Verified Parallel String Matching in Haskell

Niki Vazou[1,2] and Jeff Polakow[2]

[1]UC San Diego [2]Awake Networks

Abstract. In this paper, we prove correctness of parallelizing a string matcher using Haskell as a theorem prover. We use refinement types to specify correctness properties, Haskell terms to express proofs and Liquid Haskell to check correctness of proofs. First, we specify and prove that a class of monoid morphisms can be parallelized via parallel monoid concatenation. Then, we encode string matching as a morphism to get a provably correct parallel transformation. Our 1839LoC prototype proof shows that Liquid Haskell can be used as a fully expressive theorem prover on realistic Haskell implementations.

1 Introduction

In this paper, we prove correctness of parallelization of a naïve string matcher using Haskell as a theorem prover. We use refinement types to specify correctness properties, Haskell terms to express proofs and Liquid Haskell to check correctness of proofs.

Optimization of sequential functions via parallelization is a well studied technique [14,4]. Paper and pencil proofs of program have been developed to support the correctness of the transformation [6]. However, these paper and pencil proofs show correctness of the parallelization algorithm and do not reason about the actual implementation that may end up being buggy.

Dependent Type Systems (like Coq [3] and Adga [19]) enable program equivalence proofs for the actual implementation of the functions to be parallelized. For example, SyDPaCC [18] is a Coq extension that given a naïve Coq implementation of a function, returns an Ocaml parallelized version with a proof of program equivalence. The limitation of this approach is that the initial function should be implemented in the specific dependent type framework and thus cannot use features and libraries from one's favorite programming language.

Refinement Types [8,12,24] on the other hand, enable verification of existing general purpose languages (including ML [34,2,22], C [7,23], Haskell [31], Racket [15] and Scala [26]). Traditionally, refinement types are limited to "shallow" specifications, that is, they are used to specify and verify properties that only talk about behaviors of program functions but not functions themselves. This restriction critically limits the expressiveness of the specifications but allows for automatic SMT [1] based verification. Yet, program equivalence proofs were out of the expressive power of refinement types.

Recently, we extended refinement types with Refinement Reflection [32], a technique that reflects each function's implementation into the function's type

and is implemented in Liquid Haskell [31]. We claimed that Refinement Reflection can turn any programming language into a proof assistant. In this paper we check our claim and use Liquid Haskell to prove program equivalence. Specifically, we *define in Haskell* a sequential string matching function, `toSM`, and its parallelization, `toSMPar`, using existing Haskell libraries; then, we *prove in Haskell* that these two functions are equivalent, and we check our proofs using Liquid Haskell.

Theorems as Refinement Types Refinement Types refine types with properties drawn from decidable logics. For example, the type `{v:Int | 0 < v}` describes all integer values `v` that are greater than `0`. We refine the unit type to express theorems, define unit value terms to express proofs, and use Liquid Haskell to check that the proofs prove the theorems. For example, Liquid Haskell accepts the type assignment `() :: {v:()| 1+1=2}`, as the underlying SMT can always prove the equality `1+1=2`. We write `{1+1=2}` to simplify the type `{v:()| 1+1=2}` from the irrelevant binder `v:()`.

Program Properties as Types The theorems we express can refer to program functions. As an example, the type of `assoc` expresses that $\diamondsuit$ is associative.

```
assoc :: x:m → y:m → z:m → {x ♢ (y ♢ z) = (x ♢ y) ♢ z}
```

In § 2 we explain how to write Haskell proof terms to prove theorems like `assoc` by proving that list append `(++)` is associative. Moreover, we prove that the empty list `[]` is the identity element of list append, and conclude that the list (with `[]` and `(++)`, *i.e.,* the triple (`[a]`, `[]`, `(++)`)) is provably a monoid.

Corectness of Parallelization In § 3, we define the type `Morphism n m f` that specifies that `f` is a *morphism* between two monoids (`n`, η, $\boxdot$) and (`m`, ϵ, $\diamondsuit$), *i.e.,* `f :: n → m` where `f` η `=` ϵ and `f (x ⊡ y) = f x ♢ f y`.

A morphism `f` on a "chunkable" input type can be parallelized by:

1. chunking up the input in `j` chunks (`chunk j`),
2. applying the morphism in parallel to all chunks (`pmap f`), and
3. recombining the mapped chunks via $\diamondsuit$, also in parallel (`pmconcat i`).

We specify correctness of the above transformation as a refinement type.

```
parallelismEquivalence
  :: f:(n → m) → Morphism n m f → x:n → i:Pos → j:Pos
   → {f x = pmconcat i (pmap f (chunk j x))}
```

§ 3 describes the parallelization transformation in details, while Correctness of Parallelization Theorem 4 proves correctness by providing a term that satisfies the above type.

Case Study: Parallelization of String Matching We use the above theorem to parallelize string matching. We define a string matching function `toSM :: RString → toSM target` from a *refined string* to a *string matcher*. A refined string (§ 4.1) is a wrapper around the efficient string manipulation library `ByteString` that moreover assumes various string properties, including the monoid laws. A string matcher `SM target` (§ 4.2) is a data type that contains a refined string and a list of all the indices where the type level symbol `target` appears in the input. We prove that `SM target` is a monoid and `toSM` is a morphism, thus by the aforementioned Correctness of Parallelization Theorem 4 we can correctly parallelize string matching.

To sum up, we present the first realistic proof that uses Haskell as a theorem prover: correctness of parallelization on string matching. Our contributions are summarized as follows

- We explain how theorems and proofs are encoded and checked in Liquid Haskell by formalizing monoids and proving that lists are monoids (§ 2).
- We formalize morphisms between monoids and specify and prove correctness of parallelization of morphisms (§ 3).
- We show how libraries can be imported as trusted components by wrapping `ByteString`s as refined strings which satisfy the monoid laws (§ 4.1).
- As an application, we prove that a string matcher is a morphism between the monoids of refined strings and string matchers, thus we get provably correct parallelization of string matching (§ 4).
- Based on our 1839LoC proof we evaluate the approach of using Haskell as a theorem prover (§ 5).

2 Proofs as Haskell Functions

Refinement Reflection [32] is a technique that lets you write Haskell functions that prove theorems about other Haskell functions and have your proofs machine-checked by Liquid Haskell [31]. As an introduction to Refinement Reflection, in this section, we prove that lists are monoids by

- *specifying monoid laws* as refinement types,
- *proving the laws* by writing the implementation of the law specifications, and
- *verifying the proofs* using Liquid Haskell.

