A Tale of Two Provers
Verifying Monoidal String Matching in Liquid Haskell and Coq

Niki Vazou
University of Maryland

Leonidas Lampropoulos
University of Pennsylvania

Jeff Polakow
Awake Networks

Abstract
We demonstrate for the first time that Liquid Haskell, a refinement
we compare both proofs, uncovering the relative advantages and
ACM Reference format:
Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. 2017. A Tale of
§
§
§
§

2 Liquid Haskell as a Theorem Prover
In this section we demonstrate how Haskell can be used as a theo-
Haskell '17, USA
DOI: 10.1145/3122955.3122963

1 Introduction
Liquid Haskell [25] is a verifier for Haskell programs that semi-
• specify monoid laws as refinement types,
• prove the laws using plain Haskell functions, and
• verify the proofs using Liquid Haskell.

1. divide the input in chunks,
2. apply the morphism in parallel to all chunks, and
3. recombine in parallel the mapped chunks.

Our proof assumes the correctness of Haskell’s parallel library. We then apply these three steps (§ 5) to a sequential string
matcher to obtain a correct, parallel (and thus faster) version.

• We evaluate the applicability of Liquid Haskell as a theorem
prover by repeating the same proof in the Coq proof assistant.
We identify interesting tradeoffs in the verification approaches
encouraged by the two tools in two parts: we first draw pre-
liminary conclusions based on the general parallelization steps
(§ 4) and then we delve deeper into the comparison, highlighting
differences based on the string matching case study (§ 6). Finally
(§ 7), we complete the evaluation picture by providing additional
quantitative comparisons of the two provers.

1. divide the input in chunks,
2. apply the morphism in parallel to all chunks, and
3. recombine in parallel the mapped chunks.

Our proof assumes the correctness of Haskell’s parallel li-

1. divide the input in chunks,
2. apply the morphism in parallel to all chunks, and
3. recombine in parallel the mapped chunks.

Our proof assumes the correctness of Haskell’s parallel library. We then apply these three steps (§ 5) to a sequential string
matcher to obtain a correct, parallel (and thus faster) version.

• We evaluate the applicability of Liquid Haskell as a theorem
prover by repeating the same proof in the Coq proof assistant.
We identify interesting tradeoffs in the verification approaches
encouraged by the two tools in two parts: we first draw pre-
liminary conclusions based on the general parallelization steps
(§ 4) and then we delve deeper into the comparison, highlighting
differences based on the string matching case study (§ 6). Finally
(§ 7), we complete the evaluation picture by providing additional
quantitative comparisons of the two provers.

2.1 Reflection of Lists into Logic
To begin with, we define a standard recursive List datatype.

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

The length annotation in the definition teaches Liquid Haskell to
use length to check the termination of functions recursive on lists.

The length function is defined as a standard Haskell function.

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

The refinement type specifies that length returns a natural number,
that is, length returns a Haskell Integer value v that is moreover
refined to satisfy the constraint θ ≤ v. To check the validity of this
specification, Liquid Haskell encodes Haskell’s Integer as a logical
quantity. To check the validity of this specification, Liquid Haskell encodes Haskell’s Integer as a logical
• 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 asso-
ciated 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.

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

The length annotation in the definition teaches Liquid Haskell to
use length to check the termination of functions recursive on lists.

The length function is defined as a standard Haskell function.

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

The refinement type specifies that length returns a natural number,
that is, length returns a Haskell Integer value v that is moreover
refined to satisfy the constraint θ ≤ v. To check the validity of this
specification, Liquid Haskell encodes Haskell’s Integer as a logical

integer\textsuperscript{1} and via standard refinement type constraint generation \cite{28,26}, generates two proof obligations. For the N case it checks that the body $v = 0$ is a natural number.

\[ v = 0 \Rightarrow 0 \leq v \]

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

\[ 0 \leq \nu_r \Rightarrow v = 1 + \nu_r \Rightarrow 0 \leq v \]

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

\[
\begin{align*}
\varepsilon & : : \text{L a} \\
\varepsilon & = \text{N} \\
\varepsilon & = \text{N} \odot \text{ys} = \text{ys} \\
(\text{C } \times \text{x s}) \odot \text{ys} & = \text{C } \times \text{(x s } \odot \text{ys)}
\end{align*}
\]

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

Liquid Haskell lifts Haskell functions into the logic using the \texttt{measure} and \texttt{reflect} annotations, that preserve SMT decidability.

