i N = N
take_List i (C x xs)
  | i == 0    = N
  | otherwise = C x (take_List (i-1) xs)
```

```
drop_List i N = N
drop_List i (C x xs)
  | i == 0    = C x xs
  | otherwise = drop_List (i-1) xs
```

The above definitions follow the library built-in definitions on lists, but they need to be redefined for the reflected, user defined list data type. On the plus side, Liquid Haskell will *automatically* prove that the above definitions satisfy the specifications of the chunkable monoid, using the length defined in the previous section. Finally, the take-drop reconstruction specification is proved by induction on the size `i` and using the PLE tactic for the trivial static evaluation.

```
take_drop_spec_List i N
  = trivial
take_drop_spec_List i (C x xs) | i == 0
  = trivial
take_drop_spec_List i (C x xs)
  = take_drop_spec_List (i-1) xs
```

3.2 Parallel Map

We define a parallelized map function `pmap` using Haskell's `parallel` library. Concretely, we use the parallelization function `withStrategy`, from `Control.Parallel.Strategies`, that computes its argument in parallel given a parallel strategy.

```
pmap :: (a → b) → L a → L b
pmap f xs = withStrategy parStrategy (map f xs)
```

Parallelism in the Logic. The function `withStrategy`, that performs the runtime parallelization, is an imported Haskell library function, whose implementation is not available during verification. To use it in our verified code, we make the *assumption* that it always returns its second argument.

```
assume withStrategy :: Strategy a
                    → x:a → {v:a | v = x}
```

Moreover, to reflect the implementation of `pmap` in the logic, the function `withStrategy` should also be represented in the logic. LiquidHaskell encodes `withStrategy` in the logic as a logical, *i.e.,* total, function that merely returns its second argument, `withStrategy _ x = x`. That is, our proof does not reason about runtime parallelism; we prove the correctness of the parallelization transformation, assuming the correctness of the parallelization primitive.

Under this encoding, the parallel strategy chosen does not affect verification. In our codebase we defined `parStrategy` to be the traversable strategy.

```
parStrategy :: Strategy (L a)
parStrategy = parTraversable rseq
```

3.3 Parallel Monoidal Concatenation

The function chunk$_m$ lets us turn a monoidal value into several pieces. Dually, for any monoid (`m`, ϵ, $\diamond$), the monoid concatenation mconcat$_m$ turns a `L m` back into a single `m`.

```
mconcat_m :: L m → m
mconcat_m N        = ϵ
mconcat_m (C x xs) = x ◇ mconcat_m xs
```

Next, we parallelize the monoid concatenation by defining the function $\texttt{pmconcat}_m$ that chunks the input list of monoids and concatenates each chunk in parallel.

```
pmconcat_m :: Integer → L m → m
pmconcat_m i x | i ≤ 1 || length x ≤ i
 = mconcat_m x
pmconcat_m i x
 = pmconcat_m i (pmap mconcat_m (chunk i x))
```

Where chunk is the list chunkable operation `chunk_List`. The function $\texttt{pmconcat}_m$ `i x` calls $\texttt{mconcat}_m$ `x` in the base case, otherwise it (1) chunks the list `x` in lists of size `i`, (2) runs in parallel $\texttt{mconcat}_m$ to each chunk, and (3) recursively runs itself with the resulting list. Termination of $\texttt{pmconcat}_m$ holds, as the length of `chunk i x` is smaller than the length of `x`, when `1 < i`.

Finally, we prove the correctness of the parallelization of monoid concatenation.

Theorem 3.2. *For each monoid* ($\mathtt{m}$, ϵ, $\diamond$) *the parallel and sequential concatenations are equivalent:*

```
pmconcatEq :: i:Integer → x:L m
  → { pmconcat_m i x = mconcat_m x }
```

Proof. The proof proceeds by structural induction on the input list `x`. The details of the proof can be found in [28], here we sketch the proof.

First, we prove that `mconcat` distributes over list cutting.

```
mcut :: i:Nat → x:LLEq m i
     → {mconcat_m x = mconcat_m (take i x)
                    ◇ mconcat_m (drop i x)}

type LLEq m I = {xs: L m | I ≤ length xs}
```

We generalize the above lemma to prove that `mconcat` distributes over list chunking.

```
mchunk :: i:Integer → x:L m
  → {mconcat_m x =
      mconcat_m (map mconcat_m (chunk i x))}
```

Both lemmata are proved by structural induction on the input list `x`. Lemma `mchunk` proves `pmconcatEq` by structural induction, using left identity in the base case. □

3.4 Parallel Monoid Morphism

We conclude this section by specifying and verifying the correctness of generalized monoid morphism parallelization.

Theorem 3.3 (Correctness of Parallelization)**.** *Let* ($\mathtt{m}$, ϵ, $\diamond$) *be a monoid and* ($\mathtt{n}$, η, $\boxdot$) *be a chunkable monoid. Then, for every morphism* `f :: n → m`*, every positive number* `i` *and* `j`*, and input* `x`*,* $\texttt{f x = pmconcat i (pmap f (chunk}_n\texttt{ j x))}$ *holds.*

```
parallelismEq
 :: f:(n → m) → Morphism n m f
 → x:n → i:Pos → j:Pos →
 {f x = pmconcat_m i (pmap f (chunk_n j x))}
```

where the `Morphism n m f` *argument is a proof argument that validates that* `f` *is indeed a morphism via the refinement type alias*

```
type Morphism n m F = x:n → y:n
  → {F η = ϵ ∧ F (x ⊡ y) = F x ◇ F y}
```

Proof. We prove the equivalence in two steps. First we prove a lemma (`parallelismLemma`) that the equivalence holds when the mapped result is concatenated sequentially. Then, we prove parallelism equivalence by defining a valid inhabitant for `parallelismEq`.

Lemma 3.4. *Let* ($\mathtt{m}$, ϵ, $\diamond$) *be a monoid and* ($\mathtt{n}$, η, $\boxdot$) *be a chunkable monoid. Then, for every morphism* `f : n → m`*, every positive number* `i` *and input* `x`*,* $\texttt{f x = mconcat}_m\texttt{ (pmap f (chunk}_n\texttt{ i x))}$ *holds.*

```
parallelismLemma :: f:(n → m) → Morphism n m f
  → x:n → i:Pos
  → {f x = mconcat_m (pmap f (chunk_n i x))}
```

Proof. The proof proceeds by induction on the length of the input.