2.1 Reflection of data types into logic.

To start with, we define a List data structure and teach Liquid Haskell basic properties about List, namely, how to check that proofs on lists are *total* and how to encode functions on List into the logic.

The data list definition `L` is the standard recursive definition.

```
data L [length] a = N | C a (L a)
```

With the `length` annotation in the definition Liquid Haskell will use the `length` function to check termination of functions recursive on Lists. We define `length` as the standard Haskell function that returns natural numbers. We lift `length` into logic as a *measure* [31], that is, a *unary* function whose (1) domain is the data type, and (2) body is a single case-expression over the datatype.

```
type Nat = {v:Int | 0 ≤ v}

measure length
length          :: L a → Nat
length N         = 0
length (C x xs) = 1 + length xs
```

Finally, we teach Liquid Haskell how to encode functions on Lists into logic. The flag `"--exact-data-cons"` automatically derives measures which (1) test if a value has a given data constructor, and (2) extract the corresponding field's value. For example, Liquid Haskell will automatically derive the following List manipulation measures from the List definition.

```
isN :: L a → Bool     -- Haskell's null
isC :: L a → Bool     -- Haskell's not . null

selC1 :: L a → a      -- Haskell's head
selC2 :: L a → L a    -- Haskell's tail
```

Next, we describe how Liquid Haskell uses the above measures to automatically reflect Haskell functions on Lists into logic.

2.2 Reflection of Haskell functions into logic.

Next, we define and reflect into logic the two monoid operators on Lists. Namely, the identity element ϵ (which is the empty list) and an associative operator `(♢)` (which is list append).

```
reflect ϵ
ϵ :: L a
ϵ = N

reflect (♢)
(♢) :: L a → L a → L a
N         ♢ ys = ys
(C x xs) ♢ ys = C x (xs ♢ ys)
```

The reflect annotations lift the Haskell functions into logic in three steps. First, check that the Haskell functions indeed terminate by checking that the `length` of the input list is decreasing, as specified in the data list definition. Second, in the logic, they define the respective uninterpreted functions ϵ and `(♢)`. Finally, the Haskell functions and the logical uninterpreted functions are related by strengthening the result type of the Haskell function with the definition of the

function's implementation. For example, with the above `reflect` annotations, Liquid Haskell will *automatically* derive the following strengthened types for the relevant functions.

```
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 (selC1 xs) (selC2 xs ◇ ys)
        }
```

2.3 Specification and Verification of Monoid Laws

Now we are ready to specify the monoid laws as refinement types and provide their respective proofs as terms of those type. Liquid Haskell will verify that our proofs are valid. Note that this is exactly what one would do in any standard logical framework, like LF [13].

The type `Proof` is defined as an alias of the unit type (`()`) in the library `ProofCombinators` that comes with Liquid Haskell. Figure 1 summarizes the definitions we use from `ProofCombinators`. We express theorems as refinement types by refining the `Proof` type with appropriate refinements. For example, the following theorem states the ϵ is always equal to itself.

```
trivial :: {ϵ = ϵ}
```

Where `{ϵ = ϵ}` is a simplification for the `Proof` type `{v:Proof | ϵ = ϵ}`, since the binder `v` is irrelevant, and `trivial` is defined in `ProofCombinators` to be unit. Liquid Haskell will typecheck the above code using an SMT solver to check congruence on ϵ.

Definition 1 (Monoid). *The triple (`m`, ϵ, $\diamondsuit$) is a monoid (with identity element ϵ and associative operator $\diamondsuit$), 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}
```

Using the above definition, we prove that our list type `L` is a monoid by defining Haskell proof terms that satisfy the above monoid laws.

Left Identity is expressed as a refinement type signature that takes as input a list `x:L a` and returns a `Proof` type refined with the property $\epsilon \diamondsuit$ `x = x`

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

```
type Proof = ()
data QED   = QED

trivial :: Proof
trivial = ()

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

(***) :: a → QED → Proof
_ *** _ = ()

(∴) :: (Proof → a) → Proof → a
f ∴ y = f y
```

Fig. 1. Operators and Types defined in `ProofCombinators`

We prove left identity using combinators from `ProofCombinators` as defined in Figure 1. We start from the left hand side `empty ♢ x`, which is equal to `N ♢ x` by calling `empty` thus unfolding the equality `empty = N` into the logic. Next, the call `N ♢ x` unfolds into the logic the definition of `(♢)` on `N` and `x`, which is equal to `x`, concluding our proof. Finally, we use the operators `p *** QED` which basically 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.

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

```
idRight :: x:L a → { x ♢ ϵ = x }
idRight N = N ♢ empty ==. N *** QED

idRight (C x xs)
   =   (C x xs) ♢ empty
   ==. C x (xs ♢ empty)
   ==. C x xs ∴ idRight xs
   *** QED
```

The recursive call `idRight xs` is provided as a third optional argument in the `(==.)` operator to justify the equality `xs ♢ empty = xs`, while the operator `(∴)` is merely a function application with the appropriate precedence. Note that LiquiHaskell, via termination and totality checking, is verifying that all the proof terms are well formed because (1) the inductive hypothesis is only applying to smaller terms, and (2) all cases are covered.

Associativity is proved in a very similar manner, using structural induction.

```
assoc   :: x:L a → y:L a → z:L a
        → { x ♢ (y ♢ z) = (x ♢ y) ♢ z}
assoc N y z
  =   N ♢ (y ♢ z)
  ==. y ♢ z
  ==. (N ♢ y) ♢ z
  *** QED