- The \texttt{measure} $f$ annotation \cite{28} 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 \texttt{reflect} $f$ annotation \cite{26} lifts the arbitrary, terminating Haskell function $f$ into the logic but, for decidable type checking, $f$ is not automatically unfolded in the logic. Instead, as we shall describe, type level unfolding of the reflected function $f$ is manually performed via respective value level computations.

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

\[
\texttt{measure length}
\]

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

\[
\text{C} :: \text{x} : \text{a} \rightarrow \text{x s} : \text{L a} \\
\rightarrow \{ \text{v : L a } \mid \text{length } v = \text{length x s } + 1 \}
\]

where length is an uninterpreted function in the logic.

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

\[
\texttt{reflect } \varepsilon, (\odot)
\]

The \texttt{reflect} annotations lift $(\odot)$ and $(\varepsilon)$ into the logic by automatically strengthening the types of the functions’ specifications.

\[
\begin{align*}
\langle \odot \rangle & :: \text{x} : \text{L a} \rightarrow \text{ys} : \text{L a} \\
& \rightarrow \{ \text{v : L a } \mid \text{v } = \text{x } \odot \text{ys} \\
& \land \text{v } = \text{if } \text{isN x s then } \text{ys} \\
& \text{else } \text{C } (\text{head x s } \odot \text{tail x s } \odot \text{ys}) \}
\end{align*}
\]

Here, the $(\odot)$ and $(\varepsilon)$ appearing in the refinements are uninterpreted functions and $\text{isN}$, $\text{head}$, and $\text{tail}$ are automatically generated measures. To preserve predictable type checking, Liquid Haskell will not attempt to unfold the reflected functions into the logic \cite{14}. But after reflection, at each Haskell function call the function definition is unfolded exactly once into the logic, allowing Liquid Haskell to prove properties about Haskell functions.

\section{2.2 Le/f_t Identity}

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

Armed with these combinators, left identity is expressed as a refinement type signature that takes as input a list $\text{x} : \text{L a}$ and returns a \texttt{Proof} (i.e., unit) type refined with the property $\varepsilon \odot x = x$.

\[
\begin{align*}
\text{idLeft_List} & :: \text{x : L a } \rightarrow \{ \varepsilon \odot x = x \} \\
\text{idLeft_List} & \text{x } = \varepsilon \odot x = \text{N } \odot x = \text{N } ** \text{QED}
\end{align*}
\]

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

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

\textsuperscript{2}The \texttt{ProofCombinators} library comes with Liquid Haskell and is defined in https://github.com/ucsd-progsys/liquidhaskell/blob/develop/include/Language/Haskell/Liquid/ProofCombinators.hs.
2.3 PLE: Proof by Logical Evaluation

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

```
| automatic-instances idLeft List
```

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

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

```
idLeft List :: x : L a \rightarrow \{ \epsilon \circ x = x \}
idLeft List _ = trivial
```

That is the proof proceeds, trivially, by logical evaluation of $\epsilon \circ x$.

