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 refine-ment type checker for Haskell programs, can be used for arbitrary theorem proving by verifying a 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); we compare both proofs, uncovering the relative advantages and disadvantages of the two provers.

ACM Reference format:
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Liquid Haskell [29] is a verifier for Haskell programs that automatically checks whether the code satisfies logical specifications – expressed as refinement types – using an SMT [1] solver. For decidable and predictable SMT-based verification, Liquid Haskell used to limit all specifications to decidable theories (e.g., linear arithmetic and uninterpreted functions). Refinement reflection [27] removed this limitation allowing arbitrary, terminating Haskell functions to appear in the specifications. To preserve decidable SMT-verification under arbitrary specifications, the user has to manually prove the specifications that fall outside of the SMT-decidable theories. 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 [30].

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:
  1. dividing up the input in chunks,
  2. applying the morphism in parallel to all chunks, and
  3. recombining the mapped chunks using the monoid operation, also in parallel.

Our proof assumes the correctness of a single parallelization primitive in Haskell’s parallel library. We then apply this technique (§ 5) to a sequential string matcher to obtain a correct parallel 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 theorem (§ 4) and then we delve deeper into the comparison, highlighting differences based on the string matching case study (§ 6). Finally, we complete the evaluation picture by providing additional quantitative comparisons (§ 7).

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 ⊗ 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.

\[
\text{data } L \ [\text{length} ] \ a = N \ | \ C \ (\text{head} :: a, \ \text{tail} :: L \ a)
\]

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

\[
\text{length} \ :\ : L \ a \rightarrow (v:\text{Integer} \ | \ 0 \leq v)
\]

\[
\text{length} \ N \ = \ 0
\]

\[
\text{length} \ (C \times xs) \ = \ 1 + \text{length} \ xs
\]

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 \( 0 \leq v \). 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 [10, 29], generates two proof obligations. For the $N$ case
it checks that the body $v = 0$ is a natural number.

\[
\begin{align*}
\text{For the } C \text{ case Liquid Haskell binds the recursive call to a fresh}
\text{variable } v_r = 1 + \text{length } x & \text{ and checks that assuming the specification}
\text{for } v_r, \text{i.e., assuming that } v_r \text{ is a natural number, the body } v = 1 + v_r \\
& \text{is also non-negative.}
\end{align*}
\]

\[
\begin{align*}
0 \leq v_r & \Rightarrow v = 1 + v_r \Rightarrow 0 \leq v
\end{align*}
\]

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
$e$ (the empty list) and an associative operator $(\odot)$ (list append).

\[
\begin{align*}
e :: & : L a \quad (\odot) :: : L a \rightarrow L a \rightarrow L a \\
e = N & \quad N \odot ys = ys \\
(C \times xs) \odot ys = C \times (xs \odot ys)xs
\end{align*}
\]

Our goal is to specify the monoid laws on the above operators as
refinement types and prove them 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. For example, the validity check of both the
linear arithmetic statements (1) and (2) is efficiently decided by
the SMT. By default, Liquid Haskell 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 instance, back to the length example, the recursive
call \text{length } x is, by default, interpreted in the logic as a value
$v_r$ that only satisfies the \text{length} specification of being a natural
number.

Liquid Haskell lifts Haskell functions into the logic using the
\text{measure} and \text{reflect} annotations, preserving SMT decidability.

- The \text{measure} $f$ annotation [29] 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 \text{reflect} $f$ annotation [27] lifts the arbitrary, terminating
  Haskell function $f$ into the logic but, for decidable type check-
  ing, $f$ 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 \text{length} is defined on exactly one ADT (i.e., the List) it is
lifted in the refinement logic as a \text{measure}

\[
\begin{align*}
\text{measure length}
\end{align*}
\]

With the above measure annotation, Liquid Haskell interprets
\text{length} into the logic by automatically strengthening the types
of the \text{List} data constructors. For example, the type of \text{C} is auto-
matically strengthened to

\[
\begin{align*}
\text{C :: } : x : a & \rightarrow x : L a \\
& \rightarrow (v : L a \mid \text{length } v = \text{length } x + 1 )
\end{align*}
\]

\^1It 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.

We lift the monoid operators $e$ and $(\odot)$ in the logic via reflection.

\[
\begin{align*}
\text{reflect } e, (\odot)
\end{align*}
\]

The \text{reflect} annotations lift $(\odot)$ and $(e)$ into the logic by auto-
matically strengthening the types of the functions’ specifications.

\[
\begin{align*}
(e) :: \{ & v : L a \mid v = e \land v = N \\
(\odot) :: & x : L a \rightarrow y : L a \\
& \rightarrow (v : L a \mid v = xs \odot ys \\
& \land v = \text{isN } x \text{ then } ys \\
& \text{else } C (\text{head } x z) (\text{tail } x z \odot ys) )
\end{align*}
\]

Here, the $(\odot)$ and $(e)$ appearing in the refinements are uninter-
preted functions and \text{isN}, head, and \text{tail} are automatically gen-
erated measures. To preserve predictable type checking, Liquid
Haskell will not attempt to unfold the reflected functions into the
logic [15]. 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.

