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 [4] or Adga [20]) have been extensively used to specify theorems and machine-checked proofs about programs. These systems are equipped 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 [15], F* [28] and WhyML [11] combine good support for general purpose language features, like effectful and diverging computations, with semi-automated, SMT-based [2] verification. Focused on verification, these languages lack the, non-verified but highly optimized, real world libraries already developed in existing general purpose languages. Moreover, all the above languages aim for highly expressive specifications, which makes SMT verification undecidable (in theory [10]) and unstable (in practice [16]).

Refinement Types [7, 13, 24] on the other hand, extend *existing* general purpose languages (including ML [3, 22, 33], C [6, 23], Haskell [31], Racket [14] and Scala [26]) with *predictable* SMT-based verification. Traditionally, to achieve predictable verification, refinement types were limited to shallow specifications. For example, we could use refinement types to specify that 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},
```

but not to express deeper properties, such as the associativity of append, that require using append itself in the specification. This restriction critically limited the expressiveness of the specifications, but allowed for both automatic and predictable SMT-based [2] verification. Unfortunately, program equivalence proofs were beyond the expressive power of refinement types.

Liquid Haskell [31] extends refinement types with *refinement eflection* [32], 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 application of Liquid Haskell (1428 LoC) 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 formalize monoid morphisms and show 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 apply this result (§ 5) to obtain the first large application of Liquid Haskell as theorem prover: a verified parallelization of string matching.
- 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 [32] is a technique for writing Haskell functions which prove theorems about other Haskell functions and for machine-checking these proofs using Liquid Haskell [31]. 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 [32] 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 the 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 unstable program verification [16] encountered when arbitrary recursive functions flow into logic. However, this separation hinters precise verification. For instance, assume a list with 3 elements `xs = C 1 C 2 C 3 N`. Liquid Haskell can prove that the lengths of `xs` is a natural number, `xs::{v : List a | 0 ≤ length xs}` but based on `lenght`'s specificatio alone, cannot prove that the length of `xs` is exactly 3 `xs::{v : List a | 3 = length xs}`. 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 [31] annotations are used to precisely lift into the 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 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 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)}
```

where (◇) 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.

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
```

```
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`. 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` by the SMT solver. `p *** QED` casts any expression `p` into a **`Proof`**. `x ==. y` asserts that `x` and `y` are equal. `thm ∴ lemma` proves `thm` using the `lemma`. `x∧.y` combines two proofs `x` and `y` into one by inserting the argument proofs into the logical environment.

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 then logic. The proof combinator `x ==. y` let us equate `x` with `y` into 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 (or tactic), inspired by [16], that Liquid Haskell uses to automatically unfold reflected functions in proof terms. Unlike SMT's heuristics (like E-matching [10, 19]) that make verification unstable [16] PSE is always terminating and is enabled per function. So, when PSE is used to prove one proof/function, even though it could be unpredictable if proof generation will succeed for that specific function, the verification of the rest program is not affected, thus preserving global verification stability. PSE is not formalized in this paper but in § 7 we present how it can be used to simplify proof terms.

PSE can be used to simplify the left identity proof (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. Note that LiquiHaskell, via termination and totality checking, is verifying that all the proof terms are well formed because (1) the inductive hypothesis is only applying to smaller terms and (2) all cases are covered.

We 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 xs 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.

$$
\begin{aligned}
&\texttt{idLeft}_m &&:: \texttt{x:m} \to \{\epsilon \diamond \texttt{x} = \texttt{x}\}\\
&\texttt{idRight}_m &&:: \texttt{x:m} \to \{\texttt{x} \diamond \epsilon = \texttt{x}\}\\
&\texttt{assoc}_m &&:: \texttt{x:m} \to \texttt{y:m} \to \texttt{z:m} \to \{\texttt{x} \diamond (\texttt{y} \diamond \texttt{z}) = (\texttt{x} \diamond \texttt{y}) \diamond \texttt{z}\}
\end{aligned}
$$