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.

```
idRight List :: x : L a \rightarrow \{ x \circ \epsilon = x \}
idRight List N = N \circ \epsilon =. N *** QED
idRight List (C \times xs) = (C \times xs) \circ \epsilon
==. C \times (xs \circ \epsilon)
==. C \times xs : idRight List xs
*** QED
```

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

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

```
idRight List :: x : L a \rightarrow \{ x \circ \epsilon = x \}
idRight List N = trivial
idRight List (C _ xs) = idRight_List xs
```

PLE performs symbolic unfolding but not case splitting, that is the cases should be explicitly split by the user. For instance, in the $C$ branch the term $C \times xs \circ \epsilon$ 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

Similarly, we prove associativity using structural induction.

```
assoc_List :: x : L a \rightarrow y : L a \rightarrow z : L a
\rightarrow (x \circ (y \circ z)) = (x \circ y) \circ z
assoc_List N _ _ = trivial
assoc_List (C _ x) y z = assoc_List x y z
```

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

2.6 Lists are a Monoid

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

**Definition 2.1 (Monoid).** The triple $(m, \epsilon, \circ)$ is a monoid (with identity element $\epsilon$ and associative operator $\circ$), if the following functions are defined.

```
idLeft_m :: x : m \rightarrow (\epsilon \circ x = x)
idAssoc_m :: x : m \rightarrow (x \circ \epsilon = x)
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.
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 \((m, e, \diamond)\) 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 \(\text{take}_m\ i\ x\ \diamond\ \text{drop}_m\ i\ x\) exactly reconstructs \(x\).

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

The functional methods of chunkable monoids are \(\text{take}\) and \(\text{drop}\), while the length method is required to give the pre- and post-condition on the other operations. The proof term \(\text{take}_m\ \text{drop}_m\ \text{spec}\) specifies the reconstruction property.

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

\[
\begin{align*}
type \ Pos &= \{v : \text{Integer} \mid 0 < v\} \\
\text{chunk}_m &: \ i : \text{Pos} \rightarrow x : m \\
& \rightarrow \{v : l m \mid \text{chunk_spec}_m i x v\} \\
& \quad / \{\text{length}_m x \} \\
\text{chunk}_m i x & \quad | \text{length}_m x \leq i = C x N \\
& \quad | \text{otherwise} = \text{take}_m i x \text{c1}_m \text{chunk}_m i \text{drop}_m i x
\end{align*}
\]

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 \(\text{chunk_spec}_m\).

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

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 \(\text{lists}\) from §2 are chunkable monoids.

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

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

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 length defined in the previous section. Finally, the take-drop reconstruction specification is proved by induction on the \(i\) and using the PLE tactic for the trivial logical evaluation.

3.2 Parallel Map

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

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

**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.

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

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 \((m, e, \diamond)\), the monoid concatenation \(\text{mconcat}_m\) turns a \(L m\) back into a single \(m\).

\[
\begin{align*}
\text{mconcat}_m &= \text{L m} \rightarrow m \\
\text{mconcat}_m N &= e \\
\text{mconcat}_m (C x x s) &= x \diamond \text{mconcat}_m xs
\end{align*}
\]
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.

\[
\text{pmconcat}_m :: \text{Integer} \rightarrow \text{L m} \rightarrow \text{m}
\]
\[
\text{pmconcat}_m \ i \ x \ | \ i \leq 1 || \text{length } x \leq i
\]
\[
= \text{mconcat}_m \ x
\]
\[
\text{pmconcat}_m \ i \ x
\]
\[
= \text{pmconcat}_m \ i \ (\text{pmap} \ \text{mconcat}_m \ (\text{chunk} \ i \ x))
\]