\subsection{2.2 Left Identity}

In Liquid Haskell, we express theorems as refined type speci-
fications and proofs as their Haskell inhabitants. We construct
proof inhabitants using the combinators from the built-in library
\text{ProofCombinators} \textsuperscript{2} that are summarized in Figure 1. A \text{Proof}

is a unit type that when refined is used to specify theorems. A \text{trivial}

proof is the unit value. For example, \text{trivial} :: \{v : \text{Proof} \mid 1 + 2 = 3\} trivially proves the theorem $1 + 2 = 3$ using the

SMT solver. The expression $p \text{ Proof}$ casts any expression $p$ into a

\text{Proof}. The equality assertion $x \equiv y$ states that $x$ and $y$ are
equal; it also 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, \text{x == y \text{Proof}} proves $x = y$ using the

proof term \text{Lemma}. To avoid parenthesizing the optional proof

argument in the common case where \text{Lemma} is an application and

not a variable, we follow the same approach as Haskell’s dollar

and define the \text{.} operator with appropriate precedence (thus, we

can write $x == y$. \text{.Lemma}) Finally, \text{x \equiv y \text{.Lemma}} combines two proofs

x and y into one by inserting the argument proofs into the logic

environment.

Armed with these combinators, left identity is expressed as a re-
finement type signature that takes as input a list $x : L a$ and returns a

\text{Proof} (i.e., unit) type refined with the property $e \odot x = x$.

\[
\begin{align*}
\text{idLeftList :: } x : L a & \rightarrow (e \odot x = x ) \\
\text{idLeftList } x & = e \odot x == . N \odot x == . x \text{ ** QED}
\end{align*}
\]

We write $(e \odot x = x)$ as a simplification for $(v : \text{Proof} \mid e \odot x = x)$ since the binder $v$ is irrelevant. We begin from the left hand
side $e \odot x$, which is equal to $N \odot x$ by calling $e$ thus unfold-
ing the equality empty $N$ into the logic. The proof combinator $x$

\text{.Lemma}$, \text{let}$ us equate $x$ with $y$ in the logic and returns $x$

allowing us to continue the equational proof. Next, the call $N \odot x$

refolds the definition of $(\odot)$ on $N$ and $x$, which is equal to $x$,

concluding our proof. Finally, we use the operator $p \text{ Proof}$ which casts $p$

\textsuperscript{2}The \text{ProofCombinators} library comes with Liquid Haskell and is defined in

https://github.com/ucd-progsys/liquidhaskell/blob/develop/include/Language/

Haskell/Liquid/ProofCombinators.hs.
into a proof term. In short, the proof of left identity, proceeds by unfolding the definitions of \( \epsilon \) and \( (\cdot) \) on the empty list.

### 2.3 PSE: Proof by Static Evaluation

To automate trivial proofs, Liquid Haskell uses PSE (Proof by Static Evaluation) a terminating but incomplete heuristic, inspired by [15], that automatically unfolds reflected functions in proof terms. PSE evaluates (i.e., unfolds) a reflected function call if it can be statically decided what branch the evaluation takes, e.g., \( N \circ ys \) is unfolded to \( ys \) while \( xs \circ ys \) is not unfolded when the structure of \( xs \) cannot be statically decided. Unlike SMT’s axiom instantiation heuristics (e.g., E-matching [7,19]) that make verification unstable [15], PSE is always terminating and is enabled on a per-function basis. For instance, the annotation

\[ \text{automatic-instances idLeftList} \]

activates PSE in the \text{idLeftList} function. Therefore, when PSE is used to complete a proof the verification of the rest of the program is not affected, even though it could be unpredictable whether the specific proof synthesis succeeds. Thus, global verification stability is preserved.

PSE is used to simplify the left identity proof by automatically unfolding \( \epsilon \) to \( N \) and then \( N \circ x \) to \( x \). (We use the cornered one line frame to denote Liquid Haskell proofs that use PSE via the \text{automatic-instances annotation}.)

\[ \text{idLeftList} :: x: L a \rightarrow \{ \epsilon \circ x = x \} \]
\[ \text{idLeftList} = \text{trivial} \]

That is the proof proceeds, trivially, by symbolic evaluation of the expression \( \epsilon \circ x \).

