Functional Pearl: A Tale of Two Provers
Verifying Parallelized String Matching in Liquid Haskell and Coq

ANONYMOUS AUTHOR(S)

We demonstrate for the first time that refinement types can be used for arbitrary theorem proving by using Liquid Haskell, a refinement type checker for Haskell programs, to verify correctness of parallelization of a realistic Haskell string matcher. Refinement types have been extensively used for shallow type-based verification, but never before to prove arbitrary theorems about realistic programs. 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.

1 INTRODUCTION

"It was dependent types, it was refinement types."

Dependent type systems (like Coq (Bertot and Castéran 2004) or Adga (Norell 2007)) have been extensively used to specify theorems and machine-checked proofs about programs. These languages come with a plethora of libraries of theorems and tactics that facilitate interactive theorem proving. However, being geared towards verification, they provide minimal support for widespread features of general purpose languages like diverging computations, concurrency support or runtime-optimized libraries, some of which are only available via extraction.

Verification-oriented languages like Dafny (Leino 2010), F* (Swamy et al. 2016) and WhyML (Filliâtre and Paskevich 2013) combine good support for general purpose language features, like effectful and diverging computations, with semi-automated, SMT-based (Barrett et al. 2010) verification. Focused on verification, these languages lack the non-verified but highly optimized, real world libraries already developed in existing general purpose languages. All the above languages aim for highly expressive specifications, which makes SMT verification undecidable in theory (de Moura and Bjorner 2007) and unstable in practice (Pit-Claudel 2016).

Refinement types (Constable and Smith 1987; Freeman and Pfenning 1991; Rushby et al. 1998) on the other hand, extend *existing* general purpose languages (including ML (Bengtson et al. 2008; Rondon et al. 2008; Xi and Pfenning 1998), C (Condit et al. 2007; Rondon et al. 2010), Haskell (Vazou et al. 2014b), and Racket (Kent et al. 2016)) with *predictable* SMT-based verification. Traditionally, to achieve predictable verification, refinement types were limited to shallow specifications. For example, we use refinement types to specify that appending two lists `xs` and `ys` yields a list of length equal to the sum of the lengths of `xs` and `ys`:

```
append :: xs:[a] → ys:[a] → {v:[a] | length v = length xs + length ys}
```

However, expressing deeper properties, such as the associativity of append, which requires using append itself in the specification, has been out of reach. This restriction critically limited the expressiveness of the specifications, but allowed for both automatic and predictable SMT-based (Barrett et al. 2010) verification. Unfortunately, program equivalence proofs were beyond the expressive power of refinement types.

Liquid Haskell (Vazou et al. 2014b) extends refinement types with *refinement reflection* (Vazou and Jhala 2016), a technique that reflects each function's implementation into the function's type, turning the refined language into a theorem prover. In this paper we present the first non-trivial, 1428 LoC, application of Liquid Haskell as a theorem prover, by proving the correctness of the parallelization of a naïve string matching algorithm based on an axiomatization of primitive parallel combinators. We replicate this proof in Coq (1136 LoC) and empirically compare the two approaches.

The contributions of this paper are:

- 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 theorem prover: a verified parallelization of string matching. We do this by first formalizing monoid morphisms, and showing that such a morphism on a "chunkable" input can be correctly parallelized (§ 3) by:
  1. chunking 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.

  We then apply this result (§ 5) to a simple 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 simpler 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 FUNCTIONS AS PROOFS

Refinement reflection (Vazou and Jhala 2016) is a technique for writing Haskell functions which prove theorems about other Haskell functions and for machine-checking these proofs using Liquid Haskell (Vazou et al. 2014b). In this section, as an introduction to refinement reflection, we prove that lists form a monoid by

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

We start (§ 2.1) by defining a `List` datatype and the associated monoid elements ϵ and $\diamond$, corresponding to the empty list and concatenation. We then use refinement reflection to prove the three monoid laws (§ 2.2, § 2.4, and § 2.5) in Liquid Haskell. To simplify the proofs, we use the tactic *PSE (Proof by Static Evaluation)* (§ 2.3) that automatically expands logic terms. Finally (§ 2.5), 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 List [length] a = N | C {head :: a, tail :: List a}
```

The `length` annotation in the definition teaches Liquid Haskell to use the `length` function to check the termination of recursive list functions.

We define `length` as a standard Haskell function returning natural numbers.

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

But, what does Liquid Haskell know about `length`? 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 decicable abstraction of the Haskell function, to flow into the SMT logic. With this separation Liquid Haskell achieves decidable and predictable type checking, and prevents the potential instability of program verification that can be encountered when arbitrary recursive functions flow into logic (Pit-Claudel 2016).

However, this separation hinders precise verification. For instance, suppose we have a list with 3 elements : `xs = C 1 (C 2 (C 3 N))`. Liquid Haskell can prove that the length of `xs` is a natural number (`xs::{v : List a | 0 ≤ length xs}`), but based on `length`'s specification alone, it cannot prove that the length of `xs` is exactly 3. To increase precision Liquid Haskell provides two mechanisms, **`measure`** and **`reflect`**, to carefully lift Haskell functions into logic, while preserving SMT-decidable program verification.

The **`measure`** annotation lifts `length` into the logic by strengthening the types of the `List` data constructors. For example, the type of `C` is strengthened to

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

where `length` is an uninterpreted function in the logic. In general, measure (Vazou et al. 2014b) annotations are used to precisely lift into the SMT logic terminating, *unary* functions whose (1) domain is a data type and (2) body is a single case-expression over the datatype.

Then, we define and lift into the logic the two monoid operators on Lists: an identity element ϵ (which is the empty list) and an associative operator `(◇)` (which is list append).