Where chunk is the list chunkable operation chunk\_List. The function \( \text{pmconcat}_m \ i \ x \) calls \( \text{mconcat}_m \ x \) in the base case, otherwise \( n \) chunks the list \( x \) in lists of size \( i \), (2) runs in parallel \( \text{mconcat}_m \) to each chunk, 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 \( i < 1 \).

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

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

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

**Proof.** The proof proceeds by structural induction on the input list \( x \) and the details can be found in [29].

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

\[
\text{mcut} :: \text{i:Nat} \rightarrow \text{x:LLEq m i}
\]
\[
\rightarrow \{ \text{mconcat}_m \ x = \text{mconcat}_m \ (\text{take} \ i \ x) \\
\odot \text{mconcat}_m \ (\text{drop} \ i \ x) \}
\]

**type** LLEq m I = \( \{ x:\text{L m} | I \leq \text{length } x \} \)

We generalize the above lemma to prove that \( \text{mconcat} \) distributes over list cutting.

\[
\text{mchunk} :: \text{i:Integer} \rightarrow \text{x:L m} \rightarrow \{ \text{mconcat}_m \ x = \text{mconcat}_m \ (\text{map} \ \text{mconcat}_m \ (\text{chunk} \ i \ x)) \}
\]

Both lemmata are proved by structural induction on the input list \( x \).

**Lemma mchunk** proves \( \text{pmconcatEq} \) by structural induction, using left identity in the base case. \( \square \)

### 3.4 Parallel Monoid Morphism

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

**Theorem 3.3** (Correctness of Parallelization). Let \((m, \varepsilon, \odot)\) be a monoid and \((n, \eta, \Box)\) be a chunkable monoid. Then, for every morphism \( f :: n \rightarrow m \), every positive number \( i \) and \( j \), and input \( x \), \( f \ x \ = \ \text{pmconcat}_i \ (\text{pmap} \ f \ (\text{chunk}_n \ j \ x)) \) holds.

\[
\text{parallelismEq} :: f:(n \rightarrow m) \rightarrow \text{Morphism} \ n \ m \ f
\]
\[
\rightarrow x:n \rightarrow i:Pos \rightarrow j:Pos \rightarrow \{ f \ x = \text{pmconcat}_i \ (\text{pmap} \ f \ (\text{chunk}_n \ j \ x)) \}
\]

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

**Proof.** We prove the equivalence in two steps. First we prove a lemma \( (\text{parallelismLemma}) \) 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 \((m, \varepsilon, \odot)\) be a monoid and \((n, \eta, \Box)\) 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.

\[
\text{parallelismLemma} :: f:(n \rightarrow m) \rightarrow \text{Morphism} \ n \ m \ f
\]
\[
\rightarrow x:n \rightarrow i:Pos
\]
\[
\rightarrow \{ f \ x = \text{mconcat}_m \ (\text{pmap} \ f \ (\text{chunk}_n \ i \ x)) \}
\]

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

In the base case use rewriting and right identity on the monoid \( f \ x \). In the inductive case, we use the inductive hypothesis with \( \text{drop}X = \text{drop}_n \ i \ x \), that is provably smaller than \( x \) as \( 1 < i \).

We get basic distribution for \( f \) \( \text{take}X \odot \text{drop}X = f \ (\text{take}X \Box \text{drop}X) \), since \( f \) is a monoid morphism as encoded in the argument \( \text{thm} \ \text{take}X \ \text{drop}X \). Finally, by the \text{takeDropProp}_n property of the chunkable monoid \( n \) we merge \( \text{take}X \Box \text{drop}X \) to \( x \).

Finally, the \( \text{parallelismEq} \) function is defined using the above lemma combined with the equivalence of parallel and sequential \( \text{mconcat} \) as encoded by \( \text{parallelismEq} \) in Theorem 3.2.

**4 Monoid Morphism Parallelization in Coq**

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

#### 4.1 Intrinsic vs. Extrinsic Verification

The translation of the chunkable monoid specification of § 3.1 in Coq is a characteristic instance of how Liquid Haskell and Coq naturally favor intrinsic and extrinsic verification respectively. The (intrinsic) Liquid Haskell pre- and post-conditions of the take and drop functions are not embedded in the Coq types, but are independently, i.e., extrinsically, encoded as specification terms in the extra \text{drop\_spec} and \text{take\_spec} methods. (We use the double-lined code to frame Coq code.)