assoc (C x xs) y z
  =  (C x xs) ♢ (y ♢ z)
  ==. C x (xs ♢ (y ♢ z))
  ==. C x ((xs ♢ y) ♢ z) ∴ associativity xs y z
  ==. (C x (xs ♢ y)) ♢ z
  ==. ((C x xs) ♢ y) ♢ z
  *** QED
```

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).

Since our list implementation satisfies the three monoid laws we can conclude that `L a` is a monoid.

Theorem 1. *(`L a`, ϵ, ♢) is a monoid.*

Proof. `L a` is a monoid, as the implementation of `idLeft`, `idRight`, and `assoc` satisfy the specifications of $\texttt{idLeft}_m$, $\texttt{idRight}_m$, and $\texttt{assoc}_m$, with `m = L a`. □

3 Verified Parallelization of Monoid Morphisms

A monoid morphism is a function between two monoids which preserves the monoidal structure; *i.e.,* a function on the underlying sets which preserves identity and associativity. We formally specify this definition using a refinement type `Morphism`.

Definition 2 (Monoid Morphism). *A function `f :: n → m` is a morphism between the monoids (`m`, ϵ, ♢) and (`n`, η, ⊡) if `Morphism n m f` has an inhabitant.*

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

A monoid morphism can be parallelized when its domain can be cut into chunks and put back together again, a property we refer to as chunkable and expand upon in § 3.1. A chunkable monoid morphism is then parallelized by:

1. chunking up the input,

2. applying the morphism in parallel to all chunks, and
3. recombining the chunks, also in parallel, back to a single value.

In the rest of this section we implement and verify to be correct the above transformation.

3.1 Chunkable Monoids

Definition 3 (Chunkable Monoids). *A monoid (m, ϵ, $\diamondsuit$) is chunkable if the following four functions are defined on m.*

```
length_m :: m → Nat

drop_m :: i:Nat → x:MGEq m i → MEq m (length_m x-i)
take_m :: i:Nat → x:MGEq m i → MEq m i

takeDropProp_m :: i:Nat → x:m →
                  {x = take_m i x ◇ drop_m i x}
```

Where the type aliases `MLeq m I` *(and* `MEq m I`*) constrain the monoid* `m` *to have* $\texttt{length}_m$ *greater than (resp. equal) to* `I`*.*

```
type MGEq m I = {x:m | I ≤ length_m x }
type MEq  m I = {x:m | I = length_m x }
```

Note that the "important" methods of chunkable monoids are the `take` and `drop`, while the `length` method is required to give pre- and post-condition on the other operations. Finally, `takeDropProp` provides a proof that for each `i` and monoid `x`, appending `take i x` to `drop i x` will reconstruct `x`.

Using $\texttt{take}_m$ and $\texttt{drop}_m$ we define for each chunkable monoid (`m`, ϵ, $\diamondsuit$) a function $\texttt{chunk}_m$ `i x` that splits `x` in chunks of size `i`.

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

chunkRes_m i x v
  | length_m x ≤ i = length_m v == 1
  | i == 1         = length_m v == length_m xs
  | otherwise      = length_m v < length_m xs
```

The function $\texttt{chunk}_m$ provably terminates as $\texttt{drop}_m$ `i x` will return a monoid smaller than `x`, by the Definition of $\texttt{drop}_m$. The definitions of both $\texttt{take}_m$ and $\texttt{drop}_m$ are also used from Liquid Haskell to verify the $\texttt{length}_m$ constraints in the result of $\texttt{chunk}_m$.

3.2 Parallel Map

We define a parallelized map function `pmap` using Haskell's library `parallel`. Concretely, we use the function `Control.Parallel.Strategies.withStrategy` 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)
```

The strategy `parStrategy` does not affect verification. In our codebase we choose the traversable strategy.

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

Parallelism in the Logic. The function `withStrategy` 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, we need to reflect the function `pmap` and represent its implementation in the logic. Thus, we also need to represent the function `withStrategy` in the logic. LiquidHaskell represents `withStrategy` in the logic as a logical function that merely returns its second argument, `withStrategy _ x = x`, and does not reason about parallelism.

3.3 Monoidal Concatenation

The function $\texttt{chunk}_m$ lets us turn a monoidal value into several pieces. In the other direction, for any monoid `m`, there is a standard way of turning `L m` back into a single `m` [1]

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

For any chunkable monoid `n`, monoid morphism `f :: n → m`, and natural number `i > 0` we can write a chunked version of `f` as

```
mconcat . pmap f . chunk_n i :: n → m.
```

Before parallelizing `mconcat`, we will prove that the previous function is equivalent to `f`.

Theorem 2 (Morphism Distribution). *Let (`m`, ϵ, $\diamondsuit$) be a monoid and (`n`, η, $\boxdot$) be a chunkable monoid. Then, for every morphism `f :: n → m`, every positive number `i` and input `x`, `f x = mconcat (pmap f (chunk_n i x))` holds.*

[1] `mconcat` is usually defined as `foldr mappend mempty`

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

Proof. We prove the theorem by providing an implementation of `morphismDistribution` that satisfies its type. The proof proceeds by induction on the length of the input.

```
morphismDistribution f thm x i
  | length_n x ≤ i
  =   mconcat (pmap f (chunk_n i x))
  ==. mconcat (map f (chunk_n i x))
  ==. mconcat (map f (C x N))
  ==. mconcat (f x `C` map f N)
  ==. f is ♢ mconcat N
  ==. f is ♢ ϵ
  ==. f is ∴ idRight_m (f is)
  *** QED
morphismDistribution f thm x i
  =   mconcat (pmap f (chunk_n i x))
  ==. mconcat (map f (chunk_n i x))
  ==. mconcat (map f (C takeX) (chunk_n i dropX)))
  ==. mconcat (f takeX `C` map f (chunk_n n dropX))
  ==. f takeX ♢ f dropX
      ∴ morphismDistribution f thm dropX i
  ==. f (takeX ⊡ dropX)
      ∴ thm takeX dropX
  ==. f x
      ∴ takeDropProp_n i x
  *** QED
  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 on the input `dropX = drop`$_n$ `i x`, that is provably smaller than `x` as `1 < i`. Then, the fact that `f` is a monoid morphism, as encoded by our assumption argument `thm takeX dropX` we get basic distribution of `f`, that is `f takeX ♢ f dropX = f (takeX ⊡ dropX)`. Finally, we merge `takeX ⊡ dropX` to `x` using the property `takeDropProp`$_n$ of the chunkable monoid `n`. □

3.4 Parallel Monoidal Concatenation

We now parallelize the monoid concatenation by defining a `pmconat i x` function that chunks the input list of monoids and concatenates each chunk in parallel.

We use the `chunk` function of § 3.1 instantiated to `L m` to define a parallelized version of monoid concatenation `pmconcat`.

```
pmconcat :: Int → L m → m
pmconcat i x | i ≤ 1 || length x ≤ i
  = mconcat x
pmconcat i x
  = pmconcat i (pmap mconcat (chunk i x))
```

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

Next, we prove equivalence of parallelized monoid concatenation.