```
reflect (◇)
(◇) :: List a → List a → List a
N        ◇ ys = ys
(C x xs) ◇ ys = C x (xs ◇ ys)
```

```
reflect ϵ
ϵ :: List a
ϵ = N
```

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

```
(ϵ) :: {v:List a | v = ϵ ∧ v = N}

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

Here, `(◇)` and `(ϵ)` are uninterpreted functions, and `isN`, `head` and `tail` are automatically generated measures. In general, reflect annotations are used to reflect terminating Haskell functions into the result of the function's type. After reflection, at each function call the function definition is unfolded exactly once into the logic, allowing Liquid Haskell to prove properties about Haskell functions.

2.2 Left Identity

In Liquid Haskell, we express theorems as refined type specifications and proofs as their Haskell inhabitants. We construct proof inhabitants using the combinators from the built-in `ProofCombinators` library that are summarized in Figure 1. A **`Proof`** is a unit type that when refined is used to specify theorems. A `trivial` proof is the unit value. For example, `trivial :: {v:Proof | 1 + 2 = 3}` trivially proves the theorem `1 + 2 = 3` using the SMT solver. The expression `p *** QED` casts any expression `p` into a **`Proof`**. The equality assertion `x ==. y` states that `x` and `y` are equal, while `thm ∴ lemma` proves `thm` using the `lemma`. Finally, `x∧.y` combines two proofs `x` and `y` into one by inserting the argument proofs into the logical environment.

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

trivial    :: Proof
trivial    = ()

(***)      :: a -> QED -> Proof
_ *** _    = ()
```

```
(==.) :: x:a -> y:{a | x = y} -> {v:a | v = x}
x ==. _ = x

(∴) :: (Proof -> a) -> Proof -> a
thm ∴ lemma = thm lemma

(∧.) :: Proof -> Proof -> Proof
_ ∧. _ = ()
```

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

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

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

Here, `{ϵ ◇ x = x}` is a simplification for the **`Proof`** type `{v:Proof | ϵ ◇ x = x}`, since the binder `v` is irrelevant. We begin from the left hand side `ϵ ◇ x`, which is equal to `N ◇ x` by calling `ϵ` thus unfolding the equality `empty = N` into the logic. The proof combinator `x ==. y` lets us equate `x` with `y` in the logic and returns `x` allowing us to continue the equational proof. Next, the call `N ◇ x` unfolds into the logic the definition of `(◇)` on `N` and `x`, which is equal to `x`, concluding our proof. Finally, we use the operators `p *** QED` which casts `p` into a proof term. In short, the proof of left identity, proceeds by unfolding the definitions of `ϵ` and `(◇)` on the empty list.

2.3 PSE: Proof by Static Evaluation

PSE (Proof by Static Evaluation) is a terminating but incomplete heuristic, inspired by (Pit-Claudel 2016), that Liquid Haskell uses to automatically unfold 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 ◇ ys` is unfolded to `ys` while `xs ◇ ys` is not unfolded when the structure of `xs` cannot be statically decided. Unlike SMT's heuristics (like E-matching (de Moura and Bjorner 2007; Moskal et al. 2008)) that make verification unstable (Pit-Claudel 2016), PSE is always terminating and is enabled on a per-function basis. 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 can be used to simplify the left identity proof by automatically unfolding `ϵ` to `N` and then `N ◇ x` to `x`. (We use the cornered one line code frame to denote Liquid Haskell proofs that use PSE.)

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

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

2.4 Right Identity

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

```
idRight_List :: x:List a → { x ◇ ϵ = x }
idRight_List N = N ◇ ϵ ==. N *** QED
```

```
idRight_List (C x xs)
   =   (C x xs) ◇ ϵ
   ==. C x (xs ◇ ϵ)
   ==. C x xs ∴ idRight_List xs
   *** QED
```

The recursive call `idRight_List xs` is provided as a third optional argument in the `(==.)` operator to justify the equality `xs ◇ ϵ = xs`, while the operator (∴) is merely a function application with the appropriate precedence. 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.

Once again, we can use the PSE tactic to automatically generate all function unfoldings and simplify the right identity proof.

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

2.5 Associativity

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

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

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

2.6 Lists are a Monoid

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

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

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

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

3 VERIFIED PARALLELIZATION OF MONOID 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. Such a morphism can be parallelized by:

- § 3.1 chunking up the input in pieces,
- § 3.2 applying the morphism in parallel to all chunks, and
- § 3.3 recombining the chunks, in parallel, back to a single value using the monoid operation.

In this section we implement and verify in Liquid Haskell the correctness of this transformation; we rely on the correctness of a single parallelization primitive (`pmap`) that is *assumed* to be correct.

3.1 Lists are Chunkable Monoids

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

```
lengthₘ :: m → Nat
dropₘ   :: i:Nat → x:{m | i ≤ lengthₘ x} → {v:m | lengthₘ v = lengthₘ x - i }
takeₘ   :: i:Nat → x:{m | i ≤ lengthₘ x} → {v:m | lengthₘ v = i }

take_drop_specₘ :: i:Nat → x:m → {x = takeₘ i x ◇ dropₘ i x}
```

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

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

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

chunkₘ :: i:Pos → x:m → {v:List m | chunk_specₘ i x v } / [lengthₘ x]
chunkₘ i x
  | lengthₘ x ≤ i = C x N
  | otherwise      = takeₘ i x `C` chunkₘ i (dropₘ i x)
```

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

```
chunk_specₘ i x v
  | lengthₘ x ≤ i = length v == 1
  | i == 1         = length v == lengthₘ x
  | otherwise      = length v <  lengthₘ x
```

Liquid Haskell uses the specifications of both $\texttt{take}_m$ and $\texttt{drop}_m$ to automatically verify the $\texttt{length}_m$ constraints imposed by $\texttt{chunk\_spec}_m$.