### 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.

\[ \text{idRightList} :: x: L a \rightarrow \{ x \circ \epsilon = x \} \]
\[ \text{idRightList} N = N \circ \epsilon ==. N \text{ *** QED} \]
\[ \text{idRightList} (C \times xs) = (C \times (xs \circ \epsilon)) ==. C \times (xs \circ \epsilon) ==. C \times xs \cdot \text{idRightList} xs \text{ *** QED} \]

The recursive call \text{idRightList} \( xs \) is provided as a third optional argument in the \( ==. \) operator to justify the equality \( xs \circ \epsilon = xs \), while the operator \( (\cdot) \) 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 is verifying 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 PSE heuristic to automatically generate all function unfoldings and simplify the right identity proof.

\[ \text{idRightList} :: x: L a \rightarrow \{ x \circ \epsilon = x \} \]
\[ \text{idRightList} N = \text{trivial} \]
\[ \text{idRightList} (C \times xs) = \text{idRightList} xs \]

PSE 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 \( C \times xs \circ \epsilon \) automatically unfolds to \( C \times (xs \circ \epsilon) \). Then the SMT will use the inductive hypothesis and congruence to conclude the proof.

### 2.5 Associativity

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

\[ \text{assocList} :: x: L a \rightarrow y: L a \rightarrow z: L a \rightarrow (x \circ (y \circ z)) = ((x \circ y) \circ z) \]
\[ \text{assocList} N = \text{trivial} \]
\[ \text{assocList} (C \times x) y z = \text{assocList} x y z \]

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 \( (m, \epsilon, \circ) \) is a monoid (with identity element \( \epsilon \) and associative operator \( \circ \)), if the following functions are defined.

\[ \text{idLeft}_m :: x: m \rightarrow (\epsilon \circ x = x) \]
\[ \text{idRight}_m :: x: m \rightarrow (x \circ \epsilon = x) \]
\[ \text{assoc}_m :: x: m \rightarrow y: m \rightarrow z: m \rightarrow (x \circ (y \circ z)) = ((x \circ y) \circ z) \]

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.** \( (L, a, \epsilon, \circ) \) 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 \( f \):

\[ \text{§ 3.1 chunk up the input in chunks of size } i \text{ (chunk } i), \]
\[ \text{§ 3.2 apply } f \text{ in parallel to all chunks (pmap } f), \text{ and} \]
\[ \text{§ 3.3 recombine the chunks, in parallel } j \text{ at a time, back to a single value (pmconcat } j). \]

In this section we implement and verify in Liquid Haskell 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 \((m, \epsilon, \odot)\) to be chunkable if for every natural number \(i\) and monoid \(x\), the functions \(\text{take}_m\ i\ x\) and \(\text{drop}_m\ i\ x\) are defined in such a way that \(\text{take}_m\ i\ x \odot \text{drop}_m\ i\ x\) exactly reconstructs \(x\).

\[
\begin{align*}
\text{length}_m & : m \to \text{Nat} \\
\text{drop}_m & : \text{Nat} \to x : \{m \mid i \leq \text{length}_m x\} \\
& \to \{v : m \mid \text{length}_m v = \text{length}_m x - i\} \\
\text{take}_m & : \text{Nat} \to x : \{m \mid i \leq \text{length}_m x\} \\
& \to \{v : m \mid \text{length}_m v = i\} \\
\text{take_drop_spec}_m & : \text{Nat} \to x : m \\
& \to \{x = \text{take}_m\ i\ x \odot \text{drop}_m\ i\ x\}
\end{align*}
\]

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 \(\text{take_drop_spec}\) specifies the reconstruction property.

Next, we use the \(\text{take}_m\) and \(\text{drop}_m\) methods for each chunkable monoid \((m, \epsilon, \odot)\) to define a \(\text{chunk}_m\ i\ x\) function that splits \(x\) in chunks of size \(i\).

\[
\begin{align*}
\textbf{type} \ Pos & = \{v : \text{Integer} \mid 0 < v\} \\
\text{chunk}_m & : \text{Pos} \to x : m \\
& \to \{v : m \mid \text{chunk_spec}_m\ i\ v\} \\
& \text{[}\text{length}_m x\text{]} \\
\text{chunk}_m\ i\ x \\
& \mid \text{length}_m x \leq i \\
& = C \times N \text{ } \\
& \mid \text{otherwise} \\
& = \text{take}_m\ i\ x \odot \text{chunk}_m\ i\ (\text{drop}_m\ i\ x)
\end{align*}
\]

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

\[
\begin{align*}
\text{chunk_spec}_m\ i\ x\ v \\
& \mid \text{length}_m x \leq i = \text{length}\ v = 1 \\
& \mid i = 1 = \text{length}\ v = \text{length}_m x \\
& \mid \text{otherwise} = \text{length}\ v < \text{length}_m x
\end{align*}
\]

The specifications of both \(\text{take}_m\) and \(\text{drop}_m\) are used to automatically verify the \(\text{length}_m\) constraints imposed by \(\text{chunk_spec}_m\).