Theorem 3 (Correctness of Parallelization). *Let (m, ϵ, ♢) be a monoid. Then, the parallel and sequential concatenations are equivalent.*

```
pmconcatEquivalence
  :: i:Int → x:L m → { pmconcat i x = mconcat x }
```

Proof. We prove the theorem by providing a Haskell implementation of `pmconcatEquivalence` that satisfies its type. The details of the proof can be found in [33], here we provide the sketch of the proof.

First, we prove that `mconcat` distributes over list splitting

```
mconcatSplit
  :: i:Nat → xs:{L m | i ≤ length xs}
  → { mconcat xs = mconcat (take i xs)
                 ♢ mconcat (drop i xs) }
```

The proofs proceeds by structural induction, using monoid left identity in the base case and monoid associativity associavity and unfolding of `take` and `drop` methods in the inductive step.

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

```
mconcatChunk
  :: i:Pos → xs:L m
  → { mconcat xs = mconcat (map mconcat (chunk i xs)) }
```

The proofs proceeds by structural induction, using monoid left identity in the base case and lemma `mconcatSplit` in the inductive step.

Lemma `mconcatChunk` is sufficient to prove `pmconcatEquivalence` by structural induction, using monoid left identity in the base case. □

3.5 Parallel Monoid Morphism

We can now replace the `mconcat` in our chunked monoid morphism in § 3.3 with `pmconcat` from § 3.4 to provide an implementation that uses parallelism to both map the monoid morphism and concatenate the results.

Theorem 4 (Correctness of Parallelization). *Let* $(m, \epsilon, \diamondsuit)$ *be a monoid and* $(n, \eta, \boxdot)$ *be a chunkable monoid. Then, for every morphism* `f :: n → m`*, every positive numbers* `i` *and* `j`*, and input* `x`*,* `f x = pmconcat i (pmap f (chunk`$_n$ `j x))` *holds.*

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

Proof. We prove the theorem by providing an implementation of `parallelismEquivalence` that satisfies its type.

```
parallelismEquivalence f thm x i j
  =   pmconcat i (pmap f (chunk_n j x))
  ==. mconcat (pmap f (chunk_n j x))
      ∴ pmconcatEquivalence i (pmap f (chunk_n j x))
  ==. f x
      ∴ morphismDistribution f thm x j
  *** QED
```

The proof follows merely by application of the two previous Theorems 2 and 3. □

4 Case Study: Correctness of Parallel String Matching

§ 3 showed that any monoid morphism whose domain is chunkable can be parallelized. We now make use of that result to parallelize string matching. We start by observing that strings are a chunkable monoid. We then turn string matching for a given target into a monoid morphism from a string to a suitable monoid, `SM target`, defined in § 4.2. Finally, in § 4.4, we parallelize string matching by a simple use of the parallel morphism function of § 3.5.

4.1 Refined Strings are Chunkable Monoids

We define a new type `RString`, which is a chunkable monoid, to be the domain of our string matching function. Our type simply wraps Haskell's existing `ByteString`.

```
data RString = RS BS.ByteString
```

Similarly, we wrap the existing `ByteString` functions we will need to show `RString` is a chunkable monoid.

```
η = RS (BS.empty)
(RS x) ⊡ (RS y)= S (x `BS.append` y)

lenStr    (RS x) = BS.length x
takeStr i (RS x) = RS (BS.take i x)
dropStr i (RS x) = RS (BS.take i x)
```

Although it is possible to explicitly prove that `ByteString` implements a chunkable monoid [30], it is time consuming and orthogonal to our purpose. Instead, we just *assume* the chunkable monoid properties of `RString`– thus demonstrating that refinement reflection is capable of doing gradual verification.

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 }
assocStr _ _     = trivial
```

Haskell applications of ⊡ are interpreted in the logic via the logical ⊡ that satisfies associativity via theorem `assocStr`.

Similarly for the chunkable methods, we define the uninterpreted functions `takeStr`, `dropStr` and `lenStr` in the logic, and use them to strengthen the result types of the respective functions. With the above function definitions (in both Haskell and logic) and assumed type specifications, Liquid Haskell will check (or rather assume) that the specifications of chunkable monoid, as defined in the Definitions 1 and 3, are satisfied. We conclude with the assumption (rather that theorem) that `RString` is a chunkable monoid.

Assumption 5 (RString is a Chunkable Monoid) *(`RString`, η, ⊡) combined with the methods `lenStr`, `takeStr`, `dropStr` and `takeDropPropStr` is a chunkable monoid.*

4.2 String Matching Monoid

String matching is 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 target`, for the codomain of a string matching function, where `target` is the string being looked for. Additionally, we will define a function `toSM :: RString → SM target` which does the string matching and is indeed a monoid morphism from `RString` to `SM target` for a given `target`.

String Matching Monoid We define the data type `SM target` to contain a refined string field `input` and a list of all the indices in `input` where the `target` appears.

```
data SM (target :: Symbol) where
  SM :: input:RString
```

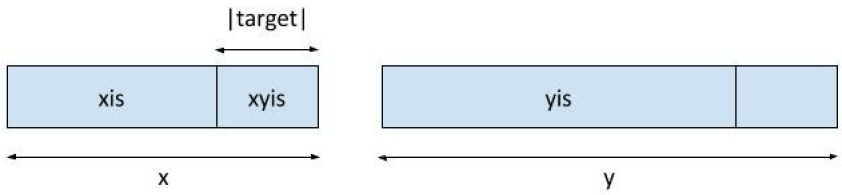

Fig. 2. Mappend indices of String Matcher

```
    → indices:[GoodIndex input target]
    → SM target
```

We use the string type literal [2] to parameterize the monoid over the target being matched. This encoding allows the type checker to statically ensure that only searches for the same target can be merged together. The input field is a refined string, and the indices field is a list of good indices. For simplicity we present lists as Haskell's built-in lists, but our implementation uses the reflected list type, `L`, defined in § 2.

A `GoodIndex input target` is a refined type alias for a natural number `i` for which `target` appears at position `i` of `input`. As an example, the good indices of `"abcab"` on `"ababcabcab"` are `[2,5]`.