```
parallelismLemma f thm x i | length_n x ≤ i
  =   idRight_m (f is)
parallelismLemma f thm x i
  =   parallelismLemma f thm dropX i
  ∧. thm takeX dropX ∧. takeDropProp_n i x
  where
    dropX = drop_n i x
    takeX = take_n i x
```

In the base case we use rewriting and right identity on the monoid `f x`. In the inductive case, we use the inductive hypothesis with $\texttt{dropX = drop}_n\texttt{ i x}$, that is provably smaller than `x` as `1 < i`. We get basic distribution for f: `f takeX ◇ f dropX = f (takeX ⊡ dropX)`, since `f` is a monoid morphism as encoded in the argument `thm takeX dropX`. Finally, by the $\texttt{takeDropProp}_n$ property of the chunkable monoid `n` we merge `takeX ⊡ dropX` to `x`. □

Finally, the `parallelismEq` function is defined using the above lemma combined with the equivalence of parallel and sequential `mconcat` as encoded by `pmconcatEq` in Theorem 3.2.

```
parallelismEq f thm x i j
  =   pmconcatEq i (pmap f (chunk_n j x))
  ∧. parallelismLemma f thm x j
```

□

4 Monoid Morphism Parallelization in Coq

To put Liquid Haskell as a theorem prover into perspective, we replicated the proof of the Parallel Monoid Morphism (Theorem 3.3) in the Coq proof assistant. In this section we present the main differences that appeared during this effort.

4.1 Intrinsic vs. Extrinsic Verification

The translation of the chunkable monoid specification of § 3.1 in Coq is a characteristic instance of how Liquid Haskell and Coq naturally favor intrinsic and extrinsic verification respectively. The (intrinsic) Liquid Haskell pre- and post-conditions of the `take` and `drop` functions are not embedded in the Coq types, but are independently, *i.e.,* extrinsically, encoded as specification terms in the extra `drop_spec` and `take_spec` methods. (We use the double-lined code to frame Coq code.)

```
lengthm : M → nat;
dropm   : nat → M → M;
takem   : nat → M → M;

drop_specm     : ∀ i x, i ≤ lengthm x →
    lengthm (dropm i x) = lengthm x - i;
take_specm     : ∀ i x, i ≤ lengthm x →
    lengthm (takem i x) = i;
take_drop_specm: ∀ i x,
    x = takem i x ◇ dropm i x;
```

Liquid Haskell favors intrinsic verification, as the shallow specifications of take and drop are embedded into the functions and automatically proved by the SMT solver. On the contrary, Coq users can (and usually) take the extrinsic verification approach, where the specifications of take and drop are encoded as independent specification terms. Since, unlike Liquid Haskell, the Coq specification terms should be explicitly proved by the user, the extrinsic approach significantly improves readability and ease-of-use of Coq code, as the function implementations are not littered by the specifications' proofs.

4.2 User-Defined vs. Library Functions

In Coq, we can leverage existing library functions and their specifications —here ssreflect's **seq** [12]— to define the chunkable monoid operations that had to be defined from scratch in Liquid Haskell (§ 3.1).

```
Definition length_list := @seq.size A;
Definition drop_list   := @seq.drop A;
Definition take_list   := @seq.take A;
```

Coq's libraries also come with already established theories. For example, to prove the drop_spec_list we just apply an existing library lemma (**seq**.size_drop), unlike Liquid Haskell that provides no such library support.

4.3 SMT- vs Tactic-Based Automation

Unlike Liquid Haskell that uses the SMT to automatically construct proofs over decidable theories, such as linear arithmetic, Coq requires explicit proof terms. For example, consider the proof of the take specification for lists.

```
Theorem take_spec_list :
  ∀ i x, i ≤ length_list x →
    length_list (drop_list i x) = i.
```

The crux of the proof lies in the application of the library lemma size_take.

```
Lemma size_take x : size (take i x) =
  if i < size x then i else size x.
```

However, the existing lemma and our desired specification differ when i is exactly equal to size x, generating a linear arithmetic proof obligation. While in Liquid Haskell such obligations are automatically discharged by the SMT, in the Coq implementation [28] we resort to an adaptation of the Presburger Arithmetic solver omega [21] for ssreflect.

This trivial example highlights the difference between using the SMT and tactics (like omega) for proof automation. SMT verification is complete over a limited number of theories such as linear arithmetic, and, in Liquid Haskell, the user has no way to expand these theories. On the contrary, in Coq the user has the option of customizing the automation (*e.g.,* by expanding the hint database or by writing more domain-specific tactics). However, even the "nuclear option", omega, is not complete. When it fails (which is not a rare situation as we found out during our development), the user has to manually complete the proof. Worse, the proofs generated by omega are far from ideal; as stated by The Coq development team [5] : "The simplification procedure is very dumb and this results in many redundant cases to explore. Much too slow."

4.4 Semantic vs. Syntactic Termination Checking

Since non-terminating programs introduce inconsistencies in the logic, all reflected Haskell functions and all Coq programs are provably terminating. A first difference between termination checking in the two provers is that Liquid Haskell allows non-reflected, Haskell functions (that do not flow into the logic) to be potentially diverging [27], while Coq, that does not explicitly distinguish between logic and implementation, does not, by default, support partial computations. Making such a distinction between logic and implementation in a dependently typed setting is in fact a research problem of its own [3].

The second difference is that Liquid Haskell uses a semantic termination checker, unlike Coq that is using a particularly restrictive syntactic criterion, where only recursive calls on subterms of some principal argument are allowed. Consider for example the chunk definition of § 3.1. Liquid Haskell semantically checks termination of chunk using the user-provided termination metric that [length x] that specifies that the length of x is decreasing at each recursive call. To persuade Coq's syntactic termination checker that chunk terminates, we extended chunk with an additional natural number fuel argument that trivially decreases at each recursive call.

