We introduce Refinement Reflection, a new framework for building SMT-based deductive verifiers. The key idea is to reflect the code implementing a user-defined function into the function’s (output) refinement type. As a consequence, at uses of the function, the function definition is instantiated in a precise fashion that permits decidable verification. We show how reflection allows the user to write equational proofs of programs just by writing other programs, e.g., using pattern-matching and recursion to perform case-splitting and induction. Thus, via, the propositions-as-types principle we show that reflection permits the specification of arbitrary functional correctness properties. While equational proofs are easy, writing them out can be exhausting. We introduce a proof-search algorithm called Proof by Logical Evaluation that uses techniques from model checking & abstract interpretation, to completely automate equational reasoning. We have implemented reflection in LIQUID HASKELL and used it to verify that the widely used instances of the Monoid, Applicative, Functor, and Monad typeclasses actually satisfy key algebraic laws required to make the clients safe, and to build the first library that actually verifies assumptions about associativity and ordering that are crucial for safe deterministic parallelism.

1 INTRODUCTION

Deductive verifiers fall roughly into two camps. Satisfiability Modulo Theory (SMT) based verifiers (e.g. Dafny and F*) use fast decision procedures to completely automate the verification of programs that only require reasoning over a fixed set of theories like linear arithmetic, string, set and bitvector operations. These verifiers, however, encode the semantics of user-defined functions with universally-quantified axioms and use incomplete (albeit effective) heuristics to instantiate those axioms. These heuristics make it difficult to characterize the kinds of proofs that can be automated, and hence, explain why a given proof attempt fails [Leino and Pit-Claudel 2016]. At the other end, we have Type-Theory (TT) based theorem provers (e.g. Coq and Agda) that use type-level computation (normalization) to facilitate principled reasoning about terminating user-defined functions, but which require the user to supply lemmas or rewrite hints to discharge proofs over decidable theories.

We introduce Refinement Reflection, a new framework for building SMT-based deductive verifiers, which permits the specification of arbitrary properties and yet enables complete, automated SMT-based reasoning about user-defined functions. In previous work, refinement types [Constable and Smith 1987; Rushby et al. 1998] — which decorate basic types (e.g. Integer) with SMT-decidable predicates (e.g. \( \{ v : \text{Integer} \mid 0 \leq v \land v < 100 \} \)) — were used to retrofit so-called shallow verification, such as array bounds checking, into several languages: 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. 2014], TypeScript [Vekris et al. 2016], and Racket [Kent et al. 2016].
1. Refinement Reflection Our first contribution is the notion of refinement reflection. To reason about user-defined functions, the function’s implementation can be reflected into its (output) refinement-type specification, thus converting the function’s type signature into a precise description of the function’s behavior. This simple idea has a profound consequence: at uses of the function, the standard rule for (dependent) function application yields a precise means of reasoning about the function (§ 4).

2. Complete Specification Our second contribution is a library of combinators that lets programmers compose sophisticated proofs from basic refinements and function definitions. Our proof combinators let programmers use existing language mechanisms like branches (to encode case splits), recursion (to encode induction), and functions (to encode auxiliary lemmas) to write proofs that look very much like their pencil-and-paper analogues (§ 2). Furthermore, since proofs are literally just programs, we use the principle of propositions-as-types [Wadler 2015] (known as Curry-Howard isomorphism [Howard 1980]) to show, via a recipe for encoding proofs with alternating quantifiers, that SMT-based verifiers can express any natural deduction proof, and and provide a pleasant implementation of natural deduction that can be used for pedagogical purposes (§ 3).

3. Complete Verification While equational proofs can be very easy and expressive, writing them out can quickly get exhausting. Our third contribution is Proof by Logical Evaluation (PLE) a new proof-search algorithm that completely automates equational reasoning. The key idea in PLE is to mimic type-level computation within SMT-logics by representing functions in a guarded form [Dijkstra 1975] and repeatedly unfolding function application terms by instantiating them with their definition corresponding to an enabled guard. We formalize a notion of equational proof and show that the above strategy is complete: i.e. it is guaranteed to find an equational proof if one exists. Furthermore, using techniques from the literature on Abstract Interpretation [Cousot and Cousot 1977] and Model Checking [Clarke et al. 1992], we show that the above proof search corresponds to a universal (or must) abstraction of the concrete semantics of the user-defined functions. Thus, as those functions are total we obtain the pleasing guarantee that proof search terminates (§ 6).

We evaluate our approach by implementing refinement reflection and PLE in Liquid Haskell [Vazou et al. 2014], thereby turning Haskell into a theorem prover. Repurposing an existing programming language allows us to take advantage of a mature compiler and an ecosystem of libraries, while keeping proofs and programs in the same language. We demonstrate the benefits of this conversion by proving typeclass laws. Haskell’s typeclass machinery has led to a suite of expressive abstractions and optimizations which, for correctness, crucially require typeclass instances to obey key algebraic laws. We show how reflection and PLE can be used to verify that widely used instances of the Monoid, Applicative, Functor, and Monad typeclasses satisfy the respective laws. Finally, we use reflection to create the first deterministic parallelism library that actually verifies assumptions about associativity and ordering that ensure determinism (§ 7).

Thus, our results demonstrate that Refinement Reflection and Proof by Logical Evaluation identify a new design for deductive verifiers which, by combining the complementary strengths of SMT- and TT- based approaches, enables complete verification of expressive specifications spanning decidable theories and user defined functions.

2 OVERVIEW
We start with an overview of how SMT-based refinement reflection lets us write proofs as plain functions and how PLE automates equational reasoning.

2.1 Refinement Types
First, we recall some preliminaries about specification and verification with refinement types.
Refinement types are the source program’s (here Haskell’s) types refined with logical predicates drawn from an SMT-decidable logic [Constable and Smith 1987; Rushby et al. 1998]. For example, we define Nat as the set of Integer values v that satisfy the predicate \( 0 \leq v \) from the quantifier-free logic of linear arithmetic and uninterpreted functions (QF-UFLIA [Barrett et al. 2010]):

```
type Nat = { v:Integer | 0 \leq v }
```

Specification & Verification Throughout this section, to demonstrate the proof features we add to LIQUID HASKELL, we will use the textbook Fibonacci function which we type as follows.

```
fib :: Nat \rightarrow Nat
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

To ensure termination, the input type’s refinement specifies a pre-condition that the parameter must be Nat. The output type’s refinement specifies a post-condition that the result is also a Nat. Refinement type checking can automatically verify that if fib is invoked with a non-negative Integer, then it terminates and yields a non-negative Integer.

Propositions We can define a data type representing propositions as an alias for unit:

```
type Prop = ()
```

which can be refined with propositions about the code, e.g. that \( 2 + 2 \) equals 4

```
type Plus_2_2 = { v: Prop | 2 + 2 = 4 }
```

For simplicity, in LIQUID HASKELL, we abbreviate the above to `type Plus_2_2 = { 2 + 2 = 4 }`.

Universal & Existential Propositions Using the standard encoding of Howard [1980], known as Curry-Howard isomorphism, refinements encode universally-quantified propositions as dependent function types of the form:

```
type Plus_comm = x:Integer \rightarrow y:Integer \rightarrow \{ x + y = y + x \}
```

As \( x \) and \( y \) refer to arbitrary inputs, any inhabitant of the above type is a proof that Integer addition commutes.

Refinements encode existential quantification via dependent pairs of the form:

```
type Int_up = n:Integer \rightarrow (m::Integer, {n < m})
```

The notation \( (m :: t, t') \) describes dependent pairs where the name \( m \) for the first element can appear inside refinements for the second element. Thus, Int_up states the proposition that for every integer \( n \), there exists one that is larger than \( n \).

While quantifiers cannot appear directly inside the refinements, dependent functions and pairs allow us to specify quantified propositions. One limitation of this encoding is that quantifiers cannot exist inside refinement’s logical connectives (like \( \land \) and \( \lor \)). In this paper, we present how to encode logical connectives using data types, e.g. conjunction as product and disjunction as a union, and show how to specify arbitrary, quantified propositions using refinement types, i.e. have complete specifications, and how to verify those propositions using refinement type checking (§ 3).

Proofs We prove the above propositions by writing Haskell programs, for example