```
type GoodIndex Input Target
  = {i:Nat | isGoodIndex Input (fromString Target) i }

isGoodIndex :: RString → RString → Int → Bool
isGoodIndex input target i
  = (subString i (lenStr target) input  == target)
  ∧ (i + lenStr target ≤ lenStr input)

subString :: Int → Int → RString → RString
subString o l = takeStr l . dropStr o
```

Monoid Methods for String Matching Next, we define the mappend and identity elements for string matching.

The *identity element* ϵ of `SM t`, for each target `t`, is defined to contain the identity `RString` (η) and the identity `List` (`[]`).

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

The Haskell definition of $\diamondsuit$, the monoid operation for `SM t`, is as follows.

```
(◇)::∀ (t::Symbol). KnownSymbol t ⇒ SM t → SM t → SM t
(SM x xis) ◇ (SM y yis)
```

[2] `Symbol` is a kind and target is effectively a singleton type.

```
  = SM (x ⊡ y) (xis′ ++ xyis ++ yis′)
  where
    tg   = fromString (symbolVal (Proxy :: Proxy t))
    xis′ = map (castGoodIndexLeft tg x y) xis
    xyis = makeNewIndices x y tg
    yis′ = map (shiftStringRight tg x y) yis
```

Note again that capturing target as a type parameter is critical, otherwise there is no way for the Haskell's type system to specify that both arguments of `(♢)` are string matchers on the same target.

The action of `(♢)` on the two `input` fields is straightforward; however, the action on the two `indices` is complicated by the need to shift indices and the possibility of new matches arising from the concatenation of the two `input` fields. Figure 2 illustrates the three pieces of the new `indices` field which we now explain in more detail.

1. Casting Good Indices If `xis` is a list of good indices for the string `x` and the target `tg`, then `xis` is also a list of good indices for the string `x ⊡ y` and the target `tg`, for each `y`. To prove this property we need to invoke the property `subStrAppendRight` on Refined Strings that establishes substring preservation on string right appending.

```
assume subStrAppendRight
    :: sl:RString → sr:RString → j:Int
    → i:{Int | i + j ≤ lenStr sl }
    → { subString sl i j = subString (sl ⊡ sr) i j }
```

The specification of `subStrAppendRight` ensures that for each string `sl` and `sr` and each integer `i` and `j` whose sum is within `sl`, the substring from `i` with length `j` is identical in `sl` and in `(sl ⊡ sr)`. The function `castGoodIndexLeft` applies the above property to an index `i` to cast it from a good index on `sl` to a good index on `(sl ⊡ sr)`

```
castGoodIndexLeft
  :: tg:RString → sl:RString → sr:RString
  → i:GoodIndex sl tg
  → {v:GoodIndex (sl ⊡ sr) target | v = i}

castGoodIndexLeft tg sl sr i
  = cast (subStrAppendRight sl sr (lenStr tg) i) i
```

Where `cast p x` returns `x`, after enforcing the properties of `p` in the logic

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

Moreover, in the logic, each expression `cast p x` is reflected as `x`, thus allowing random (*i.e.,* non-reflected) Haskell expressions to appear in `p`.

2. Creation of new indices The concatenation of two input strings `sl` and `sr` may create new good indices. For instance, concatenation of `"ababcab"` with `"cab"` leads to a new occurence of `"abcab"` at index `5` which does not occur in either of the two input strings. These new good indices can appear only at the last `lenStr tg` positions of the left input `sl`. `makeNewIndices sl sr tg` detects all such good new indices.

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

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

```
makeIndices
  :: s:RString → tg:RString → lo:Nat
  → hi:Int → [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
```

It is important to note that `makeNewIndices` does not scan all the input, instead only searching at most `lenStr tg` positions for new good indices. 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.

3. Shift Good Indices If `yis` is a list of good indices on the string `y` with target `tg`, then we need to shift each element of `yis` right `lenStr x` units to get a list of good indices for the string `x ⊡ y`.

To prove this property we need to invoke the property `subStrAppendLeft` on Refined Strings that establishes substring shifting on string left appending.

```
assume subStrAppendLeft
  :: sl:RString → sr:RString
  → j:Int → i:Int
  → {subStr sr i j = subStr (sl ⊡ sr) (lenStr sl+i) j}
```

The specification of `subStrAppendLeft` ensures that for each string `sl` and `sr` and each integers `i` and `j`, the substring from `i` with length `j` on `sr` is equal to the substring from `lenStr sl + i` with length `j` on `(sl ⊡ sr)`. The function `shiftStringRight` both shifts the input index `i` by `lenStr sl` and applies the `subStrAppendLeft` property to it, casting `i` from a good index on `sr` to a good index on `(sl ⊡ sr)`

Thus, `shiftStringRight` both appropriately shifts the index and casts the shifted index using the above theorem:

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

String Matching is a Monoid Next we prove that the monoid methods ϵ and $(\diamondsuit)$ satisfy the monoid laws.

Theorem 6 (SM is a Monoid). *(`SM t`, ϵ, $\diamondsuit$) is a monoid.*

Proof. According to the Monoid Definition 1, we prove that string matching is a monoid, by providing safe implementations for the monoid law functions. First, we prove *left identity*.

```
idLeft :: x:SM t → {ϵ ♢ x = xs }
idLeft (SM i is)
  =   (ϵ :: SM t) ♢ (SM i is)
  ==. (SM η []) ♢ (SM i is)
  ==. SM (η ⊡ i) (is1 ++ isNew ++ is2)
       ∴ idLeftStr i
  ==. SM i ([] ++ [] ++ is)
       ∴ (mapShiftZero tg i is ∧ newIsNullRight i tg)
  ==. SM i is
       ∴ idLeftList is
  *** QED
  where
    tg    = fromString (symbolVal (Proxy :: Proxy t))
    is1   = map (castGoodIndexRight tg i η) []
    isNew = makeNewIndices η i tg
    is2   = (map (shiftStringRight tg η i) is)
```

The proof proceeds by rewriting, using left identity of the monoid strings and lists, and two more lemmata.

– Identity of shifting by an empty string.

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

The lemma is proven by induction on `is` and the assumption that empty strings have length 0.

– No new indices are created.

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

The proof relies on the fact that `makeIndices` is called on the empty range from `0` to `-1` and returns `[]`.

Next, we prove *right identity*.