```
Fixpoint chunkm {M: Type} (fuel : nat)
  (i : nat) (x : M) : option (list M)
```

We defined chunk$_m$ to be None when not enough fuel is provided, otherwise it follows the Haskell recursive implementation. This makes our specifications existentially quantified:

```
Theorem chunk_specm : ∀ {M} i (x : M) ,
   i > 0 → exists l,
   chunkm (lengthm x).+1 i x = Some l
   /\ chunk_resm i x l.
```

The above specification enforces both the length specifications as encoded in chunk's Liquid Haskell type and the successful termination of the computation given sufficient fuel.

The fuel technique is a common way to encode non-structural recursion, heavily used in CompCert [15]. Various such techniques have been developed by the Coq community to tackle such recursions. In "Certified Programming with Dependent Types" [4], Chlipala compares three general techniques to bypass Coq's syntactic termination restriction: well-founded recursion (e.g. using Function (§ 2.3 of [5])), domain-theory-inspired non-termination monads (where our fuel-based approach can be roughly categorized), and co-inductive non-termination monads. However, no single method is found to be ideal.

4.5 Executable vs Axiomatized Parallelism

In Liquid Haskell, we reason about Haskell programs that use libraries from the Haskell ecosystem. For instance, in § 3.2 we used the library `parallel` for runtime parallelization and we axiomatized parallelism in logic. Coq does not have such a library, so we axiomatize not only the behavior but also the existence of parallel functions:

```
Axiom Strategy          : Type.
Axiom parStrategy       : Strategy.
Axiom withStrategy
  : ∀ {A}, Strategy → A → A.
Axiom withStrategy_spec
  : ∀ {A} (s : Strategy) (x : A),
    withStrategy s x = x.
```

In principle, one could extract these constants to their corresponding Haskell counterparts, thus recovering the behavior of the Liquid Haskell implementation.

5 Case Study: Correctness of Parallel String Matching in Liquid Haskell

In this section we apply the parallelization equivalence theorem of § 3 to parallelize a realistic, efficient string matcher. We define a string matching function `toSM :: RString → SM tg` from Refined Strings `RString` to a monoidal, string matching data structure `SM tg`. In § 5.1 we assume that `toSM`'s domain, *i.e.,* the Refined String that is a wrapper of Haskell's optimized `ByteString`, is a chunkable monoid. Then, in § 5.2 we prove that `toSM`'s range, *i.e.,* `SM tg`, is a monoid and in § 5.3 we prove that `toSM` is a morphism. Finally, in § 5.4, we parallelize `toSM` by an application of the parallel morphism function § 3.4.

5.1 Strings are assumed to be Chunkable Monoids

We define the type `RString` to be Haskell's existing, optimized, constant-indexing `ByteString` (or `BS`).

```
type RString = BS.ByteString
```

Similarly, we wrap the existing `ByteString` functions that are required by chunkable monoids.

```
η = BS.empty lenStr          x = BS.length x
x ⊡ y = x `BS.append` y      takeStr i x = BS.take i x
                             dropStr i x = BS.drop i x
```

We *axiomatize* the above wrapper functions to satisfy the properties of chunkable monoids. For instance, we define a logical uninterpreted function ⊡ and relate it to the Haskell ⊡ function via an assumed (unchecked) type.

```
assume (⊡) :: x:RString → y:RString
  → {v:RString | v = x ⊡ y}
```

Then, we use the uninterpreted function ⊡ in the logic to assume monoid laws, like associativity.

```
assume assocStr :: x:RString → y:RString
  → z:RString → {x ⊡ (y ⊡ z) = (x ⊡ y) ⊡ z}
```

We extend the above axiomatization for the rest of the chunkable monoid requirements and conclude that `RString` is a chunkable monoid following the Definition 3.1,

Assumption 1 (RString is a Chunkable Monoid). *(`RString`, η, ⊡) combined with the methods `lenStr`, `takeStr`, `dropStr` and the proof term `takeDropPropStr` is a chunkable monoid.*

We note that actually proving that `ByteString` implements a chunkable monoid in Liquid Haskell is possible, as implied by [25], but it is both time consuming and orthogonal to our purpose. Instead, here we follow the easy route of axiomatization – demonstrating that Liquid Haskell verification can be gradual.

5.2 String Matching Monoid

String matching amounts to determining all the indices in a source string where a given target string begins; for example, for source string ababab and target aba the results of string matching would be `[0, 2]`.

We now define a suitable monoid, `SM tg`, for the codomain of a string matching function, where `tg` is the target string.

An index `i` is a good index on the string `input` for the `target` if `target` appears in the position `i` of `input`. We capture this notion of "goodness" using a refinement type alias `GoodIndex` on the values `I` and `T` (in Liquid Haskell's type definitions arguments starting with upper (lower) case letters stand for value (type) parameters).