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

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

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

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

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

3.2 Parallel Map

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

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

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

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

Moreover, to reflect the implementation of pmap in the logic, withStrategy should also be represented in the logic. LiquidHaskell encodes withStrategy in the logic as a logical, *i.e.,* total, function that merely returns its second argument, withStrategy _ x = x. That is, our proof does not reason about parallelism; we assume correctness of the Haskell's library parallelization primitive. Thus, strategy chosen does not affect verification.

Under this encoding, the strategy parStrategy does not affect verification. In our codebase we choose the traversable strategy.

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

3.3 Parallel Monoidal Concatenation

The function $\mathtt{chunk}_m$ lets us turn a monoidal value into several pieces. In the other direction, for any monoid (m, ϵ, $\diamond$), the monoid concatenation $\mathtt{mconcat}_m$ turns a List m back into a single m.

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

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

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

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

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

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

```
pmconcatEquivalence :: i:Int → x:List m → { pmconcat_m i x = mconcat_m x }
```

PROOF. We prove the theorem by providing an implementation of `pmconcatEquivalence` that satisfies its refinement type specification. The proof proceeds by structural induction on the input list `x`. The details of the proof can be found in (Supplementary-Material 2017), here we sketch the proof.

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

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

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

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

```
mchunk :: i:Int → x:List m → {mconcat_m x = mconcat_m (map mconcat_m (chunk i x))}
```

Both lemmata are proven by structural induction on the input list `x`. Lemma `mchunk` is sufficient to prove `pmconcatEquivalence` by structural induction, using monoid left identity in the base case. □

3.4 Parallel Monoid Morphism

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

THEOREM 3.3 (CORRECTNESS OF PARALLELIZATION). *Let* (m, ϵ, $\diamond$) *be a monoid and* (n, η, $\boxdot$) *be a chunkable monoid. Then, for every morphism* `f :: n → m`*, every positive number* `i` *and* `j`*, and input* `x`*,* `f x = pmconcat i (pmap f (chunk_n j x))` *holds.*

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

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

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

PROOF. We prove the equivalence in two steps. First we prove a lemma (`parallelismLemma`) that the equivalence holds when the mapped result is concatenated sequentially. Then, we use the lemma to prove parallelism equivalence by the definition of a valid inhabitant for `parallelismEquivalence`.

LEMMA 3.4. *Let* (m, ϵ, $\diamond$) *be a monoid and* (n, η, $\boxdot$) *be a chunkable monoid. Then, for every morphism* `f : n → m`*, every positive number* `i` *and input* `x`*,* `f x = mconcat_m (pmap f (chunk_n i x))` *holds.*

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

PROOF. We prove the lemma by providing an implementation of `parallelismLemma` that satisfies its type. The proof proceeds by induction on the length of the input.

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

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

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

```
parallelismEquivalence f thm x i j
  =   pmconcatEquivalence i (pmap f (chunk_n j x)) ∧. parallelismLemma f thm x j
```

□

4 MONOID MORPHISM PARALLELIZATION IN COQ

To put Liquid Haskell as a theorem prover into perspective, we replicated the proof of the Parallel Monoid Morphism (Theorem 3.3) in the Coq proof assistant. A more comprehensive comparison follows the string matching case study in § 6. In this section we focus on the differences that appeared while proving the correctness of monoid morphism parallelization in the two provers.

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 `drop_spec` and `take_spec` methods. (We use the doubled lined code frame for Coq code.)

```
length_m : M → nat;
drop_m   : nat → M → M;
take_m   : nat → M → M;

drop_spec_m      : ∀ i x, i ≤ length_m x → length_m (drop_m i x) = length_m x - i;
take_spec_m      : ∀ i x, i ≤ length_m x → length_m (take_m i x) = i;
take_drop_spec_m : ∀ i x, x = take_m i x ◇ drop_m i x;
```

Liquid Haskell favors intrinsic verification, as the shallow specifications of `take` and `drop` are embedded into the functions and automatically proven 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, so that 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 (Gonthier and Mahboubi 2009)—to define the chunkable monoid operations that had to be defined from scratch in Liquid Haskell (§ 3.1).

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

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

```
Lemma size_drop s : size (drop n s) = size s - n.

Theorem drop_spec_list :
  ∀ i x, i ≤ length_list x → length_list (drop_list i x) = length_list x - i.
Proof. by apply seq.size_drop. Qed.
```

4.3 SMT- vs Tactic-Based Automation

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

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

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

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

However, the existing lemma and our desired specification differ when i is exactly equal to size x, creating the following proof obligation when rewriting with seq.size_take and branching on whether i is less than or equal to the size of x.

```
A : Type
i : nat
x : seq A
HSize : i ≤ size x
Size : (i < size x) = false
============================
size x = i
```

We are left to prove that if i is less than or equal to x, but not strictly less than x, then the two numbers are equal. To discharge this kind of obligations in our implementation (Supplementary-Material 2017) we resort to an adaptation of the advanced Pressburger Arithmetic solver omega (Pugh 1991) for ssreflect.

Of course, this example is trivial enough that it could be handled automatically without resorting to a very heavy tactic like omega. However, throughout our development we encountered multiple times the need to reason about arithmetic properties. 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), the user has to understand the reason of failure and manually complete the proof. Worse, the proofs generated by omega are far from ideal; to quote the Coq Reference Manual (Coq development team 2009) : "The simplification procedure is very dumb and this results in many redundant cases to explore. Much too slow."