```
plus_2_2 :: Plus_2_2
plus_2_2 = ()

plus_comm :: Plus_comm
plus_comm = \x y \rightarrow ()

int_up :: Int_up
int_up = \n \rightarrow (n+1,())
```

Standard refinement typing reduces the above to the respective verification conditions (VCs)

$$\text{true} \Rightarrow 2 + 2 = 4 \quad \forall x, y. \text{true} \Rightarrow x + y = y + x \quad \forall n. n < n + 1$$

which are easily deemed valid by the SMT solver, allowing us to prove the respective propositions.

**A Note on Bottom:** Readers familiar with Haskell’s semantics may be concerned that “bottom”, which inhabits all types, makes our proofs suspect. Fortunately, as described in Vazou et al. [2014], LIQID HASKELL, by default, checks that user defined functions provably terminate and are total (i.e. non-bottom, no-runtime-exception), which makes our proofs sound.

**Termination Checking** LIQID HASKELL checks that each function is terminating using a termination metric i.e. a natural number that decreases at each recursive call. For instance, generalizing the definition of fib to Integers, leads to a LIQID HASKELL termination error. Assuming the fib is typed as follows

$$\text{fib} :: n: \text{Integer} \rightarrow \text{Integer}$$

LIQID HASKELL generates a type error in the definition of fib since in the recursive calls fib (n-1) and fib (n-2) the arguments n-1 and n-2 cannot be proved to be non-negative and less than n. Both these proof obligations are satisfied when the domain of fib is restricted to natural numbers.

$$\text{fib} :: n: \text{Nat} \rightarrow \text{Nat} / [n]$$

The above type signature is explicitly annotated with the user specified termination metric / [n] declaring that n is the decreasing natural number. LIQID HASKELL heuristically assumes that the termination metric is always the first argument of the function (that can be mapped to natural numbers) thus, the above explicit termination metric can be omitted.

Since not all Haskell functions terminate, the annotation lazy diverge deactivates termination checking for the diverge function. Lazy Haskell terms could be unsoundly used as proof terms, much like Coq’s unsoundness of Admitted. For example, the following proof will get accepted by LIQID HASKELL

$$\text{lazy diverge}$$

$$\text{diverge} :: x: \text{Integer} \rightarrow \{ x = 0 \}$$

$$\text{diverge} x = \text{diverge} x$$

**Totality Checking** LIQID HASKELL further checks that all user-specified functions are totally defined. For instance the bellow definition

$$\text{fib_partial} 0 = 0$$
$$\text{fib_partial} 1 = 1$$

generates a totality type error. The above partial definition gets completed by an error invocation

$$\text{fib_partial} 0 = 0$$
$$\text{fib_partial} 1 = 1$$
$$\text{fib_partial} _ = \text{error} \ "\text{undefined}"$$

To check totality, LIQID HASKELL assumes a false precondition for error

$$\text{error} :: \{ v: \text{String} \mid \text{false} \} \rightarrow \text{a}$$

To typecheck the partial fib_partial LIQID HASKELL needs to prove that the error case is dead-code, which is impossible, thus generates a totality error. As with termination checking, LIQID HASKELL provides an unsound error function unsoundError with no false precondition.
In the rest we assume that proof terms are generated without any unsound lazy or unsafeError calls, thus are sound. Yet, lazy, diverging functions can be soundly verified [Vazou et al. 2014]. Thus, diverging code can soundly co-exist with terminating proof terms.

2.2 Refinement Reflection

Suppose we wish to prove properties about the fib function, e.g. that \(\{\text{fib} 2 = 1\}\). Standard refinement type checking runs into two problems. First, for decidability and soundness, arbitrary user-defined functions cannot belong in the refinement logic, i.e. we cannot refer to fib in a refinement. Second, the only specification that a refinement type checker has about fib is its type \(\text{Nat} \to \text{Nat}\) which is too weak to verify \(\{\text{fib} 2 = 1\}\). To address both problems, we reflect fib into the logic which sets the three steps of refinement reflection in motion.

**Step 1: Definition** The annotation creates an uninterpreted function \(\text{fib} :: \text{Integer} \to \text{Integer}\) in the refinement logic. By uninterpreted, we mean that the logical fib is not connected to the program function fib; in the logic, fib only satisfies the congruence axiom \(\forall n, m. n = m \Rightarrow \text{fib} n = \text{fib} m\). On its own, the uninterpreted function is not terribly useful: we cannot check \(\{\text{fib} 2 = 1\}\) as the SMT solver cannot prove the VC \(\text{true} \Rightarrow \text{fib} 2 = 1\) which requires reasoning about fib’s definition.

**Step 2: Reflection** In the next key step, we reflect the definition of fib into its refinement type by automatically strengthening the user defined type for fib to:

\[
\text{fib} :: \text{n:Nat} \to \{ v:\text{Nat} | v = \text{fib} n \&\& \text{fibP} n \}
\]

where fibP is an alias for a refinement automatically derived from the function’s definition:

\[
\begin{align*}
\text{fibP} n &= n = 0 \Rightarrow \text{fib} n = 0 \\
&\quad \wedge n = 1 \Rightarrow \text{fib} n = 1 \\
&\quad \wedge n >= 1 \Rightarrow \text{fib} n = \text{fib} (n-1) + \text{fib} (n-2)
\end{align*}
\]

**Step 3: Application** With the reflected refinement type, each application of fib in the code automatically unfolds the definition of fib once in the logic. We prove \(\{\text{fib} 2 = 1\}\) by:

\[
\text{pf_fib2} :: \{ \text{fib} 2 = 1 \}
\]

\[
\text{pf_fib2} = \text{let} \{ t0 = \text{fib} 0; t1 = \text{fib} 1; t2 = \text{fib} 2 \} \text{ in } ()
\]

We write in bold red, f, to highlight places where the unfolding of f’s definition is important. Via refinement typing, the above yields the following VC that is discharged by SMT, even though fib is uninterpreted:

\[
((\text{fibP} 0) \wedge (\text{fibP} 1) \wedge (\text{fibP} 2)) \Rightarrow (\text{fib} 2 = 1)
\]

The verification of pf_fib2 relies merely on the fact that纤维 is applied to (i.e. unfolded at) 0, 1 and 2. The SMT solver automatically combines the facts, once they are in the antecedent. The following is also verified:

\[
\text{pf_fib2’} :: \{v:\text{[Nat]} | \text{fib} 2 = 1 \}
\]

\[
\text{pf_fib2’} = [ \text{fib} 0, \text{fib} 1, \text{fib} 2 ]
\]

In the next subsection, we will continue to use explicit, step-by-step proofs as above, but we introduce additional tools for proof composition. Then, in § 2.4 we will eliminate unnecessary details in such proofs, using Proof by Logical Evaluation (PLE) for automation.
2.3 Equational Proofs

We can structure proofs to follow the style of calculational or equational reasoning popularized in classic texts [Bird 1989; Dijkstra 1976] and implemented in AGDA [Mu et al. 2009] and Dafny [Leino and Polikarpova 2016]. To this end, we have developed a library of proof combinators that permits reasoning about equalities and linear arithmetic.

“Equation” Combinators We equip LIQUID HASKELL with a family of equation combinators, ⊙, for logical operators in the theory QF-UFLIA, ⊙ ∈ {=, ≠, ≤, ≥, >}. (In Haskell code, to avoid collisions with existing operators, we further append a colon “:” to these operators, so that “=” becomes “=:” instead.) The refinement type of ⊙ requires that x ⊙ y holds and then ensures that the returned value is equal to x. For example, we define =: as:

\[(=:) :: x:a \rightarrow y:(a \mid x = y) \rightarrow \{ v:a \mid v = x \}\]
\[x =: _ = x\]

and use it to write the following “equational” proof:

\[\text{fib2}_1 :: \{ \text{fib } 2 = 1 \}\]
\[\text{fib2}_1 = \text{fib } 2 =: \text{fib } 1 + \text{fib } 0 =: 1 \quad \text{** QED}\]

where ** QED constructs “proof terms” by “casting” expressions to Prop in a post-fix fashion.

Proof Arguments Often, we need to compose lemmas into larger theorems. For example, to prove \(\text{fib } 3 = 2\) we may wish to reuse \(\text{fib2}_1\) as a lemma. We do so, by defining a variant of \((=:)\), written as \((=?):\), that takes a proof argument:

\[(=?): x:a \rightarrow y:a \rightarrow \{ \text{Pr} \circ | x = y \} \rightarrow \{ v:a \mid v = x \}\]
\[x =? _ _ = x\]

We use the \((=?):\) combinator to prove that \(\text{fib } 3 = 2\).

\[\text{fib3}_2 :: \{ \text{fib } 3 = 2 \}\]
\[\text{fib3}_2 = (\text{fib } 3 =: \text{fib } 2 + \text{fib } 1 =? 2 \ $ \text{fib2}_1) \quad \text{** QED}\]

Here \(\text{fib } 2\) is not important to unfold, because \(\text{fib2}_1\) already provides the same information.

“Because” Combinators Observe that the proof term \(\text{fib3}_2\) needs parentheses, since Haskell’s ($) operator has the lowest (i.e. 0) precedence. To omit parentheses in proof terms, we define a “because” combinator that operates exactly like Haskell’s ($) operator, but has the same precedence as the proof combinators (=:) and (?).:

\[(::) :: (\text{Prop} \rightarrow a) \rightarrow \text{Prop} \rightarrow a\]
\[f :: y = f y\]

We use the because combinator to remove the parentheses from the proof term \(\text{fib3}_2\).

\[\text{fib3}_2 :: \{ \text{fib } 3 = 2 \}\]
\[\text{fib3}_2 = \text{fib } 3 =: \text{fib } 2 + \text{fib } 1 =? 2 :: \text{fib2}_1 \quad \text{** QED}\]

Optional Proof Arguments Finally, we unify both combinators (=:) and (=?): using type classes to define the class method (:) that takes an optional proof argument, and generalize such definitions for each operator in ⊙.

We define the class OptEq to have one method (:) that takes two arguments of type a to be compared for equality and returns a return of type r.
class OptEq a r where
  (=.) :: a → a → r

Instantiating the result type to be the same as the argument type a, (=.) behaves exactly as (=).

instance (a~b) ⇒ OptEq a b where
  (=.) :: x:a → y:a → { x = y } → { v:b | v = x }

When the result type is instantiated to be a function that given a proof term Prop returns the argument type a, (=.) behaves exactly as (=?).

instance (a~b) ⇒ OptEq a (Prop → b) where
  (=.) :: x:a → y:a → { x = y } → { v:a | v = x }

Thus, (=.) takes two arguments to be compared for equality and, optionally, a proof term argument. With this, the proof term fib3_2 is simplified to

\[ \text{fib3}_2 :: \{ \text{fib } 3 = 2 \} \]

\[ \text{fib3}_2 = \text{fib } 3 =. \text{fib } 2 + \text{fib } 1 =. 2 :. \text{fib2}_1 ** \text{QED} \]

**Arithmetic and Ordering** Next, let's see how we can use arithmetic and ordering to prove that fib is (locally) increasing, i.e. for all n, fib n ≤ fib (n + 1).

\[ \text{type Up f} = \text{n: Nat → } \{ \text{f n ≤ f } (\text{n + 1}) \} \]

\[ \text{fibUp} :: \text{Up fib} \]

\[ \text{fibUp } 0 = \text{fib } 0 <. \text{fib } 1 ** \text{QED} \]