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’s implicit and SMT-automatic proofs, 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 import library functions and their specifications (here ssreflect’s `seq`) to define the chunkable monoid operations that had to be defined from scratch in Liquid Haskell (§3.1).

\[
\begin{align*}
\text{Definition} \ \text{length} \_ \text{list} & := \@\text{seq.size} \ A; \\
\text{Definition} \ \text{drop} \_ \text{list} & := \@\text{seq.drop} \ A; \\
\text{Definition} \ \text{take} \_ \text{list} & := \@\text{seq.take} \ A;
\end{align*}
\]

Coq’s libraries also come with already established theories. For example, to prove the `drop_spec_list` we just apply an existing library lemma (`seq.size_drop`), unlike Liquid Haskell that currently 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_spec_list` for lists.

\[
\begin{align*}
\text{Theorem} \ \text{take} \_ \text{spec} \_ \text{list} : \\
\forall i, x, i \leq \text{length} \_ \text{list} x \rightarrow \\
\text{length} \_ \text{list} (\text{drop} \_ \text{list} i x) = i.
\end{align*}
\]

The crux of the proof is the library lemma `size_take`.

\[
\begin{align*}
\text{Lemma} \ \text{size} \_ \text{take} x & : \text{size} (\text{take} i x) = \\
\text{if} i < \text{size} x \text{ then } i \text{ else } \text{size} x.
\end{align*}
\]

However, the existing lemma and our desired specification differ when \(i\) is exactly equal to \(\text{size} x\), generating a linear arithmetic proof obligation. While in Liquid Haskell such obligations are automatically discharged by the SMT, in the Coq implementation [29] we need to explicitly invoke an adaptation of the advanced Presburger Arithmetic solver `omega` [21] for ssreflect.

This trivial example highlights a major 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, but, 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-reflective Haskell functions (that do not flow into the logic) to be potentially diverging [28], 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 `\(\text{length}\_x\)` 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 `fuel` argument that trivially decreases at each recursive call.

\[
\begin{align*}
\text{Fixpoint} \ \text{chunk}\_m \ (M : \text{Type}) \ (\text{fuel} : \text{nat}) \\
\text{ (i : \text{nat}) (x : M) : \text{option} (\text{list} M)}
\end{align*}
\]

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:

\[
\begin{align*}
\text{Theorem} \ \text{chunk} \_ \text{spec} \_ \text{m} : \forall (M) i (x : M), \\
i > 0 \rightarrow \exists l, \\
\text{chunk}\_m (\text{length}\_x) .+1 i x = \text{Some} l \\
\lor \text{chunk} \_ \text{res} \_ \text{m} i x 1.
\end{align*}
\]

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

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

4.5 Executable vs. Axiomatized Parallelism

Liquid Haskell verifies Haskell programs that use libraries from the Haskell ecosystem. For instance, in §3.2 we used the library

\[
\begin{align*}
\text{length}_m & : M \rightarrow \text{nat}; \\
\text{drop}_m & : \text{nat} \rightarrow M \rightarrow M; \\
\text{take}_m & : \text{nat} \rightarrow M \rightarrow M; \\
\text{drop} \_ \text{spec} \_ m & : \forall i, x, i \leq \text{length}_m x \rightarrow \\
\text{length}_m (\text{drop}_m i x) & = \text{length}_m x - i; \\
\text{take} \_ \text{spec} \_ m & : \forall i, x, i \leq \text{length}_m x \rightarrow \\
\text{length}_m (\text{take}_m i x) & = i; \\
\text{take} \_ \text{drop} \_ \text{spec} \_ m & : \forall i, x, \\
& i > \text{size} x \rightarrow \\
& \text{x} = \text{take}_m i x \diamond \text{drop}_m i x;
\end{align*}
\]
parallel for runtime parallelization and we axiomatized parallelism in logic. Coq does not have such a library, so we axiomatized not only the behavior but also the existence of parallel functions:

| Axiom Strategy       : Type. |
| Axiom parStrategy    : Strategy. |
| Axiom withStrategy   : V {A}, Strategy A A A. |
| Axiom withStrategy_spec : V {A} (s : Strategy) (x : A), withStrategy s x x. |

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

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

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