Finally, we prove that the \(\text{Lists}\) defined in § 2 are chunkable monoids.

\[
\begin{align*}
\text{take}_m\ i\ N & = N \\
\text{take}_m\ i\ (C\times x) & \mid i = 0 = N \\
& \mid \text{otherwise} = C\times (\text{take}_m\ i-1\ x) \\
\text{drop}_m\ i\ N & = N \\
\text{drop}_m\ i\ (C\times x) & \mid i = 0 = C\times x \\
& \mid \text{otherwise} = \text{drop}_m\ i-1\ x
\end{align*}
\]

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 \(\text{length}\) defined in the previous section. Finally, the take-drop reconstruction specification is proved by induction on the size \(i\) and using the PSE tactic for the trivial static evaluation.

\[
\begin{align*}
\text{take_drop_spec}_m\ i\ N & = \text{trivial} \\
\text{take_drop_spec}_m\ i\ (C\times x) & \mid i = 0 = \text{trivial} \\
& \mid \text{otherwise} = \text{take_drop_spec}_m\ (i-1)\ x
\end{align*}
\]

3.2 Parallel Map

We define a parallelized map function \(\text{pmap}\) using Haskell’s parallel library. Concretely, we use the parallelization function \(\text{withStrategy}\), from Control.Parallel.Strategies, that computes its argument in parallel given a parallel strategy.

\[
\begin{align*}
\text{pmap} & : (a \to b) \to \text{List a} \to \text{List b} \\
\text{pmap}\ f\ xs & = \text{withStrategy}\ parStrategy\ \text{map}\ f\ xs
\end{align*}
\]

Parallelism in the Logic. The function \(\text{withStrategy}\), 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.

\[
\begin{align*}
\text{assume} \quad \text{withStrategy} & : \text{Strategy}\ a \\
& \to x : a \to \{v : a \mid v = x\}
\end{align*}
\]

Moreover, to reflect the implementation of \(\text{pmap}\) in the logic, the function \(\text{withStrategy}\) should also be represented in the logic. Liquid Haskell encodes \(\text{withStrategy}\) in the logic as a logical, \(i.e.,\) total, function that merely returns its second argument, \(\text{withStrategy}\ _x = x\). 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 \(\text{parStrategy}\) to be the traversable strategy.

\[
\begin{align*}
\text{parStrategy} & : \text{Strategy}\ (\text{L a}) \\
\text{parStrategy} & = \text{parTraversable}\ rseq
\end{align*}
\]
3.3 Parallel Monoidal Concatenation

The function \( \text{chunk}_m \) lets us turn a monoidal value into several pieces. Dually, for any monoid \((m, \cdot, \epsilon)\), the monoid concatenation \( \text{mconcat}_m \) turns a list \( m \) back into a single \( m \).

\[
\begin{align*}
\text{mconcat}_m &: \text{List} \times m \\ 
\text{mconcat}_m \ N &= \epsilon \\ 
\text{mconcat}_m \ (C \times xs) &= x \cdot \text{mconcat}_m \ xs
\end{align*}
\]

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

\[
\begin{align*}
\text{pmconcat}_m &: \text{Integer} \rightarrow \text{List} \times m \\ 
\text{pmconcat}_m \ i \times \ i \leq 1 \&\& \text{length} \ x \leq i \\ 
&= \text{mconcat}_m \ x \\ 
\text{pmconcat}_m \ i \times \\ 
&= \text{pmconcat}_m \ i \ (\text{pmap} \ \text{mconcat}_m \ (\text{chunk} \ i \ x))
\end{align*}
\]

Where \( \text{chunk} \) is the list chunkable operation \( \text{chunk}_\text{List} \). The function \( \text{pmconcat}_m \ i \times \ x \) calls \( \text{mconcat}_m \ x \) in the base case, otherwise it (1) chunks the list \( x \) in lists of size \( i \), (2) runs in parallel \( \text{mconcat}_m \), and (3) recursively runs itself with the resulting list.

Termination of \( \text{pmconcat}_m \) holds, as the length of \( \text{chunk} \ i \ x \) is smaller than the length of \( x \), when \( i < i \).

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

**Theorem 3.2.** For each monoid \( (m, \cdot, \epsilon) \) the parallel and sequential concatenations are equivalent:

\[
\text{pmconcatEq} :: i : \text{Integer} \rightarrow x : \text{List} \times m \\
\rightarrow \{ \text{pmconcat}_m \ i \times \ x = \text{mconcat}_m \ x \}
\]

**Proof.** The proof proceeds by structural induction on the input list \( x \). The details of the proof can be found in [30], here we sketch the proof.