4.4 Semantic vs. Syntactic Termination Checking

Since non-terminating programs introduce inconsistencies in the logic, all reflected Haskell functions and all Coq programs are provably terminating. A first difference between termination checking in the two provers is that Liquid Haskell allows non-reflected, Haskell functions (that do not flow into the logic) to be potentially diverging (Vazou et al. 2014b), while Coq, that does not explicitly distinguish between logic and implementation, does not, by default, support partial computations (Danielsson 2012). Making such a distinction between logic and implementation in a dependently typed setting is in fact a research problem of its own (Casinghino et al. 2014).

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

```
Fixpoint chunk_m {M: Type} (fuel : nat) (i : nat) (x : M) : option (list M) :=
  match fuel with
    | O ⇒ None
    | S fuel' ⇒
      if length_m x ≤ i then Some (cons x nil)
      else match chunk_m fuel' i (drop_m i x) with
        | Some res ⇒ Some (cons (take_m i x) res)
        | None ⇒ None
      end
  end.
```

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

```
Theorem chunk_spec_m : ∀ {M} i (x : M) ,
   i > 0 → exists l, chunk_m (length_m x).+1 i x = Some l /\ chunk_res_m i x l.
```

The specification of chunk 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 recursive definitions, heavily used in CompCert (Leroy 2006). Adam Chlipala, in his book "Certified Programming with Dependent Types" (Chlipala 2013), compares three such general techniques to bypass Coq's syntactic termination restriction, namely well-founded recursion, domain-theory-inspired non-termination monads (where our fuel-based approach can be roughly categorized), and co-inductive non-termination monads. However, no single method is found to be ideal.

4.5 Executable vs Axiomatized Parallelism

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

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

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

5 CASE STUDY: CORRECTNESS OF PARALLEL STRING MATCHING IN LIQUID HASKELL

In this section we apply the parallization equivalence theorem of § 3 to parallelize a realistic, efficient string matcher. We define a string matching function `toSM :: RString → SM target` from Refined Strings `RString` to a monoidal, string matching data structure `SM target`. In § 5.1 we assume that `toSM`'s domain, *i.e.,* the Refined String that is a wrapper in Haskell's optimized `ByteString`, is a chunkable monoid. Then, in § 5.2 we prove that `toSM`'s range, *i.e.,* `SM target`, 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 Refined Strings are assumed to be Chunkable Monoids

We define the type `RString` to be a wrapper on Haskell's existing, optimized, constant-indexing `ByteString`[1]

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

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

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

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

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

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

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

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

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,

[1] We use `BS` as a namespace for the `ByteString` module.

Assumption 5.1 (RString is a Chunkable Monoid). *(*`RString`*, η, $\boxdot$) combined with the methods* `lenStr`*,* `takeStr`*,* `dropStr` *and* `takeDropPropStr` *is a chunkable monoid.*

We note that actually proving that `ByteString` implements a chunkable monoid in Liquid Haskell is possible, as implied by (Vazou et al. 2014a), but it is both time consuming and orthogonal to our purpose. Instead, here we follow the easy route of axiomatization – demonstrating that refinement reflection is capable of doing gradual verification.

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 target`, for the codomain of a string matching function, where `target` is the string being looked for. Concretely, we define the data type `SM target` in § 5.2.1, the monoid methods for identity and mappend in § 5.2.2, and prove that these methods satisfy the monoid laws in § 5.2.3.

5.2.1 The String Matching Data Structure. 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.

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

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

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

For example, 2 and 5 are good indices of `"abcab"` on `"ababcabcab"` so Liquid Haskell will accept the assignment

`2, 5 :: GoodIndex "ababcabcab" "abcab"`.

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

We use the string type literal [2] 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.

5.2.2 The Monoid Methods. 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* ($\diamond$) of `SM target` is explained in Figure 2, where the two string matchers `SM x xis` and `SM y yis` are appended. The returned input field is just `x ⊡ y`, while the returned indices field appends three

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

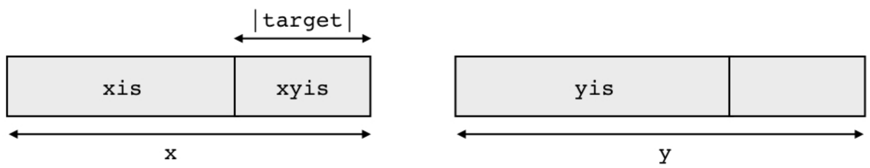

Fig. 2. Mappend indices of String Matcher

list of indices: 1) the indices `xis` on `x` casted to be good indices of the new input `x ⊡ y`, 2) the new indices `xyis` created when concatenating the two input strings, and 3) the indices `yis` on `y`, shifted right `lenStr x` units. The Haskell definition of ◇ captures the above three indexing operations.

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

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

1) Cast Good Indices. If `i` is a good index for the string `x` on the target `tg`, then `i` is also a good index for the string `x ⊡ y` and the same target, for any `y`. This property is encoded in the function `castGoodIndex`.

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

castGoodIndex tg x y i = subStrAppendRight x y (lenStr tg) i `cast` i
```

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

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

Refinement type casting is performed via the function `cast p x` that returns `x`, after enforcing the properties of `p` in the logic.

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

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

2) Creation of new indices. Appending two input strings `x` and `y` may create new good indices, *i.e.,* the indices `xyis` in Figure 2. For instance, appending `"ababcab"` with `"cab"` leads to a new occurrence of `"abcab"` at index 5 which does not occur in either of the two input strings. These new good indices can appear only at the last `lenStr tg - 1` positions of the left input `x`. `makeNewIndices x y tg` detects all such good new indices.

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

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

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

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

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

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

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

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

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

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

Theorem 5.2 (SM is a Monoid). *(`SM t`, ϵ, $\diamond$) 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. First, we prove *left identity* using PSE, left identity on string and list and two helper lemmata.