```
η = BS.empty
x ⊕ y = x 'BS.append' y

lenStr x = BS.length x

takeStr i x = BS.take i x

dropStr i x = BS.drop i x
```

We axiomatize the above refined string functions to satisfy the properties of chunkable monoids. For instance, we define a logical uninterpreted function ⊕ and relate it to the Haskell operator 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, for instance, associativity.

```
assume assocStr :: x : RString → y : RString → z : RString →
  (x □ (y □ z)) = (x □ (y □ z))
```

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

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

We note that actually proving that ByteString implements a chunkable monoid in Liquid Haskell is possible, as implied by [27], but it is both time consuming and orthogonal to our purpose. Instead, here we follow the easy route of axiomatization — demonstrating that Liquid Haskell verification can be gradual.

5.2 String Matching Monoid

String matching determines all the indices in a source string where a given target string begins. For example, for source string abab 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 (type level) string.

An index i is a good index on the string input for the target, if the target appears in the position i of the input. We encode good indexing using the refinement type alias GoodIndex I Tg (in Liquid Haskell’s type definitions arguments starting with upper and lower case letters stand for value and type parameters, respectively).

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

```
isGoodIndex input i = substring i (lenStr tg) input == tg
∧ i + lenStr tg ≤ lenStr input
```

We define the data type SM tg to contain a refined string field input and a list field indices of input’s good indices for tg. (For simplicity we use Haskell’s built-in lists notation 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 Haskell’s type checker statically ensures that only matches on the same target are appended together.

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

The identity method ε of SM target, for each target, returns the identity string (η) and the identity list (□).

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

The mappend method (⊕) of SM tg is explained in Figure 2, where the two string matchers SM xxis and SM yysis are appended. The returned input field is just x ⊕ y, while the returned indices field appends three list of indices: 1) the indices xxis on x casted to be good indices of the new input x ⊕ y, 2) the new indices yysis created when concatenating the two input strings, and 3) the indices yysis on y, shifted right lenStr x units. The Haskell definition of □ captures the creation of these three kinds of indices.

3Symbol is a kind and target is a singleton string type from GHC.TypeLits on Hackage.
Capturing the target $tg$ 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 old left indices, 2) creating new indices, and 3) shifting of the old right indices.

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

The definition of $\text{castGoodIndex}$ is a refinement type, safe cast on the argument $i$ that uses 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 $x$.

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

If the length of the $tg$ 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 $x \sqcup y$ for target $tg$ in the range from $\text{maxInt}(\text{lenStr} \ x - (\text{lenStr} \ tg - 1))$ to $\text{lenStr} \ x - 1$.

Generally, $\text{makeIndices} \ s \ tg \ lo \ hi$ returns the good indices of the input string $s$ for target $tg$ in the range from $lo$ to $hi$ by recursively checking one-by-one all the indices from $lo$ to $hi$.

3) Shift Good Indices  If $i$ is a good index for the string $y$ on the target $tg$, then shifting $i$ right $\text{lenStr} \ x$ units gives a good index for the string $x \sqcup y$ on $tg$, as encoded in the following function.

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

\[
\text{assume } \text{subStrAppendLeft} :: x : \text{RString} \to y : \text{RString} \\
\quad \to j : \text{Integer} \to i : \text{Integer} \to \\
\quad \{ \text{subStr} \ y \ i \ j \ = \ \text{subStr} \ (x \ \&\& \ y) \ (\text{lenStr} \ x + i) \ j \}
\]

### 5.2.1 String Matching is a Monoid

Next we prove that the methods \( \text{e} \) and \( \circ \) satisfy the monoid laws.

#### Theorem 5.1 (SM is a Monoid). \((\text{SM} \ t, \text{e}, \circ)\) 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:

\[
\begin{align*}
\text{idLeft} &:: x : \text{SM} \ t \to \{ \text{e} \circ x = xs \} \\
\text{idRight} &:: x : \text{SM} \ t \to \{ x \circ \text{e} = x \} \\