First, we prove that \( \text{mconcat} \) distributes over list cutting.

\[
\begin{align*}
\text{mcut} :: i : \text{Nat} \rightarrow x : \text{ListEq} \ m i \\ 
\rightarrow \{ \text{mconcat}_m \ x = \text{mconcat}_m \ (\text{take} \ i \ x) \&\& \text{mconcat}_m \ (\text{drop} \ i \ x) \}
\end{align*}
\]

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

\[
\begin{align*}
\text{pmcutEq} :: f : (n \rightarrow m) \rightarrow \text{Morphism} \ n \times m \\
\rightarrow x : n \rightarrow \text{Pos} \rightarrow j : \text{Pos} \\
\rightarrow \{ f x = \text{pmconcat}_m \ i \ (\text{pmap} \ f \ (\text{chunk}_n \ j \ x)) \}
\end{align*}
\]

where the \( \text{Morphism} \ n \times m \ f \) argument is a proof argument that validates that \( f \) is indeed a morphism via the refinement type alias.

\[
\begin{align*}
type \text{Morphism} \ n \ m \ f = x : n \rightarrow y : n \\
\rightarrow (F \eta = \epsilon \& \& F \ (x \boxdot \ y) = F \ x \& \& F \ y)
\end{align*}
\]

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

**Lemma 3.4.** Let \( (m, \cdot, \epsilon) \) be a monoid and \( (n, \eta, \boxdot) \) be a chunkable monoid. Then, for every morphism \( f : n \rightarrow m \), every positive number \( i \) and input \( x \), \( f x = \text{mconcat}_m \ (\text{pmap} \ f \ (\text{chunk}_n \ i \ x)) \) holds.

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

\[
\begin{align*}
\text{parallelismEq} :: f : (n \rightarrow m) \rightarrow \text{Morphism} \ n \times m \\
\rightarrow x : n \rightarrow \text{Pos} \\
\rightarrow \{ f x = \text{mconcat}_m \ (\text{pmap} \ f \ (\text{chunk}_n \ i \ x)) \}
\end{align*}
\]

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 example 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 \take_spec{} and \drop_spec{} methods. (We use the double-lined code frame for Coq code.)

\[
\begin{align*}
\text{length}_m & : M \rightarrow \text{nat}; \\
\text{drop}_m & : \text{nat} \rightarrow M \rightarrow M; \\
\text{take}_m & : \text{nat} \rightarrow M \rightarrow M; \\
\text{drop}_{\text{spec}} & : \forall i, x, i \leq \text{length}_m x \rightarrow \\
& \text{length}_m (\text{drop}_m i x) = \text{length}_m x - i; \\
\text{take}_{\text{spec}} & : \forall i, x, i \leq \text{length}_m x \rightarrow \\
& \text{length}_m (\text{take}_m i x) = i; \\
\text{take}_{\text{drop}_{\text{spec}}} & : \forall i, x, \\
& x = \text{take}_m i x \circ \text{drop}_m i x;
\end{align*}
\]

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{} [13]—to define the chunkable monoid operations that had to be defined from scratch in Liquid Haskell (§ 3.1).

\[
\begin{align*}
\text{Definition length_list} & := \@\text{seq.size} A; \\
\text{Definition drop_list} & := \@\text{seq.drop} A; \\
\text{Definition take_list} & := \@\text{seq.take} A;
\end{align*}
\]

Coq’s libraries also come with already established theories. For example, to prove the \drop_{\text{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.

\[
\text{Theorem take_{\text{spec}} list :} \\
\forall i, x, i \leq \text{length}_m x \rightarrow \\
\text{length}_m (\text{drop}_m i x) = i.
\]

The crux of the proof lies in the application of the library lemma size\_take.

\[
\text{Lemma size\_take x : size (take i x) =} \\
\text{if i < size x then i else size x.}
\]

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 we resort to an adaptation of the Presburger Arithmetic solver \omega{} [22] 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-terminated Haskell functions (that do not flow into the logic) to be potentially diverging [29], 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 \(\text{length} x\) 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 \(\text{fuel}\) argument that trivially decreases at each recursive call.

\[
\text{Fixpoint chunk}_m (M : \text{Type}) (\text{fuel} : \text{nat}) (i : \text{nat}) (x : M) : \text{option (List} M\text{)}
\]

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:

\[
\text{Theorem chunk}_{\text{spec}}_m : \forall \{M\} i (x : M), \\
i > 0 \rightarrow \exists l, \\
\text{chunk}_m (\text{length}_m x) \cdot l \cdot i x = \text{Some} l \\
\lor \text{chunk}_\text{res}_m i x = \text{None}.
\]

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 [16]. Various such techniques
have been developed by the Coq community to tackle such recurs-
ions. In “Certified Programming with Dependent Types” [4],
Chlipala compares three general techniques to bypass Coq’s syn-
tactic 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 catego-
rized), 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 li-
braries from the Haskell ecosystem. For instance, in §3.2 we used
the library parallel for runtime parallelization and we axioma-
tized parallelism in logic. Coq does not have such a library, so we
axiomatize not only the behavior but also the existence of parallel
functions:

<table>
<thead>
<tr>
<th>Axiom Strategy</th>
<th>: Type.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axiom parStrategy</td>
<td>: Strategy.</td>
</tr>
<tr>
<td>Axiom withStrategy</td>
<td>: ∀ {A}, Strategy → A → A.</td>
</tr>
<tr>
<td>Axiom withStrategy_spec</td>
<td>: ∀ {A} (s : Strategy) (x : A), withStrategy s x = x.</td>
</tr>
</tbody>
</table>

In principle, one could extract these constants to their correspond-
ing 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 de-
fine a string matching function toSM :: RString → SM tg from
Refined Strings RString to a monoidal, string matching data struc-
ture 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 &: y = x 'BS.append' y
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 prop-
erties 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, ⧿, ⧿)
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 [28],
but it is both time consuming and orthogonal to our purpose. In-
stead, here we follow the easy route of axiomatization – demon-
strating 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 1 = takeStr 1 . 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 e of SM target, for each target, returns the
identity string (η) and the identity list ([η]).

3Symbol is a kind and target is effectively a singleton type (see GHC.TypeLits in
Hackage)
The mapping method \( \diamond \) of SM \( \tau \)g is explained in Figure 2, where the two string matchers \( \text{SM x xis} \) and \( \text{SM y xis} \) are appended. The returned input field is just \( x \) \( \boxtimes \) \( y \), while the returned indices field appends three list of indices: 1) the indices \( x \text{xis} \) on \( x \) casted to be good indices of the new input \( x \) \( \boxtimes \) \( y \), 2) the new indices \( y \text{xis} \) created when concatenating the two input strings, and 3) the indices \( y \text{xis} \) on \( y \), shifted right \( \text{lenStr} x \) units. The Haskell definition of \( \diamond \) captures the above three indexing operations.

Capturing the target \( \tau \)g as a type parameter is critical for the Haskell type system to specify that both arguments of \( \diamond \) 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 \( \tau \)g, then \( I \) is also a good index for the string \( x \) \( \boxtimes \) \( y \) and the same target, for any \( y \). This property cannot be automatically proven by Liquid Haskell, instead it is explicitly encoded in the function \( \text{castGoodIndex} \).

```haskell
castGoodIndex :: \( x : \text{RString} \rightarrow y : \text{RString} \rightarrow i : \text{GoodIndex} \times x \tau \)
\( \rightarrow (v : \text{GoodIndex} \times (x \boxtimes y) \tau \mid v = i) \)
\( \text{castGoodIndex} \times x \tau \)
\( = \text{substringAppendRight} \times x \tau \) \( \text{lenStr} \tau \) \( i \)["cast'"]
```

The definition of \( \text{castGoodIndex} \) 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 \( x \).

```haskell
\[ x : \text{RString} \rightarrow y : \text{RString} \rightarrow j : \text{Integer} \]
\[ \rightarrow (v : \text{integer} \mid i + j \leq \text{lenStr} x \}
\[ \text{substring} \times i \ j \] = \text{substring} \times (x \boxtimes y) \ i \ j \]
```

Refinement type casting is performed safely via the function \( \text{cast} \) \( p x \) that returns \( x \) and enforces the properties of \( p \) in the logic.

```haskell
\text{cast} :: b \times a \rightarrow (v : a \mid v = x)
\text{cast} \_ x = x
```

In the logic, \( \text{cast} \) \( p x \) 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 \( x \text{xis} \) in Figure 2. For instance, appending "ababca" with "ca" leads to a new occurrence of "abca" at index 5. These new good indices can appear only at the last \( \text{lenStr} \tau - 1 \) positions of the left input \( x \). The function \( \text{makeNewIndices} \) detects all such good new indices.

```haskell
\text{makeNewIndices} :: x : \text{RString} \rightarrow y : \text{RString} \rightarrow \text{lenStr} \tau \rightarrow [\text{GoodIndex} \times (x \boxtimes y) \tau]
\text{makeNewIndices} \times x y \tau \text{lenStr} \tau \leq 2 \[
\text{otherwise} = \text{makeIndices} \times (x \boxtimes y) \tau \text{lo hi}
\text{makeNewIndices} \tau \text{lo hi}
\text{makeNewIndices} \times s \tau \text{lo hi}
```

If the length of \( \tau \)g is less than 2, then no new good indices can be created. Otherwise, the call on \( \text{makeIndices} \) returns all the good indices of the input \( x \) \( \boxtimes \) \( y \) for target \( \tau \)g in the range from \( \text{maxInt} (\text{lenStr} x - (\text{lenStr} \tau - 1)) \) to \( \text{lenStr} x - 1 \).