```
idLeft :: x:SM t → {ϵ ◇ x = xs}
idLeft (SM i is)
  =  idLeftStr i ∧. idLeftList is ∧. mapShiftZero tg i is ∧. newIsNullLeft i tg
  where tg = fromString (symbolVal (Proxy :: Proxy t))
```

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

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

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

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

Next, we prove *right identity* using PSE, right identity on string and list and two helper lemmata.

```
idRight :: x:SM t → {x ◇ ϵ = x}
idRight (SM i is)
  =  idRightStr i ∧. idRightList is ∧. mapCastId tg i η is ∧. newIsNullRight i tg
  where tg = fromString (symbolVal (Proxy :: Proxy t))
```

The first helper lemma states that casting is an identity and is proven by induction on the index list is. Identity of casting is proven

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

The second helper lemma states than appending with the empty string creates no new indexes and is proven by case splitting on the relative length of the input string s and the target t. At each case the potential new indices would be out of bounds and thus no new good indices would be created.

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

Finally we prove *associativity*. The PSE strategy failed to automatically prove associativity due to the complexity of the proof. For space, we only provide a proof sketch while the complete proof is available in (SupplementaryMaterial 2017). Our goal is to show equality of the left and right associative string matchers.

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

To prove equality of the two string matchers we show that the input and indices fields are respectively equal. Equality of the input fields follows by associativity of RStrings. To prove equality of the index list we observe, as depicted in Figure 3, 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 x and y, and the indices of the input z. After this observation the proof proceeds in three steps.

(1) First, we group the indices in the five lists of Figure 3, using list associativity and distribution of index shifting.
(2) Then, we prove equivalence of different group representations, since the representation of each group depends on the order of appending. For example, if zis1 (resp. zis2) is the group zis when right (resp. left) mappend happened first, then we have

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

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

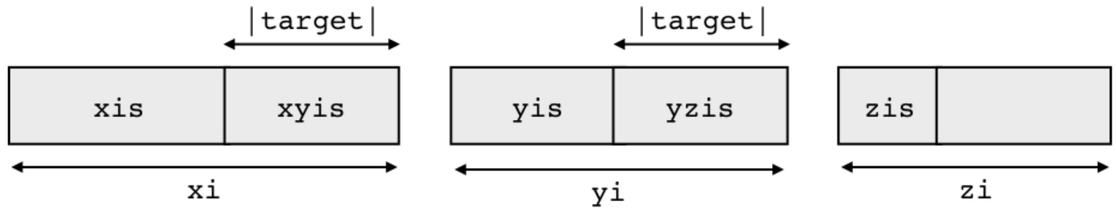

Fig. 3. Associativity of String Matching

That is, in `zis1` the indices of `z` are first shifted by the length of `yi` and then by the length of `xi`, while in `zis2` the indices of `z` are shifted by the length of `xi ⊡ yi`. We proved that `zis1` is equal to `zis2` by induction. We had to prove equivalence of all the three middle groups together by case analysis on the relative lengths of the target `tg` and the middle string `yi`.

(3) Finally, we wrap the index groups back to string matchers using list associativity and distribution of casts.

The detailed proof can be found in (Supplementary-Material 2017).

□

5.3 String Matching Monoid Morphism

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

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

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

The input field of the result is the input string; the indices field is computed by calling `makeIndices` within the range of the `input`, that is from `0` to `lenStr input - 1`. We now prove that `toSM` is a monoid morphism.

Theorem 5.3. *The function* `toSM` *is a morphism among the monoids* (`RString`, η, ⊡) *and* (`SM t`, ϵ, ◇).

Proof. To prove the theorem we define the function `morphismtoSM` as a valid inhabitant of the `Morphism` specification from Theorem 3.3.

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

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

```
morphismtoSM x y
  =   (toSM x :: SM target) ◇ (toSM y :: SM target)
  ==. (SM x is1) ◇ (SM y is2)
  ==. SM i (xis ++ xyis ++ yis)
  ==. SM i (makeIndices i tg 0 hix ++ makeIndices i tg (hix + 1) hi)
      ∴ mapCastId tg x y is1 ∧. mergeNewIndices tg x y ∧. shiftIndices 0 hiy x y tg
  ==. SM i (makeSMIndices i tg) ∴ mergeIndices i tg 0 hix hi
```

```
  ==. toSM (x ⊡ y)
  *** QED
  where xis  = map (castGoodIndex    tg x y) is1
        xyis = makeNewIndices x y tg
        yis  = map (shiftStringRight tg x y) is2
        tg   = fromString (symbolVal (Proxy::Proxy target))
        is1  = makeSMIndices x tg
        is2  = makeSMIndices y tg
        i    = x ⊡ y
        hix  = lenStr x - 1
        hiy  = lenStr y - 1
        hi   = lenStr i - 1
```

Our proof is using the following four lemmata.

- `mergeIndices` states that for every input `x` and target `tg` if we append the indices in the range from `lo` to `mid` with the indices in the range from `mid+1` to `hi`, we get exactly the indices in the range from `lo` to `hi`. The proof proceeds by induction on `mid`.
- `mergeNewIndices` states that appending the indices `xis` and `xyis` is equivalent to the good indices of `x ⊡ y` from `0` to `lenStr x - 1`. The proof case splits on the relative sizes of `tg` and `x` and is using `mergeIndices` on `mid = lenStr x1 - lenStr tg` in the case where `tg` is smaller than `x`.
- `mapCastId` states that casting a list of indices returns the same list.
- `shiftIndices` states that shifting right `i` units the indices from `lo` to `hi` is equivalent to computing the indices from `i + lo` to `i + hi` on the string `x ⊡ y`, with `lenStr x = i`.

□

5.4 Parallel String Matching

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

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

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

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

Theorem 5.4 (Correctness of Parallel String Matching). *For each parameter* `i` *and* `j`*, and input* `x`*,* `toSMPar i j x` *is always equal to* `toSM x`*.*

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

Proof. The proof follows by direct application of Theorem 3.3 on the chunkable monoid (`RString`, η, ⊡) (by Assumption 5.1) and the monoid (`SM t`, ϵ, ◇) (by Theorem 5.2).