\[ \text{fibUp } 1 = \text{fib } 1 <. \text{fib } 2 + \text{fib } 0 =. \text{fib } 2 ** \text{QED} \]

\[ \text{fibUp } n = \text{fib } n <. \text{fib } n + \text{fib } (\text{n-1}) =. \text{fib } (\text{n+1}) ** \text{QED} \]

**Case Splitting** The proof fibUp works by splitting cases on the value of n. In the base cases 0 and 1, we simply assert the relevant inequalities. These are verified as the reflected refinement unfolds the definition of fib at those inputs. The derived VC's are (automatically) proved as the SMT solver concludes 0 < 1 and 1 + 0 ≤ 1 respectively. When n is greater than two, fib n is unfolded to fib (n-1) + fib (n-2), which, as fib (n-2) is non-negative, completes the proof.

**Induction & Higher Order Reasoning** Refinement reflection smoothly accommodates induction and higher-order reasoning. For example, let's prove that every function f that increases locally (i.e. f z ≤ f (z+1) for all z) also increases globally (i.e. f x ≤ f y for all x < y)

\[ \text{type Mono} = f:(\text{Nat → Integer}) → \text{Up f} → \text{x:<_} → \text{y:(x < y) → } \{ \text{f x ≤ f y} \} \]

\[ \text{fMono} :: \text{Mono / [y]} \]

\[ \text{fMono } f \text{ up x y} \]

\[ | x+1 = y = f \text{ x ≤. f (x+1) :. up x ≤. f y} ** \text{QED} \]

\[ | x+1 < y = f \text{ x ≤. f (y-1) :. fMono f up x (y-1) ≤. f y :. up (y-1) ** QED} \]

We prove the theorem by induction on y as specified by the annotation / [y] which states that y is a well-founded termination metric that decreases at each recursive call [Vazou et al. 2014]. If x+1 = y, then we call the up x proof argument. Otherwise, x+1 < y, and we use the induction hypothesis i.e. apply fMono at y-1, after which transitivity of the less-than ordering finishes the proof. We can apply the general fMono theorem to prove that fib increases monotonically:

\[ \text{fibMono} :: \text{n:Nat → m:(n < m) → } \{ \text{fib n ≤ fib m} \} \]

\[ \text{fibMono} = \text{fMono fib fibUp} \]
2.4 Complete Verification: Automating Equational Reasoning

While equational proofs can be very easy, writing them out can quickly get exhausting. Let’s face it: fib3_2 is doing rather a lot of work just to prove that fib 3 equals 2! Happily, the calculational nature of such proofs allows us to develop the following proof search algorithm PLE that is inspired by model checking [Clarke et al. 1992]:

- **Step 1: Guard Normal Form** First, as shown in the definition of fibP above, each reflected function is transformed into a guard normal form \( \land (p_i \Rightarrow f(\bar{x}) = b_i) \) i.e. as a collection of guards \( p_i \) and their corresponding definition \( b_i \).

- **Step 2: Unfolding** Second, given a VC of the form \( \Phi \Rightarrow p \), we iteratively unfold function application terms in \( \Phi \) and \( p \) by instantiating them with the definition corresponding to an enabled guard, where we check enabled-ness by querying the SMT solver. For example, given a VC \( \text{true} \Rightarrow \text{fib} 3 = 2 \), the guard \( 3 \geq 1 \) is trivially enabled, i.e. is true, and hence we strengthen the hypothesis \( \Phi \) with the equality \( \text{fib} 3 = \text{fib} 3 - 1 + \text{fib} 3 - 2 \) corresponding to unfolding the definition of fib at 3.

- **Step 3: Fixpoint** Third, we repeat the above process until either the goal is proved or we have reached a fixpoint, i.e. no further unfolding is enabled. For example, the above fixpoint computation unfolds the definition of fib at 3, 2, 1, and 0 and then stops as no further guards are enabled.

**Automatic Equational Reasoning** In § 6 we formalize a notion of equational proof and show that the proof search procedure PLE enjoys two key properties. First, that it is guaranteed to find an equational proof if one can be constructed from unfoldings of function definitions; instantiations of lemmas and induction hypotheses must still be provided by the user. Second, that under certain conditions readily met in practice, it is guaranteed to terminate. These two properties allow us to use PLE to predictably automate proofs: the programmer needs only supply the relevant induction hypotheses or helper lemma applications. The remaining long chains of calculations are performed automatically via SMT-based PLE. (That is, they must provide case statements and recursive structure, but not chains of =. applications.) To wit, with complete proof search, the above proofs shrink to:

\[
\begin{align*}
\text{fib3}_2 :: \{\text{fib} 3 = 2\} & \quad \text{fibUp} :: \text{Up fib} & \quad \text{fMono} :: \text{Mono} / [y] \\
\text{fib3}_2 = () & \quad \text{fibUp} \; 0 = () & \quad \text{fMono} \; \text{f up x y} \\
& \quad \text{fibUp} \; 1 = () & \quad | \; x+1 \leq y \Rightarrow \text{up x} \\
& \quad \text{fibUp} \; n = () & \quad | \; x+1 < y \Rightarrow \text{up (y-1)} \& \& \; \text{fMono up x (y-1)}
\end{align*}
\]

where the combinator \( p \; \& \& \; q = () \) inserts the propositions \( p \) and \( q \) to the VC hypothesis.

**PLE vs. Axiomatization** Existing SMT based verifiers like Dafny [Leino 2010] and F* [Swamy et al. 2016] use the classical axiomatic approach to verify assertions over user-defined functions like fib. In these systems, the function is encoded in the logic as a universally quantified formula (or axiom): \( \forall n. \; \text{fibP} n \) after which the SMT solver may instantiate the above axiom at 3, 2, 1 and 0 in order to automatically prove \( \{\text{fib} 3 = 2\} \).

The automation offered by axioms is a bit of a devil’s bargain, as axioms render VC checking undecidable, and in practice automatic axiom instantiation can easily lead to infinite “matching loops”. For example, the existence of a term fib n in a VC can trigger the above axiom, which may then produce the terms fib \( (n - 1) \) and fib \( (n - 2) \), which may then recursively give rise to further instantiations ad infinitum. To prevent matching loops an expert must carefully craft “triggers” or, alternatively provide a “fuel” parameter [Amin et al. 2014] that bounds the depth of instantiation. Both these approaches ensure termination, but can cause the axiom to not be instantiated at the
app_assoc :: AppendAssoc
app_assoc [] ys zs
  = ([] ++ ys) ++ zs
  =. ys ++ zs
  =. [] ++ (ys ++ zs)  ** QED
app_assoc (x:xs) ys zs
  = ((x : xs) ++ ys) ++ zs
  =. (x : (xs ++ ys)) ++ zs
  =. x :((xs ++ ys) ++ zs)
  `.` app_assoc xs ys zs
  =. x : (xs ++ (ys ++ zs))
  =. (x : xs) ++ (ys ++ zs)  ** QED

app_assoc :: AppendAssoc
app_assoc [] ys zs = ()
app_assoc (x:xs) ys zs = app_assoc xs ys zs

app_right_id :: AppendNilId
app_right_id [] = ()
app_right_id (x:xs) = app_right_id xs

map_fusion :: MapFusion
map_fusion f g [] = ()
map_fusion f g (x:xs) = map_fusion f g xs

Fig. 1. (L) Equational proof of append associativity, (R) PLE proof, also of append-id and map-fusion.

right places, thereby rendering the VC checking incomplete. The incompleteness is illustrated by the following example from the Dafny benchmark suite [Leino 2016]

\[
\begin{align*}
pos \ n | \ n < 0 &= 0 \\
| \text{otherwise} &= 1 + \pos(n-1) \\
test : & y :\{y > 5\} \rightarrow \{\pos n = 3 + \pos(n-3)\} \\
| \_ &= ()
\end{align*}
\]

Dafny (and F’’s) fuel-based approach fails to check the above, when the fuel value is less than 3. One could simply raise-the-fuel-and-try-again but at what point does the user know when to stop? In contrast, PLE (1) does not require any fuel parameter, (2) is able to automatically perform the required unfolding to verify this example, and (3) is guaranteed to terminate.

2.5 Case Study: Laws for Lists

Reflection and PLE are not limited to integers. We end the overview by showing how they verify textbook properties of lists equipped with append (++) and map functions:

\[
\begin{align*}
\reflect (++) :: [a] \rightarrow [a] \rightarrow [a] \\
\reflect \map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
[] + y s = y s \\
\map f [] = [] \\
(x : x s) + y s = x : (x s ++ y s) \\
\map f (x : x s) = f x : \map f x s
\end{align*}
\]

In § 5.1 we will describe how the reflection mechanism illustrated via \textit{fibP} is extended to account for ADTs using SMT-decidable selection and projection operations, which reflect the definition of ++ into the refinement as: if isNil xs then ys else sel1 xs : (sel2 xs ++ ys). We require an explicit use of \textit{reflect} as not all Haskell functions can be reflected into logic, either because it is unsound to do so (e.g. due to divergence), or because of limitations of our current implementation.

\textbf{Laws} We can specify various laws about lists with refinement types. For example, the below laws state that (1) appending to the right is an \textit{identity} operation, (2) appending is an \textit{associative} operation, and (3) map \textit{distributes} over function composition:

\[
\begin{align*}
\text{type AppendNilId} &= \text{x} s :_\_ \rightarrow \{\text{x} s ++ [] = \text{x} s\} \\
\text{type AppendAssoc} &= \text{x} s :_\_ \rightarrow \text{ys :}_\_ \rightarrow \text{zs :}_\_ \rightarrow \{\text{x} s ++ (\text{ys} ++ \text{zs}) = (\text{x} s ++ \text{ys}) ++ \text{zs}\} \\
\text{type MapFusion} &= f :_\_ \rightarrow g :_\_ \rightarrow \text{x} s :_\_ \rightarrow \{\map f (f . g) \text{x} s = \map f (\map g \text{x} s)\}
\end{align*}
\]

\textbf{Proofs} On the right in Figure 1 we show the proofs of these laws using PLE, which should be compared to the classical equational proof e.g. [Wadler 1987] shown on the left. With PLE, the
user need only provide the high-level structure — the case splits and invocations of the induction hypotheses — after which PLE automatically completes the rest of the equational proof. Thus using SMT-based PLE, app_assoc shrinks down to its essence: an induction over the list xs. The difference is even more stark with map_fusion whose full equational proof is omitted, as it is twice as long.

**PLE vs. Normalization** The proofs in Figure 1 may remind readers familiar with Type-Theory based proof assistants (e.g. Coq or Agda) of the notions of type-level normalization and rewriting that permit similar proofs in those systems. While our approach of PLE is inspired by the idea of type level computation, it differs from it in two significant ways. First, from a theoretical point of view, SMT logics are not equipped with any notion of computation, normalization, canonicity or rewriting. Instead, our PLE algorithm shows how to emulate those ideas by asserting equalities corresponding to function definitions (Theorem 6.10). Second, from a practical perspective, the combination of (decidable) SMT-based theory reasoning and PLE’s proof search can greatly simplify verification. For example, consider the swap function from a Coq textbook [Appel 2016]:

```coq
swap :: [Integer] → [Integer]
swap (x1:x2:xs) = if x1 > x2 then x2:x1:x2 else x1:x2:xs
swap xs = xs
```

In Figure 2 we show four proofs that swap is idempotent: Appel’s proof using Coq (simplified by the use of a hint database and the arithmetic tactic omega), its variant in Agda (for any Decidable Partial Order), the PLE proof, and a proof using the Dafny verifier. It is readily apparent that PLE’s proof search working hand-in-glove with SMT-based theory reasoning makes proving the result relatively trivial. Of course, proof assistants like Agda, Coq, and Isabelle emit easily checkable certificates and have decades-worth of tactics, libraries and proof scripts that enable large scale proof engineering. On the other hand, Dafny’s fuel-based axiom instantiation automatically unfolds the definition of swap twice, thereby completing the proof without any user input. Note that these heuristics are orthogonal to PLE and can be combined with it, if the user wishes to trade off predictability for even more automation.

**Summary** We saw an overview of an SMT-automated refinement type checker that achieves SMT-decidable checking by restricting verification conditions to be quantifier-free and hence, decidable. In existing SMT-based verifiers (e.g. Dafny) there are two main reasons to introduce quantifiers, namely (1) to express quantified specifications, and (2) to encode the semantics of user-defined functions. Next, we use propositions-as-types to encode quantified specifications and in § 4 we show how to encode the semantics of user-defined functions via refinement reflection.
Logical Formula | Refinement Type
--- | ---
Native Terms | \( e \) | \( \{e\} \)
Implication | \( \phi_1 \Rightarrow \phi_2 \) | \( \phi_1 \rightarrow \phi_2 \)
Negation | \( \neg \phi \) | \( \phi \rightarrow \{\text{False}\} \)
Conjunction | \( \phi_1 \land \phi_2 \) | \( (\phi_1, \phi_2) \)
Disjunction | \( \phi_1 \lor \phi_2 \) | Either \( \phi_1 \phi_2 \)
Forall | \( \forall x.\phi \) | \( x : \tau \rightarrow \phi \)
Exists | \( \exists x.\phi \) | \( (x :: \tau, \phi) \)

Fig. 3. Mapping from logical predicates to quantifier-free refinement types. \( \{e\} \) abbreviates \( \{v : \text{Prop} \mid e\} \). Function binders are not relevant for negation and implication, and hence, elided.

3  EMBEDDING NATURAL DEDUCTION WITH REFINEMENT TYPES

In this section we show how user-provided quantified specifications can be naturally encoded using \( \lambda \)-abstractions and dependent pairs to encode universal and existential quantification respectively. Proof terms can be generated using the standard natural deduction derivation rules, following Propositions as Types [Wadler 2015] (also known as the Curry-Howard isomorphism [Howard 1980]). What is new is that we exploit this encoding to show for the first time that a refinement type system can represent any proof in Gentzen’s natural deduction [Gentzen 1935] while still taking advantage of SMT decision procedures to automate the quantifier-free portion of natural deduction proofs. For simplicity, in this section we assume all terms are total; we formalize and relax this requirement in the sequel.

3.1 Propositions: Refinement Types

Figure 3 maps logical predicates to types constructed over quantifier-free refinements.

**Native terms** Native terms consist of all of the (quantifier-free) expressions of the refinement languages. In § 4 we formalize refinement typing in a core calculus \( \lambda R \) where refinements include (quantifier-free) terminating expressions.

**Boolean connectives** Implication \( \phi_1 \Rightarrow \phi_2 \) is encoded as a function from the proof of \( \phi_1 \) to the proof of \( \phi_2 \). Negation is encoded as an implication where the consequent is \( \text{False} \). Conjunction \( \phi_1 \land \phi_2 \) is encoded as the pair \( (\phi_1, \phi_2) \) that contains the proofs of both conjuncts and disjunction \( \phi_1 \lor \phi_2 \) is encoded as the sum type Either that contains the proofs of one of the disjuncts, i.e. where

\[
\text{data} \quad \text{Either} \quad a \ b = \text{Left} \quad a \mid \text{Right} \quad b.
\]

**Quantifiers** Universal quantification \( \forall x.\phi \) is encoded as lambda abstraction \( x : \tau \rightarrow \phi \) and eliminated by function application. Existential quantification \( \exists x.\phi \) is encoded as a dependent pair \( (x :: \tau, \phi) \) that contains the term \( x \) and a proof of a formula that depends on \( x \). Even though refinement type systems do not traditionally come with explicit syntax for dependent pairs, one can encode dependent pairs in refinements using abstract refinement types [Vazou et al. 2013] which do not add extra complexity to the system. Consequently, we add the syntax for dependent pairs in Figure 3 as syntactic sugar for abstract refinements.

3.2 Proofs: Natural Deduction

We overload \( \phi \) to be both a proposition and a refinement type. We connect these two meanings of \( \phi \) by using the Propositions as Types [Wadler 2015], to prove that if there exists an expression (or proof term) with refinement type \( \phi \), then the proposition \( \phi \) is valid.
We construct proofs terms using Gentzen’s natural deduction system [Gentzen 1935], whose rules map directly to refinement type derivations. The rules for natural deduction arise from the propositions-as-types reading of the standard refinement type checking rule (to be defined in § 4) \( \Gamma \vdash e : \phi \) as “\( \phi \) is provable under the assumptions of \( \Gamma \)”. We write \( \Gamma \vdash_{ND} \phi \) for Gentzen’s natural deduction judgement “under assumption \( \Gamma \), proposition \( \phi \) holds”. Then, each of Gentzen’s logical rules can be recovered from the rules in Figure 5 by rewriting each judgement \( \Gamma \vdash e : \phi \) of \( \lambda^R \) as \( \Gamma \vdash_{ND} \phi \). For example, conjunction and universal elimination can be derived as:

\[
\begin{align*}
\Gamma \vdash_{ND} \phi_1 & \lor \phi_2 \\
\Gamma, \phi_1 \vdash_{ND} \phi & \quad \Gamma, \phi_2 \vdash_{ND} \phi \\
\Gamma \vdash_{ND} \phi & \\
\end{align*}
\]

\( \lor \)-E

\[
\begin{align*}
\Gamma \vdash_{ND} \phi & \\
\Gamma \vdash_{ND} \forall x. \phi & \quad \Gamma \vdash_{ND} e_x \text{ term} \\
\Gamma \vdash_{ND} \phi[x/e_x] & \\
\end{align*}
\]

\( \forall \)-E

**Programs as Proofs** As Figure 5 directly maps natural deduction rules to derivations that are accepted by refinement typing, we conclude that if there exists a natural deduction derivation for a proposition \( \phi \), then there exists an expression that has the refinement type \( \phi \).

**Theorem 3.1.** If \( \Gamma \vdash_{ND} \phi \), then we can construct an \( e \) such that \( \Gamma \vdash e : \phi \).

Note that our embedding is not an isomorphism, since the converse of Theorem 3.1 does not hold. As a counterexample, the law of the excluded middle (i.e. \( p : \{\text{True}\} \vdash \bot : p \lor \neg p \)) is evident in our system, but cannot be proved using natural deduction (i.e. \( \{\text{True}\} \lor_{ND} p \lor \neg p \)). The reason for that is that our system is using the classical logic of the SMTs, that includes the law of the excluded middle. On the contrary, in intuitionistic systems that also encode natural deduction (e.g. Coq, IDRS, NuPRL) the law of the excluded middles should be axiomatized.

### 3.3 Examples

Next, we illustrate our encoding with examples of proofs for quantified propositions ranging from textbook logical tautologies, properties of datatypes like lists, and induction on natural numbers.

**Natural Deduction as Type Derivation** We illustrate the mapping from natural deduction rules to typing rules in Figure 4 which uses typing judgments to express Gentzen’s proof of the proposition \( \phi \equiv (\exists x. \forall y. (p \ x \ y)) \Rightarrow (\forall y. \exists x. (p \ x \ y)) \)

Read bottom-up, the derivation provides a proof of \( \phi \). Read top-down, it constructs a proof of the formula as the term \( \lambda e. \text{case } e \text{ of } \{(x, e_x) \to (x, e_x y)\} \). This proof term corresponds directly to the following Haskell expression that typechecks with type \( \phi \).