Generally, \( \text{makeNewIndices} \) \( s \tau \text{lo hi} \) returns the good indices of the input string \( s \) for target \( \tau \)g in the range from \( \text{lo} \) to \( \text{hi} \) by recursively checking "goodness" of all the indices from \( \text{lo} \) to \( \text{hi} \).

```haskell
\text{makeNewIndices} :: s : \text{RString} \rightarrow \tau : \text{RString} \rightarrow \text{lo} : \text{Nat} \rightarrow \text{hi} : \text{Integer} \rightarrow [\text{GoodIndex} \times \tau]
\text{makeNewIndices} \times s \tau \text{lo hi}
\text{lo} \leq \text{hi}
\text{otherwise} = \text{rest}
```

Note that \( \text{makeNewIndices} \) does not scan all the input \( x \) and \( y \), instead only scans at most \( \text{lenStr} \tau \) 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 \( \tau \)g, then shifting \( i \) right \( \text{lenStr} x \) units gives a good index for the string \( x \) \( \boxtimes \) \( y \) on \( \tau \)g. This property is encoded in the function \( \text{shiftStringRight} \).

```haskell
\text{shiftStringRight} :: \( x : \text{RString} \rightarrow \text{range} \rightarrow i : \text{GoodIndex} \times \tau \)
```

The definition of \( \text{shiftStringRight} \) 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 \( x \) \( \boxtimes \) \( y \) from \( \text{lenStr} x + i \) of size \( j \).

```haskell
\text{shiftStringRight} \times x \tau \text{lo hi}
\text{shiftStringRight} \tau \text{lo hi}
\text{shiftStringRight} \times x \tau \text{lo hi}
```

The definition of \( \text{shiftStringRight} \) 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 \( x \) \( \boxtimes \) \( y \) from \( \text{lenStr} x + i \) of size \( j \).

```haskell
\text{assume} \text{substringAppendLeft} \times x \tau \text{lo hi} = \text{substring} \times (x \boxtimes y) \text{lo} \ j \]
```

The definition of \( \text{substringAppendLeft} \) 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 \( x \) \( \boxtimes \) \( y \) from \( \text{lenStr} x + i \) of size \( j \).
5.2.1 String Matching is a Monoid
Next we prove that the methods $\epsilon$ and $(\triangleright)$ satisfy the monoid laws.

Theorem 5.1 (SM is a Monoid). $(\text{SM} \ t, \epsilon, \triangleright)$ 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:
1. idLeft :: $x:\text{SM} \ t \rightarrow (\epsilon \triangleright x = x)$
2. idRight :: $x:\text{SM} \ t \rightarrow (x \triangleright \epsilon = x)$
3. assoc :: $x:\text{SM} \ t \rightarrow y:\text{SM} \ t \rightarrow z:\text{SM} \ t$
   $\rightarrow \{x \triangleright (y \triangleright z) = (x \triangleright y) \triangleright z\}$

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

\[
\begin{align*}
\text{idLeft (SM i is)} &= \text{idLeftStr i \wedge \text{idLeftList is}} \\
\wedge. \text{mapShiftZero tg i is} \wedge. \text{newIsNullLeft i tg}} \\
\text{where} tg &= \text{fromString (symbolVal (Proxy :: Proxy t))}
\end{align*}
\]

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.

\[
\begin{align*}
\text{mapShiftZero :: tg:RString \rightarrow i:RString} \\
\rightarrow \text{is:[GoodIndex i target]} \\
\rightarrow \text{[map (shiftStringRight tg i) is = is]}
\end{align*}
\]

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.

\[
\begin{align*}
\text{newIsNullLeft :: s:RString \rightarrow t:RString} \\
\rightarrow \text{(makeNewIndices s t = [])}
\end{align*}
\]

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 $((x \triangleright y) \triangleright z)$ and right $(x \triangleright (y \triangleright z))$ 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 mappndence of the mapp field, the indices can be split in five groups: the indices of the input $x$, the new indices from mappening $x$ and $y$, the indices of the input $y$, the new indices from mappening $y$ 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 $\text{toSM}$ which computes the string matcher for the input string on the type level target.

\[
\begin{align*}
\text{toSM :: V (tg :: Symbol). (KnownSymbol tg)} \\
\Rightarrow \text{RString} \rightarrow \text{SM} \ tg \\
\text{toSM input = SM input (go input tg')} \\
\text{where} \\
\text{tg' = fromString (symbolVal (Proxy :: Proxy t))}
\end{align*}
\]

We prove in [30] that $\text{toSM}$ is a monoid morphism.