```
idRight :: x:SM t → {x ◇ ϵ = x }
idRight (SM i is)
  =   (SM i is) ◇ (ϵ :: SM t)
  ==. (SM i is) ◇ (SM η [])
  ==. SM (i ⊡ η) (is1 ++ isNew ++ is2)
      ∴ idRightStr i
  ==. SM i (is ++ N ++ N)
      ∴ (mapCastId tg i η is ∧ newIsNullLeft i tg)
  ==. SM i is
      ∴ idRightList is
  ***  QED
  where
    tg    = fromString (symbolVal (Proxy :: Proxy t))
    is1   = map (castGoodIndexRight tg i η) is
    isNew = makeNewIndices i stringEmp tg
    is2   = map (shiftStringRight tg i η) []
```

The proof proceeds by rewriting, using right identity on strings and lists and two more lemmata.

– Identity of casting is proven

```
mapCastId :: tg:RString → x:RString → y:RString
  → is:[GoodIndex x tg] →
  → {map (castGoodIndexRight tg x y) is = is}
```

We prove identity of casts by induction on `is` and identity of casting on a single index.

– No new indices are created.

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

The proof proceeds by case splitting on the relative length of `s` and `t`. At each case we prove by induction that all the potential new indices would be out of bounds and thus no new good indices would be created.

- Finally we prove *associativity*. For space, we only provide a proof sketch. The whole proof is available online [33]. Our goal is to show equality of the left and right associative string matchers.

```
assoc :: x:SM t → y:SM t → z:SM t
      → { x ♢ (y ♢ z) = (x ♢ y) ♢ z}
```

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. Equality of the index list proceeds in three steps.

1. Using list associativity and distribution of index shifting, we group the indices in the five lists shown in Figure 3: the indices of the input `x`, the new indices from mappending `x` to `y`, the indices of the input `y`, the new indices from mappending `x` to `y`, and the indices of the input `z`.
2. The representation of each group depends on the order of appending. For example, if `zis1` (resp. `zis2`) is the group `zis` when right (resp. left) mappend happened first, then we have

   ```
   zis1 = map (shiftStringRight tg xi (yi ⊡ zi))
              (map (shiftStringRight tg yi zi) zis)

   zis2 = map (shiftStringRight tg (xi ⊡ yi) zi) zis
   ```

   That is, in right first, the indices of `z` are first shifted by the length of `yi` and then by the length of `xi`, while in the left first case, the indices of `z` are shifted by the length of `xi ⊡ yi`. In this second step of the proof we prove, using lemmata, the equivalence of the different group representations. The most interesting lemma we use is called `assocNewIndices` and proves equivalence of all the three middle groups together by case analysis on the relative lengths of the target `tg` and the middle string `yi`.
3. After proving equivalence of representations, we again use list associativity and distribution of casts to wrap the index groups back in string matchers.

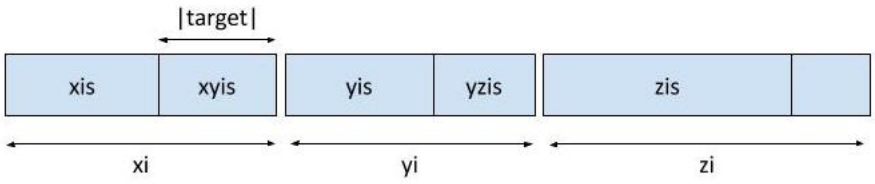

Fig. 3. Associativity of String Matching

We now sketch the three proof steps, while the whole proof is available online [33].

```
assoc x@(SM xi xis) y@(SM yi yis) z@(SM zi zis)
  -- Step 1: unwrapping the indices
  =   x ♢ (y ♢ z)
  ==. (SM xi xis) ♢ ((SM yi yis) ♢ (SM zi zis))
                          ...
  -- via list associativity and distribution of shifts
  ==. SM i (xis1 ++ ((xyis1 ++ yis1 ++ yzis1) ++ zis1))
  -- Step 2: Equivalence of representations
  ==. SM i (xis2 ++ ((xyis1 ++ yis1 ++ yzis1) ++ zis1))
      ∴ castConcat tg xi yi zi xis
  ==. SM i (xis2 ++ ((xyis1 ++ yis1 ++ yzis1) ++ zis2))
      ∴ mapLenFusion tg xi yi zi zis
  ==. SM i (xis2 ++ ((xyis2 ++ yis2 ++ yzis2) ++ zis2))
      ∴ assocNewIndices y tg xi yi zi yis
  -- Step 3: Wrapping the indices
                          ...
  -- via list associativity and distribution of casts
  ==. (SM xi xis ♢ SM yi yis) ♢ SM zi zis
  =   (x ♢ y) ♢ z
  *** QED
  where
    i     = xi ⊡ (yi ⊡ zi)

    yzis1 = map (shiftStringRight tg xi (yi ⊡ zi)) yzis
    yzis2 = makeNewIndices (xi ⊡ yi) zi tg
    yzis  = makeNewIndices yi zi tg
    ...
```

□

4.3 String Matching Monoid Morphism

Next, we define the function `toSM :: RString → SM target` which does the actual string matching computation for a set target [3]

```
toSM :: ∀ (target :: Symbol). (KnownSymbol target)
     ⇒ RString → SM target
toSM input = SM input (makeSMIndices input tg) where
  tg = fromString (symbolVal (Proxy :: Proxy target))

makeSMIndices
  :: x:RString → tg:RString → [GoodIndex x tg]
makeSMIndices x tg
  = makeIndices x tg 0 (lenStr tg - 1)
```

[3] `toSM` assumes the target is clear from the calling context; it is also possible to write a wrapper function taking an explicit target which gets existentially reflected into the type.

The input field of the result is the input string; the indices field is computed by calling `makeIndices` within the range of the `input`, that is from `0` to `lenStr input - 1`.

We now prove that `toSM` is a monoid morphism.

Theorem 7 (`toSM` is a Morphism). *`toSM :: RString` $\rightarrow$ `SM t` is a morphism between the monoids (`RString`, η, $\boxdot$) and (`SM t`, ϵ, $\diamondsuit$).*

Proof. Based on definition 2, proving `toSM` is a morphism requires constructing a valid inhabitant of the type

```
Morphism RString (SM t) toSM
  = x:RString → y:RString
  → {toSM η = ϵ ∧ toSM (x ⊡ y) = toSM x ♢ toSM y}
```

We define the function `distributestoSM :: Morphism RString (SM t) toSM` to be the required valid inhabitant.

The core of the proof starts from exploring the string matcher `toSM x ♢ toSM y`. This string matcher contains three sets of indices as illustrated in Figure 2: (1) `xis` from the input `x`, (2) `xyis` from appending the two strings, and (3) `yis` from the input `y`. We prove that appending these three groups of indices together gives exactly the good indices of `x ⊡ y`, which are also the value of the indices field in the result of `toSM (x ⊡ y)`.