```haskell
exAll :: p : (a -> a -> Bool) -> (x : a, y : a -> (p x y)) -> y : a -> (x : a, (p x y))
exAll e = \e y -> case e of \{(x, ex) -> (x, ex y)}
```

**SMT-aided proofs** The great benefit of using refinement types to encode natural deduction is that the quantifier-free portions of the proof can be automated via SMTs. For every quantifier-free
proposition $\phi$, you can convert between $\{\phi\}$, where $\phi$ is treated as an SMT-proposition and $\phi$, where $\phi$ is treated as a type; and this conversion goes both ways. For example, let $\phi \equiv p \land (q || r)$ Then flatten converts from $\phi$ to $\{\phi\}$ and expand the other way, while this conversion is SMT-aided.

```
flatten :: p:_ → q:_ → r:_ → ((p), Either {q} {r}) → {p && (q || r))
flatten (pf, Left qf) = pf &&& qf
flatten (pf, Right rf) = pf &&& rf
```

Distributing Quantifiers Next, we construct the proof terms needed to prove two logical properties: that existentials distribute over disjunctions and foralls over conjunctions, i.e.

$\phi_\exists \equiv (\exists x. p x \lor q x) \Rightarrow ((\exists x. p x) \lor (\exists x. q x))$ (1)

$\phi_\forall \equiv (\forall x. p x \land q x) \Rightarrow ((\forall x. p x) \land (\forall x. q x))$ (2)

The specification of these properties requires nesting quantifiers inside connectives and vice versa. The proof of $\phi_\exists$ (1) proceeds by existential case splitting and introduction:

```
exDistOr :: p:_ → q:_ → (x::a, Either {p x} {q x}) → Either (x::a, {p x}) (x::a, {q x})
exDistOr _ _ (x, Left px) = Left (x, px)
exDistOr _ _ (x, Right qx) = Right (x, qx)
```

Dually, we prove $\phi_\forall$ (2) via a $\lambda$-abstraction and case splitting inside the conjunction pair:

```
allDistAnd :: p:_ → q:_ → (x:a→{(p x), (q x)}) → Either (x:a→{p x}), (x:a→{q x}))
allDistAnd _ _ andx = ( (\x → case andx x of (px, _) → px) , (\x → case andx x of (_, qx) → qx) )
```

Properties of User Defined Datatypes As $\phi$ can describe properties of data types like lists, we can prove properties of such types, e.g. that for every list $xs$, if there exists a list $ys$ such that $xs == ys ++ ys$, then $xs$ has even length.

$\phi \equiv \forall xs.(∃ys. xs = ys ++ ys) \Rightarrow (\exists n.\text{len} xs = n + n))$

The proof (evenLen) proceeds by existential elimination and introduction, and uses the \text{lenAppend} lemma, which uses induction on the input list and \text{PLE} to automate equational reasoning.

```
evenLen :: xs:[a]→(ys:[a],{xs = ys ++ ys})→(n:Int,\text{len} xs = n+n)
evenLen xs (ys,pf) = (\text{len} ys, \text{lenAppend} ys ys &&& pf)