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

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

3 VERIFIED PARALLELIZATION OF MONOID MORPHISMS

A monoid morphism is a function between two monoids which preserves the monoidal structure, *i.e.,* preserves identity and distributes. We call a monoid morphism *chunkable* if its domain can be cut into chunks. A chunkable monoid morphism `f` is parallelized by:

- § 3.1 chunking up the input in chunks of size `i` (`chunk i`),
- § 3.2 applying the morphism in parallel to all chunks (`pmap f`), and
- § 3.3 recombining the chunks, in parallel `j` at a time, back to a single value (`pmconcat j`).

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

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

This transformation relies on a single Haskell's library parallelization primitive 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 as $\texttt{take}_m$ 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 the take and drop while the length method is required to give the required 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.

```
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 / [$\texttt{length}_m$ 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 chucked 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 define lists from § **??** to be chunkable monoids according to the aforementioned specifications and we observe the benefits of SMT-based verification when reasoning about linear arithmetic.

```
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 build-in definitions on lists, but, as noted earlier (§ **??**), 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 on the `length` of § **??**. 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 library `parallel`. 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 in the logic. Rather, we assume correctness of the Haskell's library parallelization primitive.

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 $\texttt{chunk}_m$ lets us turn a monoidal value into several pieces. In the other direction, for any monoid (`m`, ϵ, $\diamond$), the monoid concatenation $\texttt{mconcat}_m$ turns a chunked `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 $\texttt{pmconcat}_m$ 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 $\texttt{pmconcat}_m$ i x calls $\texttt{mconcat}_m$ x in the base case, otherwise it (1) chunks the list x in lists of size i, (2) runs in parallel $\texttt{mconcat}_m$ to each chunk, (3) recursively runs itself with the resulting list. Termination of $\texttt{pmconcat}_m$ holds, as the length of chunk i x is smaller than the length of x, when 1 < i.

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

Theorem 3.2. *For each monoid* ($\mathtt{m}$, 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 [1], here we describe the sketch of 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 correct the generalized monoid morphism parallelization.

Theorem 3.3 (Correctness of Parallelization). *Let* ($\mathtt{m}$, ϵ, $\diamond$) *be a monoid and* ($\mathtt{n}$, η, $\boxdot$) *be a chunkable monoid. Then, for every morphism* f :: n → m, *every positive numbers* i *and* j, *and input* x, f x = pmconcat i (pmap f ($\texttt{chunk}_n$ 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* ($\mathtt{m}$, ϵ, $\diamond$) *be a monoid and* ($\mathtt{n}$, η, $\boxdot$) *be a chunkable monoid. Then, for every morphism* f : n → m, *every positive number* i *and input* x, f x = $\texttt{mconcat}_m$ (pmap f ($\texttt{chunk}_n$ 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 3.4) in the Coq proof assistant. In this section we discuss the main differences between the two proofs.

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 (versus Coq) naturally favors intrinsic (versus extrinsic) verification. 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 the intrinsic verification approach, 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 on lists (here `ssreflect`'s `seq`) 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 equipped with a pool of lemmata. For example, to prove the `drop_spec_list` we just apply the library lemma `seq.size_take`, 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. Hint-Based Automation