```
correctness i j x
  =   toSMPar i j x
```

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

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

6 STRING MATCHING IN COQ

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

6.1 Efficient vs Verified Library Functions

In Liquid Haskell we used a wrapper around `ByteStrings` to represent efficient but unverified string manipulation functions. Thus, we had to assume 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 proper extraction, *e.g.,* extracting `String` to `ByteSting` at runtime.

6.2 Separate vs Inlined Lemmata

In Liquid Haskell, induction is encoded via recursive function calls. This encoding highly restricts the structure of the Liquid Haskell proofs as each inductive property should be separately encoded and proved as a top level Haskell function. On the contrary, Coq does not impose any such restriction, allowing the user to prove inductive lemmata inlined in the proof via the `induction` tactic.

6.3 Semantic vs. Syntactic Termination Checking

In § 4.4 we discussed how Coq's syntactic (unlike Liquid Haskell's semantic) termination checking requires special care for on functions that are recursive over a non syntactic metric. In the string matching proof, we encountered a characteristic example that turns non-structural to structural recursion by calculating in advance the number of recursive steps. In § 5.2.2 we defined `makeIndices s tg lo hi` to compute all the good indexes on `s` by checking whether `lo` is a good index and recursively calling itself on `lo+1` until `lo` reaches `hi`. Semantic termination of `makeIndices` is specified by the termination metric `(hi - lo)`. In Coq we make indices using a function `makeIndices s tg lo counter` that instead of using the higher bound `hi` is using the syntactically decreasing natural number `counter` that is initialized to `hi - lo`.

6.4 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 § 5.2.1, 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, Haskell execution. On the other hand, Coq separates 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 poses difficulties when the logical specifications also need to be executed. For example, to ease reasoning, we defined `isGoodIndex` to be in **`Prop`**.

```
Definition isGoodIndex (input tg : string) (i : nat) :=
  (substring i (length tg) input) = tg /\ i + length tg ≤ length input.
```

Yet, 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` [3] .

```
Definition isGoodIndexDec input tg i:
  {isGoodIndex input tg i} + {~ (isGoodIndex input tg i)}.
Proof. (* Proof omitted for simplicity *) Qed.
```

Instead of returning a simple boolean, the decidability procedure returns the proof carrying, executable sum type:

```
Inductive sumbool (A B : Prop) : Set := left : A → {A} + {B} | right : B → {A} + {B}
```

When extracted into OCaml or Haskell, `sumbool` is isomorphic to `Bool`; however, in Coq each constructor `left` and `right` carries additional proof information. This means that while the basic structure of the decidability procedure is straightforward (deciding whether both branches of the conjunction hold or not), it also contains additional content to construct appropriate proof terms.

6.5 Intrinsic vs Extrinsic Verification

In § 4.1 we briefly 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. When porting the string matching proof into Coq the difference in these two verification methods became more interesting. In § 5.2.1 we defined the Liquid Haskell string matcher `SM target` 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 let us rest assured that each string matcher only contains valid indices, but forces us to litter the implementation code with refinement type casts (*i.e.,* `castGoodIndex` in the definition of `(◇)`) that persuaded Liquid Haskell that all indices are valid string matching indices. When porting the string matching proof to Coq, to keep implementation clean from proofs, we followed an extrinsic approach.

6.5.1 The Extrinsic Approach. Following the extrinsic approach in Coq, we defined the string matcher data type on a target `tg` to contain the input string and any list of natural numbers as indices.

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

Extrinsically, we specified that a string matcher `SM tg` is valid when the `indices` list contains only valid indices (where `List.Forall` asserts that `isGoodIndex input tg` holds for all elements of the list `l`).

```
Inductive validSM tg : SM tg → Prop :=
  | ValidSM : ∀ input l,
     List.Forall (isGoodIndex input tg) l → validSM tg (Sm tg input l).
```

This extrinsic approach allows for cleaner implementations of (a first version of) the mappend operator $(\hat{\diamond})$

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

```
Definition ◇̂ {tg} (sm1 sm2 : SM tg) :=
  let '(Sm x xis) := sm1 in
  let '(Sm y yis) := sm2 in
  let xyis := makeNewIndices x y tg in
  let yis' := map (shiftStringRight tg x y) yis in
  Sm tg (x ⊡ y) (xis ++ xyis ++ yis').
```

The main difference between the above definition and the Liquid Haskell definition of § 5.2.2 is that the indices `xis` are unchanged: no validity casting is required since implementation is free from the validity requirement.

As another benefit, the extrinsic approach clarifies exactly when the correctness assumptions are necessary. For example, the associativity proof of (◇̂) only requires the middle string matcher to be valid:

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

Clarification of the exact correctness assumptions is a double-edged sword. Even though it is informative to explicitly specify that validity of the middle string matcher is required, the `sm_assoc` theorem highlights that the extrinsic (◇̂) does not satisfy the associativity monoid law, as it comes with the extra validity assumption.

6.5.2 The intrinsic Approach. To define an associative mappend string matching operator we restrict the type of `sm` to contain the `input` string, the list of `indices` and also to intrinsically carry a proof of their validity.