```
distributestoSM x y
  =   (toSM x :: SM target) ♢ (toSM y :: SM target)
  ==. (SM x is1) ♢ (SM y is2)
  ==. SM i (xis ++ xyis ++ yis)
  ==. SM i (makeIndices i tg 0 hi1 ++ yis)
      ∴ (mapCastId tg x y is1 ∧ mergeNewIndices tg x y)
  ==. SM i (makeIndices i tg 0       hi1
          ++ makeIndices i tg (hi1+1) hi)
      ∴ shiftIndicesRight 0 hi2 x y tg
  ==. SM i is
      ∴ mergeIndices i tg 0 hi1 hi
  ==. toSM (x ⊡ y)
  *** QED
  where
    xis  = map (castGoodIndexRight tg x y) is1
    xyis = makeNewIndices x y tg
    yis  = map (shiftStringRight   tg x y) is2
    tg   = fromString (symbolVal (Proxy::Proxy target))
    is1  = makeSMIndices x tg
    is2  = makeSMIndices y tg
    is   = makeSMIndices i tg
    i    = x ⊡ y
    hi1  = lenStr x - 1
```

```
    hi2  = lenStr y - 1
    hi   = lenStr i - 1
```

The most interesting lemma we use is `mergeIndices x tg lo mid hi` that states that for the input `x` and the target `tg` if we append the indices in the range from `to` to `mid` with the indices in the range from `mid+1` to `hi`, we get exactly the indices in the range from `lo` to `hi`. This property is formalized in the type of the lemma.

```
mergeIndices
 :: x:RString → tg:RString
 → lo:Nat → mid:{Int | lo ≤ mid} → hi:{Int | mid ≤ hi}
 → {   makeIndices x tg lo hi
     =  makeIndices x tg lo mid
     ++ makeIndices x tg (mid+1) hi}
```

The proof proceeds by induction on `mid` and using three more lemmata:

- `mergeNewIndices` states that appending the indices `xis` and `xyis` is equivalent to the good indices of `x ⊡ y` from `0` to `lenStr x - 1`. The proof case splits on the relative sizes of `tg` and `x` and is using `mergeIndices` on `mid = lenStr x1 - lenStr tg` in the case where `tg` is smaller than `x`.
- `mapCastId` states that casting a list of indices returns the same list.
- `shiftIndicesRight` states that shifting right `i` units the indices from `lo` to `hi` is equivalent to computing the indices from `i + lo` to `i + hi` on the string `x ⊡ y`, with `lenStr x = i`.

□

4.4 Parallel String Matching

We conclude this section with the definition of a parallelized version of string matching. We put all the theorems together to prove that the sequential and parallel versions always give the same result.

We define `toSMPar` as a parallel version of `toSM` using machinery of section 3.

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

First, `chunkStr` splits the input into `j` chunks. Then, `pmap` applies `toSM` at each chunk in parallel. Finally, `pmconat` concatenates the mapped chunks in parallel using ♢, the monoidal operation for `SM target`.

Correctness. We prove correctness of `toSMPar` directly from Theorem 4.

Theorem 8 (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:Int → j:Int → x:RString
            → {toSM x = toSMPar i j x}
```

Proof. The proof follows by direct application of Theorem 4 on the chunkable monoid (`RString`, η, $\boxdot$) (by Assumption 5) and the monoid (`SM t`, ϵ, $\diamondsuit$) (by Theorem 6).

```
correctness i j x
  =   toSMPar i j x
  ==. pmconcat i (pmap toSM (chunkStr j x))
  ==. toSM is
    ∴ parallelismEquivalence toSM distributestoSM x i j
  *** QED
```

Note that application of the theorem `parallelismEquivalence` requires a proof that its first argument `toSM` is a morphism. By Theorem 2, the required proof is provided as the function `distributestoSM`. □

5 Evaluation: Strengths & Limitations

Verification of Parallel String Matching is the first realistic proof that uses (Liquid) Haskell to prove properties *about* program functions. In this section we use the String Matching proof to quantitatively and qualitatively evaluate theorem proving in Haskell.

Quantitative Evaluation. The Correctness of Parallel String Matching proof can be found online [33]. Verification time, that is the time Liquid Haskell needs to check the proof, is 75 sec on a dual-core Intel Core i5-4278U processor. The proof consists of *1839* lines of code. Out of those

- *226* are Haskell "runtime" code,
- *112* are liquid comments on the "runtime" Haskell code,
- *1307* are Haskell proof terms, that is functions with `Proof` result type, and
- *194* are liquid comments to specify theorems.

Counting both liquid comments and Haskell proof terms as verification code, we conclude that the proof requires 7x the lines of "runtime" code. This ratio is high and takes us to 2006 Coq, when Leroy [17] verified the initial CompCert C compiler with the ratio of verification to compiler lines being 6x.

Strengths. Though currently verbose, deep verification using Liquid Haskell has many benefits. First and foremost, *the target code is written in the general purpose Haskell* and thus can use advanced Haskell features, including type literals, deriving instances, inline annotations and optimized library functions like `ByteString`. Even diverging functions can coexist with the target code, as long as they are not reflected into logic [31].

Moreover, *SMTs are used to automate the proofs* over key theories like linear arithmetic and equality. As an example, associativity of `(+)` is assumed throughout the proofs while shifting indices. Our proof could be further automated by mapping refined strings to SMT strings and using the automated SMT string

theory. We did not follow this approach because we want to show that our techinique can be used to prove any (and not only domain specific) program properties.

Finally, we get further automation via *Liquid Type Inference* [22]. Properties about program functions, expressed as type specifications with unit result, often depend on program invariants, expressed as vanilla refinement types, and vice versa. For example, we need the invariant that all indices of a string matcher are good indices to prove associativity of `(♢)`. Even though Liquid Haskell cannot currently synthesize proof terms, it performs really well at inferring and propagating program invariants (like good indices) via the abstract interpretation framework of Liquid Types.