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

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

Theorem take_spec_list :
  ∀ i x, i ≤ length_list x → length_list (drop_list i x) = i.
Proof.
  move ⇒ i x HSize.
  rewrite seq.size_take.
  destruct (i < size x) eqn:Size; ssromega.
Qed.
```

The crux of the proof lies in the application of the library lemma `size_take`. But, the lemma and the theorem differ when `i` is exactly equal to `size x`, creating the following proof obligation.

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

That is, after branching on whether `i` is less than the `size` of `x`, 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 [1] we resort to `ssromega`, an adaptation of the advanced Pressburger Arithmetic solver `omega` for `ssreflect`. .

We acknowledge that this example is so trivial that Coq's automation support can handle it given a sufficiently large hint database. However, it is interesting enough to illustrate the difference between SMT- and hint-based verification. SMT verification is complete over a limited number of theories and, in Liquid Haskell, the user has no way to expand these theories. On the contrary, in Coq the user has the option to customize the hint database. However, when automation fails (which is not a rare situation), the user has to understand the reason of failure and to manually complete the proof. Worse, the proofs generated by the decision procedure are far from ideal. Quoting the Coq Reference Manual : "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 would introduce an inconsistency in the logic, all reflected Haskell functions and all Gallina 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 [31], while Coq, that does not explicitly distinguish between logic and implementation, does not, by default, support partial computations [8].

The second difference is that Liquid Haskell uses a semantic termination checker, unlike Coq that is using the particularly restrictive syntactic criterion that 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. To extract `chunk`'s result we need to prove that enough fuel was provided. Following Coq's extrinsic verification approach, we encode Liquid Haskell's refinement type specifications in a separate Coq `chunk_spec`$_m$ theorem.

```
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 for `chunk` enforces both the length specifications as encoded in `chunk`'s Liquid Haskell type but also 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 [17]. Other than the fueling technique, Adam Chlipala's CPDT compares three such general techniques to bypass Coq's syntactic termination restriction, namely well-founded recursion, domain-theory-inspired non-termination monads, and co-inductive non-termination monads.

4.5 General Purpose vs Verification Specific Languages

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 can extract these constants to their corresponding Haskell counterparts to obtain the same behavior as in Liquid Haskell implementation.

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

In § 3 we presented that any monoid morphism whose domain is chunkable can be parallelized. We now make use of that result to parallelize string matching. We start by observing that strings are a chunkable monoid. We then turn string matching for a given target into a monoid morphism from a string to a suitable monoid, `SM target`, defined in § 5.2. Finally, in § 5.4, we parallelize string matching by a simple use of the parallel morphism function of § 3.4.

5.1 Refined Strings are assumed to be Chunkable Monoids

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

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

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

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

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

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

For instance, we define a logical uninterpreted function ⊡ and relate it to the Haskell ⊡ function via an assumed (unchecked) type.

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

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

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

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

Similarly for the chunkable methods, we define the uninterpreted functions `takeStr`, `dropStr` and `lenStr` in the logic, and use them to strengthen the result types of the respective functions. With the above function definitions (in both Haskell and logic) and assumed type specifications, Liquid Haskell will check (or rather assume) that the specifications of chunkable monoid, as defined in the Definitions 2.1 and 3.1, are satisfied. We conclude with the assumption (rather that theorem) that `RString` is a chunkable monoid.

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

5.2 String Matching Monoid

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

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

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

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

We use the string type literal [1] to parameterize the monoid over the target being matched. This encoding allows the type checker to statically ensure that only searches for the same target can be merged together. The input field is a refined string, and the indices field is a list of good indices. For simplicity we present lists as Haskell's built-in lists, but our implementation uses the reflected list type, `L`, defined in § 2.

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

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

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

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

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

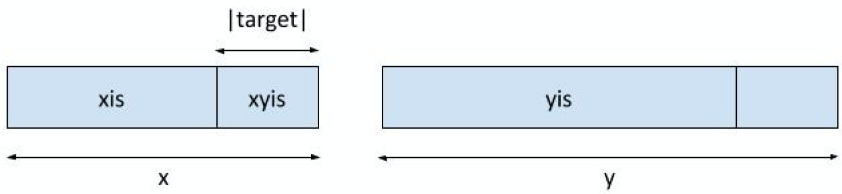

Fig. 2. Mappend indices of String Matcher

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

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

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

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

```
(◇)::∀ (t::Symbol). KnownSymbol t ⇒ SM t → SM t → SM t
(SM x xis) ◇ (SM y yis)
  = SM (x ⊡ y) (xis' ++ xyis ++ yis')
  where
    tg   = fromString (symbolVal (Proxy :: Proxy t))
    xis' = map (castGoodIndexLeft tg x y) xis
    xyis = makeNewIndices x y tg
    yis' = map (shiftStringRight tg x y) yis
```

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

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

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

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

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

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

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

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

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

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

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

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

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

Generally, `makeIndices s tg lo hi` returns the good indices of the input string `s` for target `tg` in the range from `lo` to `hi`.