\text{lenAppend} :: xs:_ → ys:_ → {\text{len} (xs ++ ys) = \text{len} xs + \text{len} ys}
\text{lenAppend} [] _ = ()
\text{lenAppend} (x:xs) ys = \text{lenAppend} xs ys
```

To summarize, we use the propositions-as-types principle to make two important contributions. The proof proceeds by induction (N. Vazou, A. Tondwalkar, V. Choudhury, R. G. Scott, R. R. Newton, P. Wadler, and R. Jhala). Finally, we specify and verify induction on natural numbers.

3.4 Consequences

We apply the induction hypothesis to the right conjunct instantiated at n-1.

\[ \text{ind} :: p : \_ \rightarrow ((p \ 0), \ (n : \text{Nat} \rightarrow (p \ (n-1)) \rightarrow (p \ n))) \rightarrow n : \text{Nat} \rightarrow (p \ n) \]

\[ \text{ind} \ p \ (p\varnothing, \ pn) \ \varnothing = p\varnothing \]

\[ \text{ind} \ p \ (p\varnothing, \ pn) \ n = pn \ n \ (\text{ind} \ p \ (p\varnothing, \ pn) \ (n-1)) \]

4 REFINEMENT REFLECTION: $\lambda^R$

Refinement reflection encodes recursive functions in the quantifier-free, SMT logic and it is formalized in three steps. First, we develop a core calculus $\lambda^R$ with an undecidable type system based on denotational semantics. We show how the soundness of the type system allows us to prove theorems using $\lambda^R$. Next, in §5 we define a language $\lambda^S$ that soundly approximates $\lambda^R$ while enabling decidable SMT-based type checking. Finally, in §6 we develop a complete proof search algorithm to automate equational reasoning.
We define \( \mathbb{L} \) which represents finite lists of integers, has two data constructors: \( \text{nil} \) and \( \text{cons} \).

**Values & Expressions** The values of \( \mathbb{L} \) are constants, values, expressions, binders and programs.

**Constants** The constants of \( \mathbb{L} \) include primitive relations \( \circ \), here, the set \{ =, < \}. Moreover, they include the booleans \( \text{True}, \text{False} \), integers \(-1, 0, 1, \ldots\), and logical operators \( \land, !, \ldots \).

**Data Constructors** Data constructors are special constants. For example, the data type \([\text{Int}]\), which represents finite lists of integers, has two data constructors: \([\text{nil}] \) (nil) and \(: (\text{cons})\).

**Values & Expressions** The values of \( \mathbb{L} \) include constants, \( \lambda \)-abstractions \( \lambda x. e \), and fully applied data constructors \( D \) that wrap values. The expressions of \( \mathbb{L} \) include values, variables \( x \), applications \( e e \), and case expressions.

**Binders & Programs** A binder \( b \) is a series of possibly recursive \texttt{let} definitions, followed by an expression. A program \( p \) is a series of \texttt{reflect} definitions, each of which names a function that is reflected into the refinement logic, followed by a binder. The stratification of programs via binders is required so that arbitrary recursive definitions are allowed in the program but cannot be inserted into the logic via refinements or reflection. (We can allow non-recursive \texttt{let} binders in expressions \( e \), but omit them for simplicity.)

### 4.2 Operational Semantics

We define \( \rightarrow \) to be the small step, call-by-name \( \beta \)-reduction semantics for \( \mathbb{L} \). We evaluate reflected terms as recursive \texttt{let} bindings, with termination constraints imposed by the type system:

\[
\text{reflect } x : \tau = e \in p \rightarrow \text{let } \text{rec } x : \tau = e \in p
\]

We define \( \rightarrow^* \) to be the reflexive, transitive closure of \( \rightarrow \). Moreover, we define \( \approx_\beta \) to be the reflexive, symmetric, and transitive closure of \( \rightarrow \).

**Constants** Application of a constant requires the argument be reduced to a value; in a single step, the expression is reduced to the output of the primitive constant operation, \textit{i.e.} \( c v \rightarrow \delta(c, v) \). For example, consider \( = \), the primitive equality operator on integers. We have \( \delta(=, n) \equiv =_n \) where \( \delta(=, m) \) equals \( \text{True} \) iff \( m \) is the same as \( n \).

**Equality** We assume that the equality operator is defined \textit{for all} values, and, for functions, is defined as extensional equality. That is, for all \( f \) and \( f' \), \( (f = f') \rightarrow \text{True} \) iff \( \forall v. f v \approx_\beta f' v \). We
assume source terms only contain implementable equalities over non-function types; while function extensional equality only appears in refinements.

4.3 Types

\( \lambda^R \) types include basic types, which are refined with predicates, and dependent function types. Basic types \( B \) comprise integers, booleans, and a family of data-types \( T \) (representing lists, trees etc.). For example, the data type \([\text{Int}]\) represents lists of integers. We refine basic types with predicates (boolean-valued expressions \( e \)) to obtain basic refinement types \( \{ v : B \mid e \} \). We use \( \downarrow \) to mark provably terminating computations and use refinements to ensure that if \( e : \{ v : B \mid e' \} \), then \( e \) terminates. As discussed by Vazou et al. [2014] termination labels can be checked using refinement types and are used to ensure that refinements cannot diverge as required for soundness of type checking under lazy evaluation. Termination checking is crucial for this work, as combined with syntactic checks for exhaustive definitions, it ensures totality (well-formedness) of expressions as required both by propositions-as-types (§ 3) and termination of PLE (§ 6). Finally, we have dependent function types \( \lambda x : \tau_x \to \tau \) where the input \( x \) has the type \( \tau_x \) and the output \( \tau \) may refer to the input binder \( x \). We write \( B \) to abbreviate \( \{ v : B \mid \text{True} \} \), and \( \tau_x \to \tau \) to abbreviate \( \lambda x : \tau_x \to \tau \) if \( x \) does not appear in \( \tau \).

**Denotations** Each type \( \tau \) denotes a set of expressions \( \llbracket \tau \rrbracket \), that is defined via the operational semantics [Knowles and Flanagan 2010]. Let \( \text{shape}(\tau) \) be the type we get if we erase all refinements from \( \tau \) and \( e : \text{shape}(\tau) \) be the standard typing relation for the typed lambda calculus. Then, we define the denotation of types as:

\[
\llbracket \{ x : B \mid r \} \rrbracket \triangleq \{ e \mid e \vdash \text{if } e \mapsto^* w \text{ then } r[x/w] \mapsto^* \text{True} \}
\]

\[
\llbracket \{ x : B^\downarrow \mid r \} \rrbracket \triangleq \{ \llbracket \{ x : B \mid r \} \rrbracket \} \cap \{ e \mid \exists w. e \vdash^* w \}
\]

\[
\llbracket x : \tau_x \to \tau \rrbracket \triangleq \{ e \mid e : \text{shape}(\tau_x \to \tau), \forall e_x \in \llbracket \tau_x \rrbracket. (e e_x) \in \llbracket \tau / x \rrbracket \}
\]

**Constants** For each constant \( c \) we define its type \( \text{prim}(c) \) such that \( c \in \llbracket \text{prim}(c) \rrbracket \). For example,

\[
\text{prim}(3) \triangleq \{ v : \text{Int}^\downarrow \mid v = 3 \}
\]

\[
\text{prim}(+) \triangleq x : \text{Int}^\downarrow \to y : \text{Int}^\downarrow \to \{ v : \text{Int}^\downarrow \mid v = x + y \}
\]

\[
\text{prim}(\leq) \triangleq x : \text{Int}^\downarrow \to y : \text{Int}^\downarrow \to \{ v : \text{Bool}^\downarrow \mid v \equiv x \leq y \}
\]

4.4 Refinement Reflection

Reflection strengthens function output types with a refinement that reflects the definition of the function in the logic. We do this by treating each reflect-binder \(( \text{reflect } f : \tau = e \in p )\) as a let rec-binder \(( \text{let rec } f : \text{Reflect}(\tau, e) = e \text{ in } p )\) during type checking (rule T-REFL in Figure 7).

**Reflection** We write \( \text{Reflect}(\tau, e) \) for the reflection of the term \( e \) into the type \( \tau \), defined as

\[
\text{Reflect}(\{ v : B^\downarrow \mid r \}, e) \triangleq \{ v : B^\downarrow \mid r \land v = e \}
\]

\[
\text{Reflect}(x : \tau_x \to \tau, \lambda x.e) \triangleq x : \tau_x \to \text{Reflect}(\tau, e)
\]

As an example, recall from § 2 that the reflect \( \text{fib}P \) strengthens the type of \( \text{fib} \) with the refinement \( \text{fib}P \). That is, let the user specified type of \( \text{fib} \) be \( t_{\text{fib}} \) and the its definition be definition \( \lambda n. e_{\text{fib}} \).

\[
t_{\text{fib}} \triangleq \{ v : \text{Int}^\downarrow \mid 0 \leq v \} \to \{ v : \text{Int}^\downarrow \mid 0 \leq v \}
\]

\[
e_{\text{fib}} \triangleq \text{case } x = n \leq 1 \text{ of } \{ \text{True} \to n; \text{False} \to \text{fib}(n-1) + \text{fib}(n-2) \}
\]

Then, the reflected type of \( \text{fib} \) will be:

\[