```
type GoodIndex I T = {i:Nat | isGoodIndex I T i }

isGoodIndex input tg i
  = subString i (lenStr tg) input  == tg
  ∧ i + lenStr tg ≤ lenStr input

subString o l = takeStr l . dropStr o
```

We define the data type `SM target` to contain a refined string field `input` and a list field `indices` of `input`'s good indices for `target`. (For simplicity we use Haskell's built-in lists to refer to the reflected `List` type of § 2.)

```
data SM (tg :: Symbol) where
  SM :: input:  RString
     → indices:[GoodIndex input (fromString tg)]
     → SM tg
```

We use the string type literal [3] to parameterize the string matcher over the target being matched. This encoding turns the string matcher into a monoid as the type checker can statically ensure that only matches on the same target can be appended together.

Next, we define the monoid identity and mappend methods for string matching.

The *identity method* ϵ of `SM target`, for each `target`, returns the identity string (η) and the identity list (`[]`).

```
ϵ :: ∀ (target :: Symbol). SM target
ϵ = SM η []
```

The *mappend method* (◇) of `SM tg` is explained in Figure 2, where the two string matchers `SM x xis` and `SM y yis` are appended. The returned input field is just `x ⊡ y`, while the returned indices field appends three list of indices: 1) the indices `xis` on `x` casted to be good indices of the new input `x ⊡ y`, 2) the new indices `xyis` created when concatenating the two input strings, and 3) the indices `yis` on `y`, shifted right `lenStr x` units. The Haskell

[3] `Symbol` is a kind and `target` is effectively a singleton type (see GHC.TypeLits in Hackage

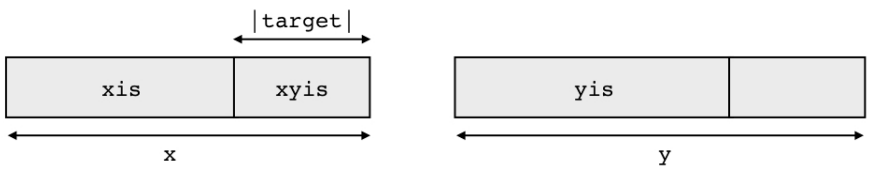

Figure 2. Mappend indices of String Matcher

definition of ◇ captures the above three indexing operations.

```
(◇) :: ∀ (tg::Symbol). KnownSymbol tg
    ⇒ SM tg → SM tg → SM tg
(SM x xis) ◇ (SM y yis)
  = SM (x ⊡ y) (xis' ++ xyis ++ yis')
  where
   vtg = fromString (symbolVal (Proxy :: Proxy tg))
   xis' = map (castGoodIndex vtg x y) xis
   xyis = makeNewIndices x y vtg
   yis' = map (shiftStringRight vtg x y) yis
```

Capturing the target tg as a type parameter is critical for the Haskell type system to specify that both arguments of (◇) are string matchers on the same target. Next, we explain the details of the three indexing operations, namely 1) *casting* the left old indices, 2) *creating* new indices, and 3) *shifting* of the old right indices.

1) Cast Good Indices If i is a good index for the string x on the target tg, then i is also a good index for the string x ⊡ y and the same target, for any y. This property cannot be automatically proven by Liquid Haskell, instead it is explicitly encoded in the function castGoodIndex.

```
castGoodIndex
  :: tg:RString → x:RString → y:RString
  → i:GoodIndex x tg
  → {v:GoodIndex (x ⊡ y) tg | v = i}
castGoodIndex tg x y i
  = subStrAppendRight x y (lenStr tg) i `cast` i
```

The definition of castGoodIndex is a refinement type cast on the argument i, using the assumed string property that appending any string y to the string x preserves the substrings of x between i and j, when i + j does not exceed the length of x.

```
assume subStrAppendRight
   :: x:RString → y:RString → j:Integer
   →  i:{Integer | i + j ≤ lenStr x }
   →  { subString x i j = subString (x ⊡ y) i j }
```

Refinement type casting is performed safely via the function cast p x that returns x and enforces the properties of p in the logic.

```
cast :: b → x:a → {v:a | v = x}
cast _ x = x
```

In the logic, cast p x is reflected as x, allowing p to be any arbitrary (*i.e.,* non-reflected) Haskell expression.

2) Creation of new indices Appending two input strings x and y may create new good indices, *i.e.,* the indices xyis in Figure 2. For instance, appending "ababcab" with "cab" leads to a new occurrence of "abcab" at index 5. These new good indices can appear only at the last lenStr tg - 1 positions of the left input x. The function makeNewIndices detects all such good new indices.

```
makeNewIndices :: x:RString → y:RString
  → tg:RString → [GoodIndex {x ⊡ y} tg]
makeNewIndices x y tg
  | lenStr tg < 2 = []
  | otherwise     = makeIndices (x ⊡ y) tg lo hi
  where
    lo = maxInt (lenStr x - (lenStr tg - 1)) 0
    hi = lenStr x - 1
```

If the length of the tg is less than 2, then no new good indices can be created. Otherwise, the call on makeIndices returns all the good indices of the input x ⊡ y for target tg in the range from maxInt (lenStr x - (lenStr tg - 1)) 0 to lenStr x - 1.

Generally, makeIndices s tg lo hi returns the good indices of the input string s for target tg in the range from lo to hi by recursively checking "goodness" of all the indices from lo to hi.

```
makeIndices :: s:RString → tg:RString
  → lo:Nat → hi:Integer → [GoodIndex s tg]
makeIndices s tg lo hi
  | hi < lo            = []
  | isGoodIndex s tg lo = lo:rest
  | otherwise          = rest
  where
    rest = makeIndices s tg (lo + 1) hi
```

Note that makeNewIndices does not scan all the input x and y, instead only scans at most lenStr tg positions. Thus, the time complexity to create the new indices is linear on the size of the target but independent of the size of the input, allowing parallelization of string matching to lead to runtime speedups.

3) Shift Good Indices If i is a good index for the string y on the target tg, then shifting i right lenStr x units gives a good index for the string x ⊡ y on tg. This property is encoded in the function shiftStringRight.