\text{assoc} &:: x : \text{SM} \ t \to y : \text{SM} \ t \to z : \text{SM} \ t \\
&\to \{ x \circ (y \circ z) = (x \circ y) \circ z \}
\end{align*}
\]

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

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

The first lemma says that shifting indices by the length of the empty string is an identity and is proved by induction on the input list.

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

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.

\[
\begin{align*}
\text{newIsNullLeft} :: s : \text{RString} \to t : \text{RString} \\
&\to \{ \text{makeNewIndices} \ \eta \ s \ t = [] \}
\end{align*}
\]

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 \( (x \circ y) \circ z \) and right \( x \circ (y \circ z) \) associative string matchers. To prove equality of the two string matchers we show that the input and indices fields are respectively equal. Equality of the input fields follows by associativity of RStrings. To prove equality of the index list we observe that irrespective of the mappend precedence, the indices can be split in five groups: the indices of the input \( x \), the new indices from mappending \( x \) and \( y \), the indices of the input \( y \), the new indices from mappending \( y \) and \( z \), and the indices of the input \( z \). After this observation the proof proceeds in three steps. First, we group the indices in the five lists indices using list associativity and distribution of casts. 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.

\[
\begin{align*}
\text{toSM} :: \forall (tg :: \text{Symbol}). \ (\text{KnownSymbol} \ tg) \\
&\to \text{RString} \to \text{SM} \ tg \\
\text{toSM} \ \text{input} = \text{SM} \ \text{input} \ (\text{go input \tg}') \\
\text{where}
&tg' = \text{fromString} \ (\text{symbolVal} \ (\text{Proxy} :: \text{Proxy} \ t)) \\
&\text{go \ x \ tg} = \text{makeIndices} \ x \ tg \ \emptyset \ (\text{lenStr} \ x - 1)
\end{align*}
\]

We prove in [29] that \( \text{toSM} \) is a monoid morphism.

#### Theorem 5.2. The function \( \text{toSM} \) is a morphism between the monoids \((\text{RString}, \eta, \circ)\) and \((\text{SM} \ t, \text{e}, \circ)\); since the below morphism function has a valid inhabitant.

\[
\begin{align*}
\text{morphismtoSM} :: x : \text{RString} \to y : \text{RString} \\
&\to \{ \text{toSM} \ \eta = \text{e} \ \&\&\& \ \text{toSM} \ (x \ \&\&\& \ y) = \text{toSM} \ x \ \&\&\& \ \text{toSM} \ y \}
\end{align*}
\]

#### 5.4 Parallel String Matching

Finally, we define \( \text{toSMPar} \) as a parallel version of \( \text{toSM} \), using machinery of §3, and prove that the sequential and parallel versions always give the same result.

\[
\begin{align*}
\text{toSMPar} :: \forall (tg :: \text{Symbol}). \ (\text{KnownSymbol} \ tg) \\
&\to \text{Integer} \to \text{Integer} \to \text{RString} \to \text{SM} \ tg \\
\text{toSMPar} \ i \ j \ x \ tg = \text{pmconcat} \ i \ \\text{. pmmap toSM} \ \text{. chunkStr} \ j
\end{align*}
\]

First, \( \text{chunkStr} \) splits the input into chunks of size \( j \). Then, \( \text{pmmap} \) applies \( \text{toSM} \) at each chunk in parallel. Finally, \( \text{pmconcat} \) concatenates the mappend chunks in parallel using 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 equal to \( \text{toSM} \ x \).

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

**Proof.** The proof follows by direct application of Theorem 3.3 on the chunkable monoid \((\text{RString}, \eta, \circ)\) (by Assumption 1) and the monoid \((\text{SM} \ t, \text{e}, \circ)\) (by Theorem 5.1).

\[
\begin{align*}
\text{correctness} \ i \ j \ x \\
&= \text{toSMPar} \ i \ j \ x \\
&= . \ \text{pmconcat} \ i \ (\text{pmmap toSM} \ (\text{chunkStr} \ j \ x)) \\
&= . \ \text{toSM} \ x \\
&. \ \text{parallelismEq} \ \text{toSM} \ \text{morphismtoSM} \ x \ i \ j
\end{align*}
\]

Note that application of the theorem \( \text{parallelismEq} \) requires a proof that its first argument \( \text{toSM} \) is a morphism. By Theorem 3.3, the required proof is provided as the function \( \text{morphismtoSM} \).

#### 6 String Matching in Coq

In this section we present the highlights of replicating the Liquid Haskell proof of correctness for the parallelization of a string matching algorithm into Coq.
6.1 Efficient vs. Verified Library Functions

In Liquid Haskell we used a wrapper around Bytestings 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 tg), then isGoodIndex input tg 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.

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

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

\[