```
Inductive sm tg : Type :=
| mk_sm : ∀ input indices, List.Forall (isGoodIndex input tg) indices → sm tg.
```

Dependent Pattern Matching The implementation of the monoidal operation for `sm` exemplifies one inconvenience of intrinsic verification. Ideally, we would like to reuse the definition and properties of the non-associative (◇̂) to define a correct (◇) in the lines of the following code, where `<proof>` would be filled in by some auxiliary validity lemma.

```
Definition ◇ {tg} sm1 sm2 :=
  match sm1, sm2 with
  | mk_sm xs1 l1 H1, mk_sm xs2 l2 H2 ⇒
     match (Sm tg xs1 l1) ◇̂ (Sm tg xs2 l2) with
     | Sm xs' l' ⇒ mk_sm tg xs' l' <proof>
     end
  end
```

However, dependent pattern matching in Coq does not, by default, provide an equality between `(Sm tg xs1 l1) ◇̂ (Sm tg xs2 l2)` and `Sm xs' l'` in scope for `<proof>`. Instead, the user must resort to what is known as the *convoy pattern* (Chlipala 2013): the result of the inner match becomes a function that takes evidence of the needed equality as an argument, while the entire match is applied to such evidence.

```
Definition ◇ {tg} sm1 sm2 :=
  match sm1, sm2 with
  | mk_sm xs1 l1 H1, mk_sm xs2 l2 H2 ⇒
      let s := (Sm tg xs1 l1) ◇̂ (Sm tg xs2 l2) in
      let App := erefl s in
      (match s as s0 return ((Sm tg xs1 l1) ◇̂ (Sm tg xs2 l2) = s0 → sm tg)
```

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

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.

```
      with
      | Sm xs' l' ⇒ fun _ ⇒ mk_sm tg xs' l' _
      end) App
  end
```

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

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

We prove the above lemma using *Proof Irrelevance*, an admittable axiom, consistent with Coq's logic, which states that any two proofs of the same property are equal.