```
shiftStringRight :: tg:RString → x:RString
  → y:RString → i:GoodIndex y tg
  → {v:(GoodIndex (x ⊡ y) tg) | v = i + lenStr x}
shiftStringRight tg x y i
  = subStrAppendLeft x y (lenStr tg) i
     `cast` i + lenStr x
```

The definition of shiftStringRight performs the appropriate index shifting and casts the refinement type of the shifted index. Type casting uses the assumed property on strings that substrings are preserved on left appending, *i.e.,* the substring of y from i of size j is equal to the substring of x ⊡ y from lenStr x + i of size j.

```
assume subStrAppendLeft :: x:RString → y:RString
  → j:Integer → i:Integer →
  {subStr y i j = subStr (x ⊡ y) (lenStr x + i) j}
```

5.2.1 String Matching is a Monoid

Next we prove that the methods ϵ and (◇) satisfy the monoid laws.

Theorem 5.1 (SM is a Monoid)**.** *(*`SM t`*, ϵ, ◇) is a monoid.*

Proof. We prove that string matching is a monoid by providing safe proof terms for the monoid laws of Definition 2.1:

```
idLeft  :: x:SM t → {ϵ ◇ x = xs}
idRight :: x:SM t → {x ◇ ϵ = x}
assoc   :: x:SM t → y:SM t → z:SM t
        → { x ◇ (y ◇ z) = (x ◇ y) ◇ z}
```

First, we prove *left identity* using PLE, left identity on string and list and two helper lemmata.

```
idLeft (SM i is)
  =  idLeftStr i ∧. idLeftList is
  ∧. mapShiftZero tg i is ∧. newIsNullLeft i tg
  where
    tg = fromString (symbolVal (Proxy :: Proxy t))
```

The first helper lemma states that shifting indices by the length of the empty string is an identity which is proven by induction on the index list `is`.

```
mapShiftZero :: tg:RString → i:RString
  → is:[GoodIndex i target]
  → {map (shiftStringRight tg η i) is = is}
```

The second helper lemma states than appending with the empty string creates no new indexes, as the new indexes would belong into the empty range from `0` to `-1`.

```
newIsNullLeft :: s:RString → t:RString
  → {makeNewIndices η s t = []}
```

Similarly, we prove *right identity* using two helper lemmata that encode that casting is an identity and that appending with the empty string creates no new indexes.

Finally we prove *associativity* by showing equality of the left (`(x ◇ y) ◇ z`) and right (`x ◇ (y ◇ z)`) associative string matchers. To prove equality of the two string matchers we show that the input and indices fields are respectively equal. Equality of the input fields follows by associativity of RStrings. To prove equality of the index list we observe that irrespective of the mappend precedence, the indices can be split in five groups: the indices of the input `x`, the new indices from mappending `x` and `y`, the indices of the input `y`, the new indices from mappending `y` and `z`, and the indices of the input `z`. After this observation the proof proceeds in three steps. First, we group the indices in the five lists indices using list associativity and distribution of index shifting. Then, we prove equivalence of different group representations, since the representation of each group depends on the order of appending. Finally, we wrap the index groups back to string matchers using list associativity and distribution of casts. □

5.3 String Matching Monoid Morphism

Next, we define the function `toSM` which computes the string matcher for the input string on the type level target.

```
toSM :: ∀ (tg :: Symbol). (KnownSymbol tg)
     ⇒ RString → SM tg
toSM input = SM input (go input tg')
  where
    tg' = fromString (symbolVal (Proxy :: Proxy tg))
    go x tg = makeIndices x tg 0 (lenStr x - 1)
```

We prove in [28] that `toSM` is a monoid morphism.

Theorem 5.2. *The function* `toSM` *is a morphism between the monoids (*`RString`*, η, ⊡) and (*`SM t`*, ϵ, ◇); since the below function* `morphismtoSM` *has a valid inhabitant.*

```
morphismtoSM :: x:RString → y:RString →
  { toSM η = ϵ ∧ toSM (x ⊡ y) = toSM x ◇ toSM y}
```

5.4 Parallel String Matching

Finally, we define `toSMPar` as a parallel version of `toSM`, using machinery of section 3, and prove that the sequential and parallel versions always give the same result.

```
toSMPar :: ∀ (tg::Symbol). (KnownSymbol tg)
        ⇒ Integer → Integer → RString → SM tg
toSMPar i j = pmconcat i . pmap toSM . chunkStr j
```

First, `chunkStr` splits the input into chunks of size `j`. Then, `pmap` applies `toSM` at each chunk in parallel. Finally, `pmconcat` concatenates the mappend chunks in parallel using `(◇)`, the monoidal operation for `SM tg`. Correctness of `toSMPar` directly follows from Theorem 3.3.

Theorem 5.3 (Correctness of Parallel String Matching)**.** *For each parameter* `i` *and* `j`*, and input* `x`*,* `toSMPar i j x` *is always equal to* `toSM x`*.*

```
correctness :: i:Integer → j:Integer → x:RString
            → {toSM x = toSMPar i j x}
```

Proof. The proof follows by direct application of Theorem 3.3 on the chunkable monoid (`RString`, η, ⊡) (by Assumption 1) and the monoid (`SM t`, ϵ, ◇) (by Theorem 5.1).