\text{Definition isGoodIndexDec input tg i:}
\]
\[
\{\text{isGoodIndex input tg i} \} +
\]
\[
\{\text{isGoodIndex input tg i1}).
\]

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

6.3 Intrinsic vs. Extrinsic Verification

In § 4.1 we already discussed how Liquid Haskell favors intrinsic while Coq favor extrinsic verification. In the intrinsic, Liquid Haskell world the specifications come embedded into the functions and data types, while in Coq’s extrinsic world specifications and definitions are clearly separated. In the string matching proof we run into the case where intrinsic verification was unavailable 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, but it 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 to contain the input string and any list of natural numbers as indices.

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

Extrinsicly, 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:

\[
\text{Theorem sm_assoc sm prop sm2 sm3 :}
\]
\[
\text{validSM sm sm2 \rightarrow}
\]
\[
\text{prop sm \circ (sm2 \circ sm3) = (sm \circ sm2) \circ sm3}
\]

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 mappend string matching operator we intrinsically restrict the type of sm to carry a proof of valid indices.

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

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.

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

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, the Coq proof intrinsic reasoning (used to prove associativity) required the assumption of proof irrelevance. On the contrary in Liquid Haskell’s proof, specifications are intrinsically embedded in the definitions but their proofs are automatically and externally constructed by the SMT solver. In Liquid Haskell the user does not have access to the automatically generated proof terms, i.e., proof equality cannot even be specified (and is never required).

7 Evaluation

7.1 Quantitative Comparison

Table 1 summarizes the quantitative evaluation of our two proofs as implemented in [29]: the generalized equivalence property of parallelization of monoid morphisms and its application on the parallelization of a naïve string matcher. We used three provers to conduct our proofs: Coq, Liquid Haskell, and Liquid Haskell extended with the PLE (Proof by Logical 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...
Table 1. Quantitative evaluation. We report verification **Time** (in seconds) and LoC required to verify monoid morphism **parallelization** and its application to the **string matcher**. We split proofs of Coq (1136 LoC in total), Liquid Haskell (1428 LoC in total) and Liquid Haskell with PLE (1134 LoC in total) into **specifications**, **proof terms** and **executable** code.

<table>
<thead>
<tr>
<th>Property</th>
<th>Coq</th>
<th>Liquid Haskell</th>
<th>Liquid Haskell with PLE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Spec</td>
<td>Proof</td>
</tr>
<tr>
<td>Parallelization</td>
<td>5</td>
<td>121</td>
<td>329</td>
</tr>
<tr>
<td>String Matcher</td>
<td>33</td>
<td>127</td>
<td>437</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>38</td>
<td>248</td>
<td>766</td>
</tr>
</tbody>
</table>

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

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

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

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

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

**7.2 Qualitative Comparison**

We summarize the essential differences in theorem proving using Liquid Haskell versus Coq based on our experience (§ 4 and § 6).

These differences validate and illustrate the distinctions that have been previously [3, 22, 23] described between the two provers.

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

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

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

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

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

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

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

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

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

9 Conclusion

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

Acknowledgments

We thank Antal Spector-Zabusky, Arthur Azevedo de Amorim, George Karachalias, Nadari Sitchinava, Ranjit Jhala, Valentint Robert, and the anonymous reviewers for their insightful comments. This work was partially supported by Awake Networks and then NSF awards #1421243, #1521523, #1561209, and #1518844.

References