\text{Reflect}(t_{\text{fib}}, e_{\text{fib}}) = n : \{ v : \text{Int}^\downarrow \mid 0 \leq v \} \to \{ v : \text{Int}^\downarrow \mid 0 \leq v \land v = e_{\text{fib}} \}
\]
**Termination Checking** We defined \( \text{Reflect}(\cdot, \cdot) \) to be a *partial* function that only reflects provably terminating expressions, *i.e.* expressions whose result type is marked with \( \downarrow \). If a non-provably terminating function is reflected in an \( \lambda^R \) expression then type checking will fail (with a reflection type error in the implementation). This restriction is crucial for soundness, as diverging expressions can lead to inconsistencies. For example, reflecting the diverging \( f \times 1 + f \times \) into the logic leads to an inconsistent system that is able to prove \( \theta = 1 \).

**Automatic Reflection** Reflection of \( \lambda^R \) expressions into the refinements happens automatically by the type system, not manually by the user. The user simply annotates a function \( f \) as \( \text{reflect} \ f \). Then, the rule \( \text{T-Refl} \) in Figure 7 is used to type check the reflected function by strengthening the \( f \)'s result via \( \text{Reflect}(\cdot, \cdot) \). Finally, the rule \( \text{T-Let} \) is used to check that the automatically strengthened type of \( f \) satisfies \( f \)'s implementation.

### 4.5 Typing Rules

Next, we present the type-checking rules of \( \lambda^R \), as found in Figure 7.

**Environments and Closing Substitutions** A *type environment* \( \Gamma \) is a sequence of type bindings \( x_1 : \tau_1, \ldots, x_n : \tau_n. \) An environment denotes a set of *closing substitutions* \( \theta \) which are sequences of expression bindings: \( x_1 \mapsto e_1, \ldots, x_n \mapsto e_n \) such that:

\[
[[\Gamma]] = \{ \theta \mid \forall x : \tau \in \Gamma. \theta(x) \in [[\theta \cdot \tau]] \}
\]

where \( \theta \cdot \tau \) applies a substitution to a type (and likewise \( \theta \cdot p \), to a program).

A reflection environment \( R \) is a sequence that binds the names of the reflected functions with their definitions \( f_1 \mapsto e_1, \ldots, f_n \mapsto e_n \). A reflection environment respects a type environment when all reflected functions satisfy their types:

\[
\Gamma \models R \models \forall (f \mapsto e) \in R. \exists \tau. (f : \tau) \in \Gamma \land (\Gamma; R \vdash e : \tau)
\]

**Typing** A judgment \( \Gamma; R \vdash p : \tau \) states that the program \( p \) has the type \( \tau \) in the type environment \( \Gamma \) and the reflection environment \( R \). That is, when the free variables in \( p \) are bound to expressions described by \( \Gamma \), the program \( p \) will evaluate to a value described by \( \tau \).

**Rules** All but two of the rules are the standard refinement typing rules [Knowles and Flanagan 2010; Vazou et al. 2014] except for the addition of the reflection environment \( R \) at each rule. First, rule \( \text{T-Refl} \) is used to extend the reflection environment with the binding of the function name with its definition \( (f \mapsto e) \) and moreover to strengthen the type of each reflected binder with its definition, as described previously in § 4.4. Second, rule \( \text{T-Exact} \) strengthens the expression with a singleton type equating the value and the expression (i.e. reflecting the expression in the type). This is a generalization of the "selfification" rules from [Knowles and Flanagan 2010; Ou et al. 2004] and is required to equate the reflected functions with their definitions. For example, the application \( \text{fib} \ 1 \) is typed as \( \{ v : \text{Int} \mid \text{fib} \ 1 \land v = \text{fib} \ 1 \} \) where the first conjunct comes from the (reflection-strengthened) output refinement of \( \text{fib} \) § 2 and the second comes from rule \( \text{T-Exact} \).

**Well-formedness** A judgment \( \Gamma \vdash \tau \) states that the refinement type \( \tau \) is well-formed in the environment \( \Gamma \). Following Vazou et al. [2014], \( \tau \) is well-formed if all the refinements in \( \tau \) are \( \text{Bool} \)-typed, provably terminating expressions in \( \Gamma \).

**Subtyping** A judgment \( \Gamma; R \vdash \tau_1 \leq \tau_2 \) states that the type \( \tau_1 \) is a subtype of \( \tau_2 \) in the environments \( \Gamma \) and \( R \). Informally, \( \tau_1 \) is a subtype of \( \tau_2 \) if, when the free variables of \( \tau_1 \) and \( \tau_2 \) are bound to expressions described by \( \Gamma \), the denotation of \( \tau_1 \) is *contained in* the denotation of \( \tau_2 \). Subtyping of basic types reduces to denotational containment checking, shown in rule \( \leq\text{-Base} - \lambda^R \). That is, \( \tau_1 \) is a subtype of \( \tau_2 \) under \( \Gamma \) if for any closing substitution \( \theta \) in \( [[\Gamma]] \), \( [[\theta \cdot \tau_1]] \) is contained in \( [[\theta \cdot \tau_2]] \).
Typing

\[ x : \tau \in \Gamma \quad \text{T-VAR} \quad \Gamma ; R \vdash x : \tau \]
\[ \Gamma ; R \vdash c : \text{prim}(c) \quad \text{T-CON} \quad \Gamma ; R \vdash p : \tau \]
\[ (\forall \cdot) \quad \tau \in \Gamma \quad \Gamma ; R \vdash {\cdot} : \tau \]
\[ \Gamma ; R \vdash e : \{ v : B \mid e_r \} \quad \text{T-EXACT} \quad \Gamma ; R \vdash e : \{ v : B \mid e_r \land v = e \} \]
\[ \Gamma ; R \vdash e_1 : (x : \tau x) \to \tau \quad \text{T-APP} \quad \Gamma ; R \vdash e_2 : \tau x \quad \Gamma ; R \vdash e : \tau \]
\[ \forall i. \text{prim}(D_i) = \bar{y}_j : \tau_j \to \{ v : T \mid e_{r_i} \} \quad \Gamma ; R \vdash \text{case } x = e \text{ of } \{ D_i \bar{y}_i \to e_i \} : \tau \]
\[ \Gamma , x : \tau x \vdash \text{reflect } f \vdash : \tau f = e \in p : \tau \quad \text{T-REFL} \]

Well Formedness

\[ \Gamma , v : B ; \emptyset \vdash e : \text{Bool} \quad \text{WF-BASE} \quad \Gamma \vdash \tau x \quad \Gamma , x : \tau x \vdash \tau \quad \text{WF-FUN} \]

Subtyping

\[ \forall \emptyset \in \llbracket \Gamma \rrbracket . \llbracket \emptyset \cdot \{ v : B \mid e_1 \} \rrbracket \subseteq \llbracket \emptyset \cdot \{ v : B \mid e_2 \} \rrbracket \quad \llbracket \text{BASE-} \lambda R \]
\[ \Gamma ; R \vdash \{ v : B \mid e_1 \} \leq \{ v : B \mid e_2 \} \]
\[ \Gamma ; R \vdash \tau _x \leq \tau x \quad \Gamma , x : \tau x \vdash \tau \leq \tau' \quad \llbracket \text{BASE-} \lambda R \]
\[ \Gamma ; R \vdash \tau _x \to \tau \leq x : \tau _x \to \tau' \]

Fig. 7. Typing of \( \lambda^R \)

Soundness Following \( \lambda^U \) [Vazou et al. 2014], in Vazou et al. [2017] we prove that evaluation preserves typing and typing implies denotational inclusion.

**Theorem 4.1.** [Soundness of \( \lambda^R \)]

- **Denotations** If \( \Gamma ; R \vdash p : \tau \) then \( \forall \emptyset \in \llbracket \Gamma \rrbracket . \emptyset \cdot p \in \llbracket \emptyset \cdot \tau \rrbracket \).
- **Preservation** If \( \emptyset ; \emptyset \vdash p : \tau \) and \( p \hookrightarrow w \), then \( \emptyset ; \emptyset \vdash w : \tau \).

Theorem 4.1 lets us prove that if \( \phi \) is a \( \lambda^R \) type interpreted as a proposition (using the mapping of Figure 3) and if there exists a \( p \) so that \( \emptyset ; \emptyset \vdash p : \phi \), the \( \phi \) is valid. For example, in § 2 we verified that the term \( \text{fibUp} \) proves \( n : \text{Nat} \to \{ \text{fib} n \leq \text{fib} (n + 1) \} \). Via soundness of \( \lambda^R \), we get that for each valid input \( n \), the result refinement is valid.

\[ \forall n.0 \leq n \hookrightarrow \text{True} \Rightarrow \text{fib} n \leq \text{fib} (n + 1) \hookrightarrow \text{True} \]
5 ALGORITHMIC CHECKING: $\lambda^S$

$\lambda^S$ is a first order approximation of $\lambda^R$ where higher-order features are approximated with uninterpreted functions and the undecidable type subsumption rule $\preceq$-BASE-$\lambda^R$ is replaced with a decidable one (i.e., $\preceq$-BASE-PLE), yielding an sound and decidable SMT-based algorithmic type system. Figure 6 summarizes the syntax of $\lambda^S$, the sorted (SMT-) decidable logic of quantifier-free equality, uninterpreted functions and linear arithmetic (QF-EUF-LIA) [Barrett et al. 2010; Nelson 1980]. The terms of $\lambda^S$ include integers $n$, booleans $b$, variables $x$, data constructors $D$ (encoded as constants), fully applied unary $\Theta_1$ and binary $\bowtie$ operators, and application $x \overline{p}$ of an uninterpreted function $x$. The sorts of $\lambda^S$ include built-in integer Int and Bool for representing integers and booleans. The interpreted functions of $\lambda^S$, e.g. the logical constants $=$ and $<$, have the function sort $s \rightarrow s$. Other functional values in $\lambda^R$, e.g. reflected $\lambda^R$ functions and $\lambda$-expressions, are represented as first-order values with the uninterpreted sort Fun $s s$.