```
correctness i j x
  =   toSMPar i j x
  ==. pmconcat i (pmap toSM (chunkStr j x))
  ==. toSM x
   ∴ parallelismEq toSM morphismtoSM x i j *** QED
```

Note that application of the theorem `parallelismEq` requires a proof that its first argument `toSM` is a morphism. By Theorem 3.3, the required proof is provided as the function `morphismtoSM`. □

6 String Matching in Coq

In this section we present the highlights of replicating the Liquid Haskell proof of correctness for the parallelization of a string matching algorithm into Coq.

6.1 Efficient vs Verified Library Functions

In Liquid Haskell we used a wrapper around `ByteString`s to represent efficient but unverified string manipulation functions. Thus, we *assumed* that the `ByteString` functions satisfy the monoid laws. On the contrary, our Coq proof used the verified but inefficient, built-in implementation of `String`s. We relied on the library theorems to prove most of the required `String` properties, while we still admitted theorems not directly provided by the library (*e.g.*, the interoperation between take and drop). Although Coq does not directly provide optimized libraries, one can achieve runtime efficiency by extracting e.g. String to ByteString at runtime.

6.2 Executable vs Inductive Specifications

In Liquid Haskell refinements on types constitute a decidable, provably terminating, boolean subset of Haskell values, *i.e.,* refinements can be executed at runtime returning either True or False. For example, using the GoodIndex type alias of § 5.2, if Liquid Haskell decides that i is a good index on the input for the target (*i.e.,* i :: GoodIndex input target), then isGoodIndex input target i provably returns True at runtime. On the other hand, Coq distinguishes between the logical (**Prop**) and the executable (**Type**) portions of the code. This separation both facilitates reasoning on the logical code and allows for a clean extraction procedure, but introduces difficulties when the logical specifications also need to be executed. For example, we can define isGoodIndex to live in **Prop**[4].

```
Definition isGoodIndex in tg i
  := substring i (length tg) in = tg.
```

In order to *test* whether a given index i is a good index for some given input and target strings, we need a decidability (*i.e.,* executable) procedure for isGoodIndex.

```
Definition isGoodIndexDec input tg i:
  {isGoodIndex input tg i} +
  {~ (isGoodIndex input tg i)}.
```

Instead of returning a simple boolean, the decidability procedure returns a proof carrying, executable sum that also contains additional content to construct appropriate proof terms.

6.3 Intrinsic vs Extrinsic Verification

In § 4.1 we already discussed how Liquid Haskell favors intrinsic while Coq favor extrinsic verification. In the intrinsic, Liquid Haskell world the specifications come embedded into the functions and data types, while in Coq's extrinsic world specifications and definitions are clearly separated. In the string matching proof we run into the case where intrinsic verification was unavoidable in Coq, leading to (syntactic) proof equivalence obligations that could only be resolved via the axiom of proof irrelevance.

The Liquid Haskell Approach In § 5.2 we defined the Liquid Haskell string matcher SM tg to contain an input and the list of indices, *i.e.,* a list intrinsically refined to contain only indices that are good for input on the target. This intrinsic specification assures that each string matcher only contains valid indices while the validity proof is not a Haskell object, instead is externally performed by the SMT solver.

The Extrinsic Approach When porting the string matching proof to Coq, to keep implementation clean from proofs, we followed an extrinsic approach. We defined the string matcher data type on a target tg to contain the input string and any list of natural numbers as indices.

```
Inductive SM (tg : string) :=
  | Sm : ∀ (in : string) (is : list nat), SM tg.
```

[4]A different approach would be to define isGoodIndex as a boolean computation and then use ssreflect's *views* to obtain convenient elimination principles. We opted for the logical approach to better highlight the prover's differences.

Extrinsically, we specified that a string matcher SM tg is valid when the indices list contains only valid indices.

```
Inductive validSM tg : SM tg → Prop
```

With the above extrinsic definition of the String Matcher, the associativity property of (◇) does not hold, as the property explicitly requires the middle string matcher to be valid:

```
Theorem sm_assoc tg (sm1 sm2 sm3 : SM tg) :
  validSM tg sm2 →
  sm1 ◇ (sm2 ◇ sm3) = (sm1 ◇ sm2) ◇ sm3.
```

Thus, the extrinsic (◇) does not satisfy the associativity monoid law, as it comes with the extra validity assumption.

The Intrinsic Approach requires Proof Irrelevance To define an associative mappend string matching operator we intrinsically restrict the type of sm to carry a proof of valid indices.

```
Inductive sm tg : Type :=
| mk_sm : ∀ in is,
   Forall (isGoodIndex in tg) is → sm tg.
```

Extending the string matching sm to carry validity proofs implies that two string matchers are equal only when their respective proofs are *syntactically* equal. To discharge the proof equality obligation, we accept two string matchers to be equal irrespective of equality on their proof terms.