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

It is important to note that makeNewIndices does not scan all the input, instead only searching at most lenStr tg positions for new good indices. Thus, the time complexity to create the new indices is linear on the size of the target but independent of the size of the input.

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

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

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

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

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

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

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

Theorem 5.2 (SM is a Monoid). *(SM t, ϵ, $\diamond$) is a monoid.*

Proof. According to the Monoid Definition 2.1, we prove that string matching is a monoid, by providing safe implementations for the monoid law functions. First, we prove *left identity* using PSE.

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

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

- Identity of shifting by an empty string.

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

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

- No new indices are created.

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

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

Next, we prove *right identity*.

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

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

- Identity of casting is proven

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

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

- No new indices are created.

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

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

- Finally we prove *associativity*. The PSE strategy failed to automatically prove associativity due to the complexity of the proof. For space, we only provide a proof sketch. The whole proof is available online [1]. Our goal is to show equality of the left and right associative string matchers.

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

To prove equality of the two string matchers we show that the input and indices fields are respectively equal. Equality of the input fields follows by associativity of RStrings. Equality of the index list proceeds in three steps.

(1) Using list associativity and distribution of index shifting, we group the indices in the five lists shown in Figure 3: the indices of the input `x`, the new indices from mappending `x` to `y`, the indices of the input `y`, the new indices from mappending `x` to `y`, and the indices of the input `z`.

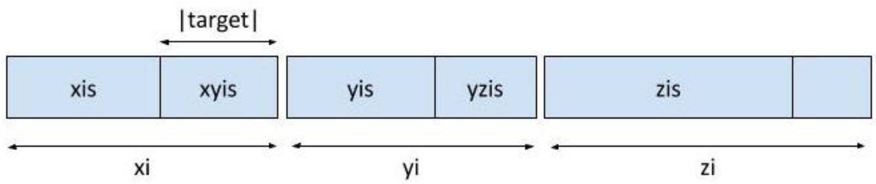

Fig. 3. Associativity of String Matching

(2) The representation of each group depends on the order of appending. For example, if `zis1` (resp. `zis2`) is the group `zis` when right (resp. left) mappend happened first, then we have

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

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

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

(3) After proving equivalence of representations, we again use list associativity and distribution of casts to wrap the index groups back in string matchers.

We now sketch the three proof steps, while the whole proof is available online [1].

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

```
  where
    i     = xi ⊡ (yi ⊡ zi)

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

□

5.3 String Matching Monoid Morphism

Next, we define the function `toSM :: RString → SM target` which does the actual string matching computation for a set target [2]

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

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

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 (`toSM` is a Morphism). `toSM :: RString → SM t` *is a morphism between the monoids (*`RString`*, η, ⊡) and (*`SM t`*, ϵ, $\diamond$).*

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

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

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

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