Theorem 5.2. The function $\text{toSM}$ is a morphism between the monoids $(\text{RString}, \eta, \triangleright)$ and $(\text{SM} \ t, \epsilon, \triangleright)$, since the below function $\text{morphismsToSM}$ has a valid inhabitant.

\[
\begin{align*}
\text{morphismsToSM :: x:RString \rightarrow y:RString} \\
\rightarrow \{ \text{toSM} \eta = \epsilon \wedge \text{toSM} (x \triangleright y) = \text{toSM} x \triangleright \text{toSM} y\}
\end{align*}
\]

5.4 Parallel String Matching
Finally, we define $\text{toSMPar}$ as a parallel version of $\text{toSM}$, using machinery of section 3, and prove that the sequential and parallel versions always give the same result.

\[
\begin{align*}
\text{toSMPar :: V (tg :: Symbol). (KnownSymbol tg)} \\
\Rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{RString} \rightarrow \text{SM} \ tg \\
\text{toSMPar i j = pmconcat i . pmap toSM . chunkStr j}
\end{align*}
\]

First, $\text{chunkStr}$ splits the input into chunks of size $j$. Then, $\text{pmap}$ applies $\text{toSM}$ at each chunk in parallel. Finally, $\text{pmconcat}$ concatenates the mappnded chunks in parallel using $(\triangleright)$, the monoidal operation for $\text{SM} \ tg$. Correctness of $\text{toSMPar}$ directly follows from Theorem 3.3.

Theorem 5.3 (Correctness of Parallel String Matching). For each parameter $i$ and $j$, and input $x$, $\text{toSMPar i j x}$ is always equal to $\text{toSM} x$.

\[
\begin{align*}
\text{correctness :: i:Integer \rightarrow j:Integer} \\
\rightarrow x:RString \\
\rightarrow (\text{toSM} x = \text{toSMPar i j x})
\end{align*}
\]

Proof. The proof follows by direct application of Theorem 3.3 on the chunkable monoid $(\text{RString}, \eta, \triangleright)$ (by Assumption 1) and the monoid $(\text{SM} \ t, \epsilon, \triangleright)$ (by Theorem 5.1).
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 ByteStrings 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 Strings. 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.

\[\text{Definition isGoodIndex in tg i := substring i (length tg) in } = \text{ tg}.\]

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

\[\text{Definition isGoodIndexDec input tg i := \{isGoodIndex input tg i\} + \{~(isGoodIndex input tg i)\}.}\]

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 favors 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.

\[\text{Inductive SM (tg : string) := }\]
\[| \text{Sm : \forall (in : string) (is : list nat), SM tg}.\]

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

\[\text{Inductive validSM tg : SM tg \rightarrow Prop}\]

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

\[\text{Theorem sm_assoc tg (sm1 sm2 sm3 : SM tg) : validSM sg2 sm3} \rightarrow\]
\[\text{validSM sg1 sm2 sm3 = (sm1 sm2 sm3).}\]

Thus, the extrinsic \((\circ)\) 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 mapstring string matching operator we intrinsically restrict the type of sm to carry a proof of valid indices.

\[\text{Inductive sm tg : Type := }\]
\[| \text{mk_sm : \forall in is, Forall (isGoodIndex in tg) is } \rightarrow \text{ sm tg}.\]

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.

\[\text{Lemma proof_irrelevant_equality}\]
\[\text{tg xs xs' l H l' H' : xs = xs' } \rightarrow l = l'\]
\[\rightarrow \text{mk_sm tg xs l H = mk_sm tg xs' l' H'}.\]

We prove the above lemma using Proof Irrelevance, an admissible 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

\[\Box\]
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, 23, 25] described between the two provers.

7 Evaluation

7.1 Quantitative Comparison.

Table 1 summarizes the quantitative evaluation of our two proofs as implemented in [30]: 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 PSE (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 as much time while it needs x34 time using PSE. 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 PSE is an under-developed, non-optimized approach to heuristically synthesize proof terms by static evaluation. In small proofs, like the generalized parallelization theorem, PSE 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 PSE heuristic, as the proof terms are shrunk without any modification to the executable portion.

Evaluation of PSE. PSE is used to synthesize non-sophisticated proof terms, leading to fewer lines of proof code but slower verification time. We used PSE to synthesize 31 out of the 43 total number of proof terms. PSE failed to synthesize the rest proof terms due to: 1. incompleteness: PSE 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 PSE, 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).
### 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 PSE (1134 LoC in total) into specifications, proof terms and executable code.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallelization</td>
<td>5</td>
<td>121</td>
<td>329</td>
<td>39</td>
<td>8</td>
<td>54</td>
<td>164</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>String Matcher</td>
<td>33</td>
<td>127</td>
<td>437</td>
<td>83</td>
<td>87</td>
<td>199</td>
<td>831</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>38</td>
<td>248</td>
<td>766</td>
<td>122</td>
<td>95</td>
<td>253</td>
<td>995</td>
<td>180</td>
<td></td>
</tr>
</tbody>
</table>

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.

References