```
Lemma proof_irrelevant_equality
  tg xs xs' l H l' H' : xs = xs' → l = l'
  → mk_sm tg xs l H = mk_sm tg xs' l' H'.
```

We prove the above lemma using *Proof Irrelevance*, an admittable axiom, consistent with Coq's logic, which states that any two proofs of the same property are equal. Thus, in the Coq proof intrinsic reasoning (used to prove associativity) required the assumption of proof irrelevance. On the contrary in Liquid Haskell's proof, specifications are intrinsically embedded in the definitions but their proofs are automatically and externally constructed by the SMT solver. In Liquid Haskell the user does not have access to the automatically generated proof terms, *i.e.,* proof equality cannot even be specified (and is never required).

7 Evaluation

7.1 Quantitative Comparison.

Table 1 summarizes the quantitative evaluation of our two proofs as implemented in [28]: the generalized equivalence property of parallelization of monoid morphisms and its application on the parallelization of a naïve string matcher. We used three provers to conduct our proofs: Coq, Liquid Haskell, and Liquid Haskell extended with the PLE (Proof by Static Evaluation § 2.3) heuristic. The Liquid Haskell proof was originally specified and verified by the first author within 2 months. Most of this time was spent on iterating between incorrect implementations of the string matching implementation (and the proof) based on Liquid Haskell's type errors. After the Liquid Haskell proof was finalized, it was ported to Coq by the second author within 2 weeks. We note that the proofs were neither optimized for size nor for verification time.

Verification time. We verified our proofs using a machine with an Intel Core i7-4712HQ CPU and 16GB of RAM. Verification in Coq is the fastest requiring 38 sec in total. Liquid Haskell requires x2.5

Table 1. Quantitative evaluation. We report verification **Time** (in seconds) and LoC required to verify monoid morphism **parallelization** and its application to the **string matcher**. We split proofs of Coq (1136 LoC in total), Liquid Haskell (1428 LoC in total) and Liquid Haskell with PLE (1134 LoC in total) into **specifications**, **proof terms** and **executable** code.

Property	Coq				Liquid Haskell				Liquid Haskell + PLE			
	Time	Spec	Proof	Exec	Time	Spec	Proof	Exec	Time	Spec	Proof	Exec
Parallelization	5	121	329	39	8	54	164	78	5	62	73	78
String Matcher	33	127	437	83	87	199	831	102	1287	223	596	102
Total	38	248	766	122	95	253	995	180	1292	285	669	180

as much time while it needs x34 time using PLE. This slowdown is expected given that, unlike Coq that is checking the proof, Liquid Haskell uses the SMT solver to synthesize proof terms during verification, while PLE is an under-developed, non-optimized approach to heuristically synthesize proof terms by static evaluation. In small proofs, like the generalized parallelization theorem, PLE can speedup verification time as proofs are quickly synthesized due to the fewer reflected functions and smaller proof terms.

Verification size. We split the total lines of code into three categories for both Coq and Liquid Haskell.

- **Spec** represents the theorem and lemma definitions, and the refinement type specifications, resp..
- **Proofs** represents the Coq proof scripts and the Haskell proof terms (*i.e.,* `Proof` resulting functions), resp..
- **Exec** represents the executable portion of the code.

Counting both specifications and proofs as verification code, we conclude that in Coq the proof requires 8x the lines of the executable code, mostly required to deal with the non-structural recursion. This ratio drops to 7x for Liquid Haskell, because the executable code in the Haskell implementation is increased to include a basic string matching interface for testing the application. Finally, the ratio drops to 5x with the PLE heuristic, as the proof terms are shrunk without any modification to the executable portion.

Evaluation of PLE. PLE is used to synthesize non-sophisticated proof terms, leading to fewer lines of proof code but slower verification time. We used PLE to synthesize 31 out of the 43 total number of proof terms. PLE failed to synthesize the rest proof terms due to: 1.*incompleteness:* PLE is unable to synthesize proof terms when the proof structure does not follow the structure of the reflected functions, or 2. *verification slowdown:* in big proof terms there are many intermediate terms to be evaluated which dreadfully slows verification. Formalization and optimization of PLE, so that it synthesizes more proof terms faster, is left as future work.

7.2 Qualitative Comparison.

We summarize the essential differences in theorem proving using Liquid Haskell versus Coq based on our experience (§ 4 and § 6). These differences validate and illustrate the distinctions that have been previously [3, 22, 23] described between the two provers.

Theorem Provers vs. Proof Assistants. Coq is not only a theorem prover, but a proof assistant that provides a semi-interactive proving environment to explain failing proofs. Liquid Haskell, on the other hand, is designed as an automated refinement type checker. Thus it is agnostic to the specific application of theorem proving providing no interactive environment to aid proof generation. In case of failure, Liquid Haskell provides the exact source location of the failing theorem, but will not currently attempt any Coq-like subgoal analysis to assist theorem proving.

General Purpose vs. Verification Specific Languages. Haskell is a general purpose language with concurrency support and optimized libraries (*e.g.,* `Bytestring`, `parallel`) that can be used (§ 4.5) to build real applications. Coq provides minimal support for such features: dealing with essential non-structural recursion patterns is inconvenient while access to parallel primitives can only be gained through extraction. However, unlike Liquid Haskell, Coq comes with a large standard library of theorems and tactics that ease the burden of the prover (§ 4.2 and § 6.1). Finally, Coq's trusted computing base (TCB) is just it's typechecker, while Liquid Haskell's TCB contains GHC's type inference, Liquid Haskell constraint generation and the SMT solver itself.

SMT-automation vs. Tactics. Liquid Haskell uses an SMT-solver to automate proofs over decidable theories (such as linear arithmetic, uninterpreted functions); which reduces the proof burden but increases the verification time. On the other hand, Coq users enjoy some level of proof automation via library or hand-crafted tactics, but even sophisticated decidability procedures, like `omega` for Presburger arithmetic, have incomplete implementations and produce large, slow-to-check proof terms (§ 4.3).

Intrinsic vs. Extrinsic verification. Liquid Haskell naturally uses intrinsic verification; *i.e.,* specifications are embedded in the definitions of the functions, are proved (automatically by SMTs) at function definitions, and are assumed at function calls. Coq naturally uses extrinsic verification to separate the functionality of definitions from their specifications. The specifications can then be independently proven (§ 4.1), making function definitions cleaner.