```
distributestoSM x y
  =   (toSM x :: SM target) ◇ (toSM y :: SM target)
  ==. (SM x is1) ◇ (SM y is2)
  ==. SM i (xis ++ xyis ++ yis)
  ==. SM i (makeIndices i tg 0 hi1 ++ yis)
      ∴ (mapCastId tg x y is1 ∧ mergeNewIndices tg x y)
  ==. SM i (makeIndices i tg 0 hi1 ++ makeIndices i tg (hi1+1) hi)
      ∴ shiftIndicesRight 0 hi2 x y tg
```

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

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

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

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

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

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

□

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 `j` chunks. Then, `pmap` applies `toSM` at each chunk in parallel. Finally, `pmconat` concatenates the mapped chunks in parallel using ◇, the monoidal operation for `SM target`.

Correctness. We prove correctness of `toSMPar` directly from Theorem 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, $\boxdot$) (by Assumption 5.1) and the monoid (`SM t`, ϵ, $\diamond$) (by Theorem 5.2).

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

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

6 STRING MATCHING IN COQ

In the previous section we saw in detail a proof of correctness for the parallelization of a string matching algorithm in Liquid Haskell. Just like in the monoid case, we replicated our proof in the Coq proof assistant. In this section we present the highlights of this effort, identifying more complementary strengths and weaknesses.

6.1 Existing Library Support

In the Liquid Haskell proof, we used a wrapper around `ByteStrings` to represent strings for efficiency; we also assumed the correctness of the `ByteString` operations instead of verifying them. In Coq, we used the built-in implementation of `Strings`. This allowed us to rely on the existing library theorems instead of assuming properties. We still admitted some theorems (e.g. the interoperation between `take` and `drop`), where direct counterparts where not available.

6.2 Proof inlining

In Liquid Haskell induction is encoded via recursive function calls. This highly restricts the structure of the Liquid Haskell proofs as each property proved by induction should be independenly encoded as a recursive function/lemma. Thus, the user is forced to separately specify and proof each inductive lemma required for the proof. On the contrary, Coq does not impose any such restriction, allowing the user to prove lemmata inlined in the proof.

This convenience comes with the disadvantage that many times the proof repeatedly proves the same lemmata. For example, the `catIndices` property required to prove both distribution and associativity was expressed as a separate lemma in the Liquid Haskell proof, due to the lack of a different way ti express it, while was by demand, inlined proven twice in the Coq proof.

6.3 Executable vs Inductive Specifications

In Liquid Haskell, `GoodIndex input tg` is a refinement type capturing the indices of the string `input` where the target string `tg` appears. Recall that `GoodIndex` is defined using the executable boolean predicate `isGoodIndex`.

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

In fact, all refinements are executable boolean predicates written in Haskell. On the other hand, Coq users usually define *inductive* specifications which allow easier reasoning:

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

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

```
Definition isGoodIndexDec input tg i:
  {isGoodIndex input tg i} + {~ (isGoodIndex input tg i)}.
Proof.
  destruct (string_dec (substring i (length tg) input) tg) eqn:Eq;
  destruct (i + length tg ≤ length input) eqn:Ineq;
  auto; right ⇒ Contra; inversion Contra; eauto.
Qed.
```

Instead of returning a simple boolean, the decidability procedure returns a 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 that can be used in proofs. 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.

6.4 Non Structural Recursion

Just like in § 5, string matching in Liquid Haskell requires an auxilliary recursive predicate `makeIndices` that uses non structural induction: a call to `makeIndices s tg lo hi` has a recursive call to `makeIndices s tg (lo+1) hi`, whose termination measure is `(hi - lo)`. In § 4 we dealt with non-structural recursion using fuel. In this case, it is easy to implement an equivalent predicate in Coq so that it *is* structurally recursive by calculating in advance how many steps (`cnt`) we need to take:

```
Fixpoint makeIndicesAux (s tg : string) (lo cnt : nat) : list nat :=
  match cnt with
    | O ⇒ nil
    | S cnt' ⇒
      let rest := makeIndicesAux s tg lo.+1 cnt' in
      if isGoodIndexDec s tg lo then lo::rest else rest
  end.