Limitations. There are severe limitations that should be addressed to make theorem proving in Haskell a pleasant and usable technique. As mentioned earlier *the proofs are verbose*. There are a few cases where the proofs require domain specific knowledge. For example, to prove associativity of string matching `x ♢ (y ♢ z) = (x ♢ y) ♢ z` we need a theorem that performs case analysis on the relative length of the input field of `y` and the target string. Unlike this case split though, most proofs do not require domain specific knowledge and merely proceed by term rewriting and structural inductuction that should be automated via Coq-like [3] tactics or/and Dafny-like [16] heuristics. For example, synquid [21] could be used to automatically synthesize proof terms.

Currently, we suffer from two engineering limitations. First, all reflected function should exist in the same module, as reflection needs access to the function implementation that is unknown for imported functions. This is the reason why we need to use a user defined, instead of Haskell's built-in, list. In our implementation we used `CPP` as a current workaround of the one module restriction. Second, class methods cannot be currently reflected. Our current workaround is to define Haskell functions instead of class instances. For example (`append`, `nil`) and (`concatStr`, `emptyStr`) define the monoid methods of List and Refined String respectively.

Overall, we believe that the strengths outweigh the limitations which will be addressed in the near future, rendering Haskell a powerful theorem prover.

6 Related Work

SMT-Based Verification SMT solvers have been extensively used to automate reasoning on verification languages like Dafny [16], Fstar [28] and Why3 [10]. These languages are designed for verification, thus have limited support for commonly used language features like parallelism and optimized libraries that we use in our verified implementation. Refinement Types [8,12,24] on the other hand, target existing general purpose languages, such as ML [34,2,22], C [7,23], Haskell [31], Racket [15] and Scala [26]. However, before Refinement Reflection [32] was introduced, they only allowed "shallow" program specifications, that is, properties that only talk about behaviors of program functions but not functions themselves.

Dependent Types Unlike Refinement Types, dependent type systems, like Coq [3], Adga [19] and Isabelle/HOL [20] allow for "deep" specifications which talk about program functions, such as the program equivalence reasoning we presented. Compared to (Liquid) Haskell, these systems allow for tactics and heuristics that automate proof term generation but lack SMT automations and general-purpose language features, like non-termination, exceptions and IO. Zombie [5,27] and Fstar [28] allow dependent types to coexist with divergent and effectful programs, but still lack the optimized libraries, like `ByteSting`, which come with a general purpose language with long history, like Haskell.

Parallel Code Verification Dependent type theorem provers have been used before to verify parallel code. BSP-Why [11] is an extension to Why2 that is using both Coq and SMTs to discharge user specified verification conditions. Daum [9] used Isabelle to formalize the semantics of a type-safe subset of C, by extending Schirmer's [25] formalization of sequential imperative languages. Finally, Swierstra [29] formalized mutable arrays in Agda and used them to reason about distributed maps and sums.

One work closely related to ours is SyDPaCC [18], a Coq library that automatically parallelizes list homomorphisms by extracting parallel Ocaml versions of user provided Coq functions. Unlike SyDPaCC, we are not automatically generating the parallel function version, because of engineering limitations (§ 5). Once these are addressed, code extraction can be naturally implemented by turning Theorem 4 into a Haskell type class with a default parallelization method. SyDPaCC used maximum prefix sum as a case study, whose morphism verification is much simpler than our string matching case study. Finally, our implementation consists of 2K lines of Liquid Haskell, which we consider verbose and aim to use tactics to simplify. On the contrary, the SyDPaCC implementation requires three different languages: 2K lines of Coq with tactics, 600 lines of Ocaml and 120 lines of C, and is considered "very concise".

7 Conclusion

We made the first non-trivial use of (Liquid) Haskell as a proof assistant. We proved the parallelization of chunkable monoid morphisms to be correct and applied our parallelization technique to string matching, resulting in a formally verified parallel string matcher. Our proof uses refinement types to specify equivalence theorems, Haskell terms to express proofs, and Liquid Haskell to check that the terms prove the theorems. Based on our 1839LoC sophisticated proof we conclude that Haskell can be successfully used as a theorem prover to prove arbitrary theorems about real Haskell code using SMT solvers to automate proofs over key theories like linear arithmetic and equality. However, Coq-like tactics or Dafny-like heurestics are required to ease the user from manual proof term generation.

References

1. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. 2010.
2. J. Bengtson, K. Bhargavan, C. Fournet, A.D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*, 2008.
3. Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.
4. Guy E. Blelloch. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Pub, 1993.
5. C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. In *POPL*, 2014.
6. Murray Cole. Parallel programming, list homomorphisms and the maximum segment sum problem. 1993.
7. Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *ESOP*, 2007.
8. R. L. Constable and S. F. Smith. Partial objects in constructive type theory. In *LICS*, 1987.
9. M Daum. Reasoning on Data-Parallel Programs in Isabelle/Hol. In *C/C++ Verification Workshop*, 2007.
10. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – Where Programs Meet Provers. In *ESOP*, 2013.
11. J Fortin and F. Gava. BSP-Why: A tool for deductive verification of BSP algorithms with subgroup synchronisation. In *Int J Parallel Prog*, 2015.
12. T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, 1991.
13. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 1993.
14. Joseph JáJá. *Introduction to Parallel Algorithms*. 1992.
15. Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. Occurrence typing modulo theories. In *PLDI*, 2016.
16. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. LPAR, 2010.
17. Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL 06*, 2006.
18. Frédéric Loulergue, Wadoud Bousdira, and Julien Tesson. Calculating Parallel Programs in Coq using List Homomorphisms. In *International Journal of Parallel Programming*, 2016.
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. Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, 2016.
22. P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.
23. P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, 2010.
24. J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in pvs. *IEEE TSE*, 1998.
25. N Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, TU Munich, 2006.
26. Georg Stefan Schmid and Viktor Kuncak. SMT-based Checking of Predicate-Qualified Types for Scala. In *Scala*, 2016.

27. Vilhelm Sjöberg and Stephanie Weirich. Programming up to congruence. *POPL*, 2015.
28. Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *POPL*, 2016.
29. Wouter Swierstra. More dependent types for distributed arrays. 2010.
30. N. Vazou, E. L. Seidel, and R. Jhala. Liquidhaskell: Experience with refinement types in the real world. In *Haskell Symposium*, 2014.
31. N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement Types for Haskell. In *ICFP*, 2014.
32. Niki Vazou and Ranjit Jhala. Refinement Reflection. arXiv:1610.04641, 2016.
33. Niki Vazou and Jeff Polakow. Code for verified string indexing. 2016. `https://github.com/nikivazou/verified_string_matching`.
34. H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, 1998.