Semantic vs. Syntactic Termination Checking. Liquid Haskell uses a semantics termination checker that proves termination given a wellfounded termination metric. On the contrary, Coq allows fixpoints to be defined only by using syntactical subterms of some principal argument in recursive calls, requiring advanced transformation techniques (§ 4.4) for definitions outside of this restrictive recursion pattern.

8 Related Work

SMT-Based Verification SMT solvers have been used to automate reasoning on verification oriented languages like Dafny [13], F* [23] and Why3 [8]. Designed for verification, there languages offer limited support for the advanced language features – like parallelism and optimized libraries – that we use in our verified implementation. All these languages allow for highly expressive specifications, which makes SMT verification undecidable in theory [6] and unstable in practice [14]. Refinement types [11] on the other hand, extend existing general purpose languages with decidable specifications. That is, without refinement reflection [26], refinement types only allow "shallow" program specifications, *i.e.,* properties that only talk about abstractions of program functions but not functions themselves.

Dependent Types On the other hand, dependent type systems, like Coq [2], Adga [19] and Isabelle/HOL [20], allow for "deep" specifications which talk about program functions, such as the equivalence reasoning we presented. These systems allow for tactics and heuristics that aid proof generation but lack SMT automations and general-purpose language features, like non-termination. Zombie [3] and F* [23] allow dependent types to co-exist with divergent and effectful programs, but still lack the optimized libraries, like `ByteString`, which come with mature languages like Haskell.

Haskell itself is becoming a dependently typed language. Eisenberg [7] aims to make type-level computations as expressive as term-level computations. Though expressive enough, dependent Haskell does not provide SMT- nor tactic-based automation, making realistic theorem proving, *e.g.,* our 1136 LoC tactic-aided Coq proof, unapproachable [16]. In the future, we would like to combine Haskell's dependent types with Liquid Haskell's automation towards an expressive and usable prover. In fact, our monoid string matcher proof already depends on Haskell's type level strings.

Parallel Code Verification Dependent type theorem provers have been used before to verify parallel code. BSP-Why [10] is an extension to Why2 that is using both Coq and SMTs to discharge user specified verification conditions. Swierstra [24] formalized mutable arrays in Agda to reason about distributed maps and sums. Finally, on a work closely related to ours, SyDPaCC [17] is a Coq library that automatically parallelizes list homomorphisms by extracting parallel OCaml versions of user provided Coq functions. SyDPaCC used maximum prefix sum as a case study, whose morphism verification is simpler than string matching. Compared to our 1428 LoC Liquid Haskell executable and verified code, the SyDPaCC implementation uses three different languages: 2K lines of Coq, 600 lines of OCaml and 120 lines of C, and is considered "very concise". However, they actually extract a parallel version to OCaml while our Coq development would require similar additional non-Coq code if we were to extract it to obtain an executable program.

9 Conclusion

We used Liquid Haskell as a theorem prover to verify parallelization of monoid morphisms and specifically a realistic string matcher. We ported our 1428 LoC proof to Coq (1136 LoC) and compared the two provers. We conclude that the strong point of Liquid Haskell as a theorem prover is that the proof refers to executable Haskell code while being SMT-automated over decidable theories (like linear arithmetic). On the other hand, Coq aids verification providing a semi-interactive proving environment and a large pool of already developed theorems, tactics, and methodologies that the user can lean on. The development of Coq-like proving environment, library theorems, and proof automation techniques is feasible and is required to establish Liquid Haskell as a usable theorem prover.

Acknowledgments

We thank Antal Spector-Zabusky, Arthur Azevedo de Amorim, George Karachalias, Nodari Sitchinava, Ranjit Jhala, Valentin Robert, and the anonymous reviewers for their insightful comments. This work was partialy supported by Awake Networks and NSF awards #1421243 *Random Testing for Language Design* and #1521523 *Expeditions in Computing: The Science of Deep Specification.*

References

[1] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. 2010.
[2] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions.* Springer Verlag, 2004.
[3] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. In *POPL*, 2014.
[4] A. Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press, 2013.
[5] T. Coq development team. *The Coq proof assistant reference manual*, 2009. URL http://coq.inria.fr/doc/.
[6] L. de Moura and N. Bjorner. Efficient E-matching for Smt Solvers. In *CADE*, 2007.
[7] R. A. Eisenberg. *Dependent Types in Haskell: Theory and Practise.* PhD thesis, UPenn, 2016.
[8] J. Filliâtre and A. Paskevich. Why3 – Where Programs Meet Provers. In *ESOP*, 2013.
[9] C. Flanagan. Hybrid type checking. In *POPL*, 2006.
[10] J. Fortin and F. Gava. BSP-Why: A tool for deductive verification of BSP algorithms with subgroup synchronisation. In *Int J Parallel Prog*, 2015.
[11] T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, 1991.
[12] G. Gonthier and A. Mahboubi. A small scale reflection extension for the Coq system. Technical report, Microsoft Research INRIA, 2009.
[13] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. LPAR, 2010.
[14] K. R. M. Leino and C. Pit-Claudel. Trigger selection strategies to stabilize program verifiers. 2016.
[15] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, 2006.
[16] S. Lindley and C. McBride. Hasochism: the pleasure and pain of dependently typed haskell programming. In *Haskell*, 2013.
[17] F. Loulergue, W. Bousdira, and J. Tesson. Calculating Parallel Programs in Coq using List Homomorphisms. In *International Journal of Parallel Programming*, 2016.
[18] M. Moskal, J. Lopuszański, and J. R. Kiniry. E-matching for Fun and Profit. In *Electron. Notes Theor. Comput. Sci.*, 2008.
[19] U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers, 2007.
[20] L. C. Paulson. Isabelle - A Generic Theorem prover. *Lecture Notes in Computer Science*, 1994.
[21] W. Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, 1991.
[22] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.
[23] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *POPL*, 2016.
[24] W. Swierstra. More dependent types for distributed arrays. *Higher-Order and Symbolic Computation*, 2010.
[25] N. Vazou. *Liquid Haskell: Haskell as a theorem prover.* PhD thesis, UCSD, 2016.
[26] N. Vazou and R. Jhala. Refinement Reflection. arXiv:1610.04641, 2016.
[27] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement Types for Haskell. In *ICFP*, 2014.
[28] N. Vazou, L. Lampropoulos, and J. Polakow. Implementation. 2017. https://github.com/nikivazou/verified_string_matching.