Definition makeIndices (s tg : string) (lo hi : nat) : list nat :=
  makeIndicesAux s tg lo (hi - lo).
```

6.5 Intrinsic vs Extrinsic Verification

There is an even more fundamental difference between the two `makeIndices` implementations: the type of the one in Coq does not mention any correctness properties unlike its Liquid Haskell counterpart! In order to prove things in Liquid Haskell we had to use an intrinsic verification approach: by using refinement types the assumptions to our theorems are tied to the input types while the correctness results are tied to the output types. Thus, incorrect programs are impossible to even express. However, this approach has a couple of unfortunate drawbacks.

First, the computational content of the definitions must sometimes be cluttered to make the types work out. For example, when implementing the monoid operation ◇ for `SM`, we used `castGoodIndexLeft` to transform the refinement of the input (stating that every index in `xis` is a good index for `x`) to the refinement of the output (every index in `xis′` is a good index for `x++y`). However, computationally, this is identity: `xis′` is equal to `xis`. Second, *everything* we want to prove about a piece of code has to be proved at the same time. For instance, if we wanted to impose an additional constraint for good indices, we would have to revise all of our definitions, potentially adding more clutter.

In contrast, Coq users usually follow an extrinsic verification approach; operations over simply-typed expressions like `makeIndices` are proved correct after the fact:

```
Lemma makeIndicesAux_correct :
  ∀ cnt s tg lo,
    List.Forall (isGoodIndex s tg) (makeIndicesAux s tg lo cnt).
Proof.
  move ⇒ cnt; induction cnt ⇒ s tg lo //=;
  destruct (isGoodIndexDec s tg lo); simpl; auto.
Qed.
```

The use of an extrinsic approach in our proof development greatly simplifies the process. Specifically, the `SM` datatype is just a pair, while its validity is captured by a different inductive type.

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

  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 the monoid operator `◇_sm`,

```
Definition ◇_sm tg (sm1 sm2 : SM tg) :=
  let '(Sm x xis) := sm1 in
  let '(Sm y yis) := sm2 in
  let xis' := xis in
  let xyis := makeNewIndices x y tg in
  let yis' := map (shiftStringRight tg x y) yis in
  Sm tg (x  y) (List.app xis' (List.app xyis yis')).
```

where `xis′` is *by definition* equal to `xis`. At the same time, the extrinsic approach clarifies exactly when the correctness assumptions for the index lists are necessary. For example, in the associativity proof of `◇_sm` we only require the middle string to be valid:

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

On the other hand, the validity of ◇_sm requires the validity of both inputs as preconditions:

```
Lemma sm_valid tg xs1 l1 xs2 l2 xs' l' :
  List.Forall (isGoodIndex xs1 tg) l1 →
  List.Forall (isGoodIndex xs2 tg) l2 →
  ◇_sm tg (Sm tg xs1 l1) (Sm tg xs2 l2) = Sm tg xs' l' →
  List.Forall (isGoodIndex xs' tg) l'.
```

6.6 Dependent Pattern Matching

Unfortunately, we can not use extrinsic verification all the way through. A pair of strings and lists of natural numbers (like SM, the result of ◇ in Coq) is *not* a monoid by itself; only *valid* SMs form a monoid. To be able to use the monoid proofs of earlier sections we define a more restricted type sm, that carries along a proof of "goodness".

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

Implementing the monoidal operation for this version of sm exemplifies the inconvenience of intrinsic approaches. Ideally, we would like to reuse the definition and properties of ◇_sm directly, writing something like the following piece of code, where we would use sm_valid to construct the proof to fill <proof>.