```
Axiom proof_irrelevance : ∀ (P : Prop) (p1 p2 : P), p1 = p2
```

Thus, in our Coq proof intrinsic reasoning (used to prove associativity) required the assumption of proof irrelevance. This was not the case for Liquid Haskell's verification, where 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, not to mention required.

7 EVALUATION

7.1 Quantitative Comparison.

Table 1 summarizes the quantitative evaluation of our two proofs: 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 a Liquid Haskell expert within 2 months. Most of this time was spend 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 an experiend Coq user 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-development, 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, leading to faster Liquid Haskell verification.

Verification size. We split the total numbers 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 in both provers 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 in `chunk` and `p◇`. 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 application. Finally, the ratio drops to 5x when the PSE heuristic is used, as the proof terms are shrinked 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). These differences validate and illustrate the distinctions that have been previously (Casinghino et al. 2014; Rondon et al. 2008; Swamy et al. 2016) described between refinement and dependent types.

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

SMT-automation vs. Tactics. Liquid Haskell uses an SMT-solver to automate proofs over decidable theories (such as linear arithmetic, uninterpreted functions); this reduces the proof burden compared to fully interactive proofs, 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 Pressburger 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, should be proven (automatically by SMTs) at function definitions, and are assumed at function calls. Coq provides a choice between intrinsic and extrinsic verification, with the latter being more common: extrinsic verification separates the functionality of definitions from their specifications, which can then be independently proven (§ 4.1), thus making function definitions cleaner.

Semantic vs. Syntactic Termination Checking. Liquid Haskell uses a semantics termination checker that proves termination given a wellfounded termination metric. On the contrary, Coq allows fixpoints to be defined only by using syntactical subterms of some principal argument in recursive calls. For definitions outside of this restrictive recursion pattern, one must resort to advanced transformation techniques as discussed in § 4.4 and § 6.3.

8 RELATED WORK

SMT-Based Verification. SMT solvers have been used to automate reasoning on verification oriented languages like Dafny (Leino 2010), F* (Swamy et al. 2016) and Why3 (Filliâtre and Paskevich 2013). These languages are designed for verification, thus have limited support for commonly used language features like parallelism and optimized libraries that we use in our verified implementation. All these languages allow for highly expressive specifications, which makes SMT verification undecidable in theory (de Moura and Bjorner 2007) and unstable in practice (Pit-Claudel 2016). Refinement Types (Freeman and Pfenning 1991) on the other hand, extend existing general purpose languages to allow decidable specifications. That is, before Refinement Reflection (Vazou and Jhala 2016) was introduced, refinement types only allowed "shallow" program specifications, *i.e.,* properties that only talk about behaviors of program functions but not functions themselves.

Dependent Types. Unlike Refinement Types, dependent type systems, like Coq (Bertot and Castéran 2004), Adga (Norell 2007) and Isabelle/HOL (Paulson 1994) allow for "deep" specifications which talk about program functions, such as the program equivalence reasoning we presented. Compared to (Liquid) Haskell, these systems allow for tactics and heuristics that automate proof term generation but lack SMT automations and general-purpose language features, like non-termination and exceptions. Zombie (Casinghino et al. 2014) and F* (Swamy et al. 2016) allow dependent types to coexist with divergent and effectful programs, but still lack the optimized libraries, like `ByteSting`, which come with a general purpose language with long history, like Haskell.

Haskell itself is becoming a dependently typed language. On an ongoing line of work, Eisenberg (2016) 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 (Lindley and McBride 2013). 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 string matching proof already depends on Haskell's type level strings to encode the monoid string matcher.

Parallel Code Verification. Dependent type theorem provers have been used before to verify parallel code. BSP-Why (Fortin and Gava 2015) is an extension to Why2 that is using both Coq and SMTs to discharge user specified verification conditions. Daum (Daum 2007) used Isabelle to formalize the semantics of a type-safe subset of C, by extending Schirmer's (Schirmer 2006) formalization of sequential imperative languages. Finally, Swierstra (Swierstra 2010) formalized mutable arrays in Agda to reason about distributed maps and sums.

One closely related work is SyDPaCC (Loulergue et al. 2016), 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. Our Coq development would require similar additional non-Coq code if we were to extract it to obtain an executable program; the Liquid Haskell development does not.

9 CONCLUSION

We used Liquid Haskell as a theorem prover to verify, for first time, correctness of a realistic Haskell program: parallelization of a string matcher. We ported our 1428 LoC proof to Coq (1136 LoC) and compared the two provers. We conclude that the strong points of Liquid Haskell as a theorem prover are: 1) the proof refers to executable Haskell code directly using advanced Haskell features like optimized libraries and parallel code and 2) the proof is SMT-automated over decidable theories (like linear arithmetic). On the other hand, Coq allows for an extrinsic approach to verification that leads to cleaner code and proofs, while it aids verification providing a large pool of already developed theorems, tactics, and methodologies that the user can lean on.

REFERENCES

C. Barrett, A. Stump, and C. Tinelli. 2010. The SMT-LIB Standard: Version 2.0.

J. Bengtson, K. Bhargavan, C. Fournet, A.D. Gordon, and S. Maffeis. 2008. Refinement Types for Secure Implementations. In *CSF*.

Y. Bertot and P. Castéran. 2004. *Coq'Art: The Calculus of Inductive Constructions.* Springer Verlag.

C. Casinghino, V. Sjöberg, and S. Weirich. 2014. Combining proofs and programs in a dependently typed language. In *POPL*.

A. Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press.

J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula. 2007. Dependent Types for Low-Level Programming. In *ESOP*.

R. L. Constable and S. F. Smith. 1987. Partial Objects In Constructive Type Theory. In *LICS*.

The Coq development team. 2009. *The Coq proof assistant reference manual.* http://coq.inria.fr/doc/

N. A. Danielsson. 2012. Operational semantics using the partiality monad. In *ICFP*.

M. Daum. 2007. Reasoning on Data-Parallel Programs in Isabelle/Hol. In *C/C++ Verification Workshop*.

L. de Moura and N. Bjorner. 2007. Efficient E-matching for SMT Solvers. In *CADE*.

R. A. Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practise.* Ph.D. Dissertation. UPenn.

J. Filliâtre and A. Paskevich. 2013. Why3 – Where Programs Meet Provers. In *ESOP*.

J. Fortin and F. Gava. 2015. BSP-Why: A tool for deductive verification of BSP algorithms with subgroup synchronisation. In *Int J Parallel Prog*.

T. Freeman and F. Pfenning. 1991. Refinement Types for ML. In *PLDI*.

G. Gonthier and A. Mahboubi. 2009. *A Small Scale Reflection Extension for the Coq system.* Technical Report Research Report Number 6455. Microsoft Research INRIA. http://hal.inria.fr/docs/00/84/38/81/PDF/main.pdf

A. M. Kent, D. Kempe, and S. Tobin-Hochstadt. 2016. Occurrence typing modulo theories. In *PLDI*.

K. R. M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness *(LPAR)*.

X. Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL fi06*.

S. Lindley and C. McBride. 2013. Hasochism: the pleasure and pain of dependently typed haskell programming. In *Haskell*.

F. Loulergue, W. Bousdira, and J. Tesson. 2016. Calculating Parallel Programs in Coq using List Homomorphisms. In *International Journal of Parallel Programming*.

M. Moskal, J. Lopuszański, and J. R. Kiniry. 2008. E-matching for Fun and Profit. In *Electron. Notes Theor. Comput. Sci.*

U. Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph.D. Dissertation. Chalmers.

L. C. Paulson. 1994. Isabelle - A Generic Theorem Prover. *Lecture Notes in Computer Science* (1994).

K. R. M. Leinoand C. Pit-Claudel. 2016. Trigger Selection Strategies to Stabilize Program Verifiers. *CAV* (2016).

W. Pugh. 1991. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Supercomputing*.

P. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid Types. In *PLDI*.

P. Rondon, M. Kawaguchi, and R. Jhala. 2010. Low-Level Liquid Types. In *POPL*.

J. Rushby, S. Owre, and N. Shankar. 1998. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE TSE* (1998).

N. Schirmer. 2006. *Verification of Sequential Imperative Programs in Isabelle/HOL.* Ph.D. Dissertation. TU Munich.

Non-Anonymous Supplementary-Material. 2017. Code for Verified String Indexing. Provided in Non-anonymous Supplementary Material.

N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. Zinzindohoue, and S. Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *POPL*.

W. Swierstra. 2010. More dependent types for distributed arrays. *Higher-Order and Symbolic Computation* (2010).

N. Vazou and R. Jhala. 2016. Refinement Reflection *(arXiv:1610.04641)*.

N. Vazou, E. L. Seidel, and R. Jhala. 2014a. LiquidHaskell: Experience With Refinement Types in the Real World. In *Haskell Symposium*.

N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. 2014b. Refinement Types for Haskell. In *ICFP*.

H. Xi and F. Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types.. In *PLDI*.