```
  ◇ sm1 sm2 :=
    match sm1, sm2 with
    | mk_sm xs1 l1 H1, mk_sm xs2 l2 H2 ⇒
       match ◇_sm tg (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 (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*: 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.

```
  ◇ sm1 sm2 :=
    match sm1, sm2 with
    | mk_sm xs1 l1 H1, mk_sm xs2 l2 H2 ⇒
        let s := ◇_sm tg (Sm tg xs1 l1) (Sm tg xs2 l2) in
        let App := erefl s in
        (match s as s0 return (◇_sm tg (Sm tg xs1 l1) (Sm tg xs2 l2) = s0 → sm tg)
        with
        | Sm xs' l' ⇒ fun _ ⇒ mk_sm tg xs' l' _
        end) App
    end
```

6.7 Proof Irrelevance

But we have an even bigger problem. Unlike Liquid Haskell where the intrinsic specifications only live at the logic level, in Coq they are part of the terms. Which means that the associativity proof of ◇ requires that all of

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 of the proofs. We report verification time and LoC (Lines of Code) required to prove the general **parallelization** equivalence of monoid morphisms and its application to the **string matcher**. We split the 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. **Time** is verification time in seconds.

the string, integer list and validity proof components of the resulting sms are *syntactically* equal. However, such proofs are *not* necessarily equal!

To that end, we use *Proof Irrelevance*, an admittable axiom, consistent with Coq's logic (but not necessarily other axioms like *univalence*), which states that any two proofs of the same property are equal.

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

For sanity check, we only use it once to prove an equality lemma for sms, that only require the string and integer list components to be equal.

```
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'.
```

Using the proof irrelevant equality we were able to prove the monoid instance of sm (and similar tricks were necessary for monoid$_m$orphism). We conjecture that this proof is impossible without using such an axiom.

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 none of the proofs were optimized neither 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 [5, 22, 28] 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 lacks such features minimal support for such features: dealing with essential non-structural recursion patterns is incovenient 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 and 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. Moreover, Coq's logic can reason about proofs themselves. This led to us using proof irrelevance (§ 6.7) to explicitly ignore such reasoning, which would be entirely impossible in Liquid Haskell.

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. Whenever a definition falls out of this restrictive recursion pattern, one must resort to more advanced techniques, each with their own drawbacks (§ 4.4 and 6.4).

8 RELATED WORK

SMT-Based Verification. SMT solvers have been extensively used to automate reasoning on verification languages like Dafny [15], Fstar [28] and Why3 [11]. These languages are designed for verification, thus have limited support for commonly used language features like parallelism and optimized libraries that we use in our verified implementation. Refinement Types [7, 13, 24] on the other hand, target existing general purpose languages, such as ML [3, 22, 33], C [6, 23], Haskell [31], Racket [14] and Scala [26]. However, before Refinement Reflection [32] was introduced, they only allowed "shallow" program specifications, that is, properties that only talk about behaviors of program functions but not functions themselves.

Dependent Types. Unlike Refinement Types, dependent type systems, like Coq [4], Adga [20] and Isabelle/HOL [21] allow for "deep" specifications which talk about program functions, such as the program equivalence reasoning we presented. Compared to (Liquid) Haskell, these systems allow for tactics and heuristics that automate proof term generation but lack SMT automations and general-purpose language features, like non-termination, exceptions and IO. Zombie [5, 27] and Fstar [28] allow dependent types to coexist with divergent and effectful programs, but still lack the optimized libraries, like `ByteSting`, which come with a general purpose language with long history, like Haskell.

Parallel Code Verification. Dependent type theorem provers have been used before to verify parallel code. BSP-Why [12] is an extension to Why2 that is using both Coq and SMTs to discharge user specified verification conditions. Daum [9] used Isabelle to formalize the semantics of a type-safe subset of C, by extending Schirmer's [25] formalization of sequential imperative languages. Finally, Swierstra [29] formalized mutable arrays in Agda and used them to reason about distributed maps and sums.

One work closely related to ours is SyDPaCC [18], a Coq library that automatically parallelizes list homomorphisms by extracting parallel Ocaml versions of user provided Coq functions. Unlike SyDPaCC, we are not automatically generating the parallel function version, because of engineering limitations (§ 7). Once these are addressed, code extraction can be naturally implemented by turning Theorem 3.3 into a Haskell type class with a default parallelization method. SyDPaCC used maximum prefix sum as a case study, whose morphism verification is much simpler than our string matching case study. Finally, our implementation consists of 2K lines of Liquid Haskell, which we consider verbose and aim to use tactics to simplify. On the contrary, the SyDPaCC implementation requires three different languages: 2K lines of Coq with tactics, 600 lines of Ocaml and 120 lines of C, and is considered "very concise".

9 CONCLUSION

We described how Liquid Haskell equipped with Refinement Reflection can be used as a theorem prover by presenting its first non-trivial application to a realistic Haskell program: parallelization of a string matcher. We ported our 1428 LoC proof to the Coq proof assistant (1136 LoC) and compared the two provers capturing the essential differences of using dependent and refinement types for theorem proving. We conclude that the strong points of Liquid Haskell as a theorem prover is that the proof refers to executable Haskell code that directly uses advanced Haskell's features like optimized libraries, parallel or diverging code and that the proof is SMT-automated over decidable theories (like linear arithmetic). The strong points of Coq is that the proof is checked assuming a minimum trusted code base and proof development is assisted by a pool of library theorems and tactics. Make some reference to the book here.

REFERENCES

[1] Code for verified string indexing. 2017. Provided in Non-anonymous Supplementary Material.
[2] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. 2010.
[3] J. Bengtson, K. Bhargavan, C. Fournet, A.D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*, 2008.

[4] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.
[5] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. In *POPL*, 2014.
[6] Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *ESOP*, 2007.
[7] R. L. Constable and S. F. Smith. Partial objects in constructive type theory. In *LICS*, 1987.
[8] Nils Anders Danielsson. Operational semantics using the partiality monad. In *ICFP*, 2012.
[9] M Daum. Reasoning on Data-Parallel Programs in Isabelle/Hol. In *C/C++ Verification Workshop*, 2007.
[10] L de Moura and N Bjorner. Efficient E-matching for Smt Solvers. In *CADE*, 2007.
[11] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – Where Programs Meet Provers. In *ESOP*, 2013.
[12] J Fortin and F. Gava. BSP-Why: A tool for deductive verification of BSP algorithms with subgroup synchronisation. In *Int J Parallel Prog*, 2015.
[13] T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, 1991.
[14] Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. Occurrence typing modulo theories. In *PLDI*, 2016.
[15] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. LPAR, 2010.
[16] Rustan Leino and Clment Pit-Claudel. Trigger selection strategies to stabilize program verifiers. 2016.
[17] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL fi06*, 2006.
[18] Frédéric Loulergue, Wadoud Bousdira, and Julien Tesson. Calculating Parallel Programs in Coq using List Homomorphisms. In *International Journal of Parallel Programming*, 2016.
[19] Micha lMoskal, Jakub Lopuszański, and Joseph R. Kiniry. E-matching for Fun and Profit. In *Electron. Notes Theor. Comput. Sci.*, 2008.
[20] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers, 2007.
[21] L. C. Paulson. Isabelle fi?! A Generic Theorem prover. *Lecture Notes in Computer Science*, 1994.
[22] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.
[23] P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, 2010.
[24] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in pvs. *IEEE TSE*, 1998.
[25] N Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, TU Munich, 2006.
[26] Georg Stefan Schmid and Viktor Kuncak. SMT-based Checking of Predicate-Qualified Types for Scala. In *Scala*, 2016.
[27] Vilhelm Sjöberg and Stephanie Weirich. Programming up to congruence. *POPL*, 2015.
[28] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *POPL*, 2016.
[29] Wouter Swierstra. More dependent types for distributed arrays. 2010.
[30] N. Vazou, E. L. Seidel, and R. Jhala. Liquidhaskell: Experience with refinement types in the real world. In *Haskell Symposium*, 2014.
[31] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement Types for Haskell. In *ICFP*, 2014.
[32] Niki Vazou and Ranjit Jhala. Refinement Reflection. arXiv:1610.04641, 2016.
[33] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, 1998.