5.1 Transforming $\lambda^R$ into $\lambda^S$

The judgment $\Gamma \vdash e \rightsquigarrow p$ states that a $\lambda^R$ term $e$ is transformed, under an environment $\Gamma$, into a $\lambda^S$ term $p$. If $\Gamma \vdash e \rightsquigarrow p$ and $\Gamma$ is clear from the context we write $[e]$ and $[p]$ to denote the translation from $\lambda^R$ to $\lambda^S$ and back. Most of the transformation rules are identity and can be found in [Vazou et al. 2017]. Here we discuss the non-identity ones.

Embedding Types We embed $\lambda^R$ types into $\lambda^S$ sorts as:

\[
\begin{align*}
\text{[Int]} & \equiv \text{Int} \\
\text{[T]} & \equiv U \\
\text{[U]} & \equiv \text{Bool}
\end{align*}
\]

Embedding Constants Elements shared on both $\lambda^R$ and $\lambda^S$ translate to themselves. These elements include booleans, integers, variables, binary and unary operators. SMT solvers do not support currying, and so in $\lambda^S$, all function symbols must be fully applied. Thus, we assume that all applications to primitive constants and data constructors are fully applied, e.g. by converting source terms like $(+ 1)$ to $(\lambda z \rightarrow z + 1)$.

Embedding Functions As $\lambda^S$ is first-order, we embed $\lambda$s using the uninterpreted function $1am$.

\[
\begin{align*}
\Gamma, x : \tau_x \vdash e \rightsquigarrow p & \quad \Gamma; \emptyset \vdash (\lambda x.e) : (x : \tau_x \rightarrow \tau) \\
\Gamma \vdash \lambda x.e \rightsquigarrow 1am_{\tau_x \rightarrow \tau} x p
\end{align*}
\]

The term $\lambda x.e$ of type $\tau_x \rightarrow \tau$ is transformed to $1am_{\tau_x \rightarrow \tau} x p$ of sort Fun $s_x s$, where $s_x$ and $s$ are respectively $[\tau_x]$ and $[\tau]$, $1am_{\tau_x \rightarrow \tau}$ is a special uninterpreted function of sort $s_x \rightarrow s \rightarrow$ Fun $s_x s$, and $x$ of sort $s_x$ and $r$ of sort $s$ are the embedding of the binder and body, respectively. As $1am$ is an SMT-function, it does not create a binding for $x$. Instead, $x$ is renamed to a fresh SMT name.

Embedding Applications We embed applications via defunctionalization [Reynolds 1972] using the uninterpreted app:

\[
\begin{align*}
\Gamma \vdash e \rightsquigarrow p' & \quad \Gamma \vdash e \rightsquigarrow p & \quad \Gamma; \emptyset \vdash e : \tau_x \rightarrow \tau \\
\Gamma \vdash e e' \rightsquigarrow \text{app}_{\tau_x \rightarrow \tau}^s p p'
\end{align*}
\]

The term $e e'$, where $e$ and $e'$ have types $\tau_x \rightarrow \tau$ and $\tau_x$, is transformed to $\text{app}_{\tau_x \rightarrow \tau}^s p p' : s$ where $s$ and $s_x$ are $[\tau]$ and $[\tau_x]$, the $\text{app}_{\tau_x \rightarrow \tau}^s$ is a special uninterpreted function of sort Fun $s_x s \rightarrow s_x s$, and $p$ and $p'$ are the respective translations of $e$ and $e'$.

Embedding Data Types We embed data constructors to a predefined $\lambda^S$ constant $s_D$ of sort $[\text{prim}(D)]$: $\Gamma \vdash D \rightsquigarrow s_D$. For each datatype, we create reflected measures that check the top-level constructor and select their individual fields. For example, for lists, we create measures.
We make the type checking from Figure 7 algorithmic by checking subtyping via our novel, SMT-based and because PLE is sound (Theorem 6.2).

The above translation yields the reflected definition for append (++)

Then check

The above selectors can be modeled precisely in the refinement logic via SMT support for ADTs [Nelson 1980]. To generalize, let \( D_i \) be a data constructor such that prim\((D_i)\) \( \equiv \tau_{i,1} \rightarrow \cdots \rightarrow \tau_{i,n} \rightarrow \tau \) Then check \( isD_i \) has the sort \( \text{Fun} [\tau] \text{Bool} \) and select \( sel_{D_i,j} \) has the sort \( \text{Fun} [\tau] [\tau_{i,j}] \).

**Embedding Case Expressions** We translate case-expressions of \( \lambda^R \) into if terms in \( \lambda^S \), by using the check functions in the guards and the select functions for the binders of each case.

\[
\Gamma \vdash e \rightsquigarrow p \quad \Gamma \vdash e_i[\overline{y_i}/\text{sel}_{D_i, x}] [x/e] \rightsquigarrow p_i
\]

\[
\Gamma \vdash \text{case } x = e \text{ of } [D_i \overline{y_i} \rightarrow e_i] \rightsquigarrow \text{if app is}_{D_i} \ p \text{ then } p_i \text{ else } \ldots \text{ else } p_n
\]

The above translation yields the reflected definition for append (++) from (§ 2.5).

**Semantic Preservation** The translation preserves the semantics of the expressions. Informally, if \( \Gamma \vdash e \rightsquigarrow p \), then for every substitution \( \theta \) and every logical model \( \sigma \) that respects the environment \( \Gamma \) if \( \theta \cdot e \rightsquigarrow^* v \) then \( \sigma \models p = [v] \).

### 5.2 Algorithmic Type Checking

We make the type checking from Figure 7 algorithmic by checking subtyping via our novel, SMT-based Proof by Logical Evaluation (PLE) Next, we formalize how PLE makes checking algorithmic and in § 6 we describe the PLE procedure in detail.

**Verification Conditions** Recall that in § 5.1 we defined \([\cdot ]\) as the translation from \( \lambda^R \) to \( \lambda^S \). Informally, the implication or verification condition (VC) \( \Gamma \models p_1 \Rightarrow p_2 \) is valid only if the set of values described by \( p_1 \) is subsumed by the set of values described by \( p_2 \) under the assumptions of \( \Gamma \). \( \Gamma \) is embedded into logic by conjoining the refinements of terminating binders [Vazou et al. 2014]:

\[
[\Gamma] = \bigcup_{x \in \Gamma} [\Gamma, x] \quad \text{where we embed each binder as} \quad [\Gamma, x] = \begin{cases} [e] & \text{if } \Gamma(x) = [x : B^{\emptyset} | e] \\ \text{True} & \text{otherwise.} \end{cases}
\]

**Validity Checking** Instead of directly using the VCs to check validity of programs, we use the procedure PLE that strengthens the assumption environment \([\Gamma] \) with equational properties. Concretely, given a reflection environment \( R \), type environment \( \Gamma \), and expression \( e \), the procedure PLE\(([R], [\Gamma], [e]) \) — we will define \([R]\) in § 6.1 — returns \( \text{true} \) only when the expression \( e \) evaluates to \( \text{true} \) under the reflection and type environments \( R \) and \( \Gamma \).

**Subtyping via VC Validity Checking** We make subtyping, and hence, typing decidable, by replacing the denotational base subtyping rule \( \preceq\text{-BASE}\-\lambda^R \) with the conservative, algorithmic version \( \preceq\text{-BASE}\-\text{PLE} \) that uses PLE to check the validity of the subtyping.

\[
\text{PLE}([R], [\Gamma, v : [v : B^{\emptyset} | e]], [e']) \vdash \Gamma ; R \vdash \preceq\text{-BASE}\-\text{PLE} \quad \frac{\Gamma ; R \vdash \preceq\text{-BASE}\-\lambda^R}{\vdash \Gamma ; R \vdash \preceq\text{-BASE}\-\lambda^S}
\]

This typing rule is sound as functions reflected in \( R \) always respect the typing environment \( \Gamma \) (by construction) and because PLE is sound (Theorem 6.2).

**Lemma 5.1.** If \( \Gamma ; R \vdash \preceq\text{-BASE}\-\text{PLE} \quad \frac{\{v : B | e_1\}}{\{v : B | e_2\}} \) then \( \Gamma ; R \vdash \{v : B | e_1\} \leq \{v : B | e_2\} \).

**Soundness of \( \lambda^S \)** We write \( \Gamma ; R \vdash \preceq\text{-BASE}\-\lambda^S \) for the judgments that can be derived by the algorithmic subtyping rule \( \preceq\text{-BASE}\-\lambda^S \) (instead of \( \preceq\text{-BASE}\-\lambda^R \)) Lemma 5.1 implies the soundness of \( \lambda^S \).

**Theorem 5.2 (Soundness of \( \lambda^S \)).** If \( \Gamma ; R \vdash \preceq\text{-BASE}\-\lambda^S \) then \( \Gamma \vdash e : \tau \).

\textbf{Terms} \quad p, t, b \ ::= \ \lambda^2 \ \text{i}f\text{-free predicates from Figure 6}  \\
\textbf{Functions} \quad F \ ::= \ \lambda \chi.\langle p \Rightarrow b \rangle  \\
\textbf{Definitional Environment} \quad \Psi \ ::= \ \emptyset \ | \ f \mapsto F, \Psi  \\
\textbf{Logical Environment} \quad \Phi \ ::= \ \emptyset \ | \ p, \Phi

\begin{align*}
\text{Unfold} & : \ (\Psi, \Phi) \rightarrow \Phi \\
\text{Unfold}(\Psi, \Phi) &= \ \Phi \cup \bigcup_{f(\overline{t}) \prec \Phi} \text{Instantiate}(\Psi, \Phi, f, \overline{t}) \\
\text{Instantiate} &\left(\Psi, \Phi, f, \overline{t}\right) = \left\{ (\|f(\overline{x})\| = b_i)[\overline{t}/\overline{x}] \ | \ (p_i \Rightarrow b_i) \in \overline{a}, \ \text{SmtValid}(\Phi, p_i[\overline{t}/\overline{x}]) \right\} \\
\text{where} &\quad \lambda\overline{x}.\langle \overline{a} \rangle = \ \Psi(f)  \\
\text{PLE} & : \ (\Psi, \Phi, p) \rightarrow \text{Bool} \\
\text{PLE}(\Psi, \Phi, p) &= \ \text{loop}\left(0, \ \Phi \cup \bigcup_{f(\overline{t}) \prec \Phi} \text{Instantiate}(\Psi, \Phi, f, \overline{t})\right) \\
\text{where} &\quad \text{loop} (i, \Phi_i) = \left\{ \\
&\quad \text{true} \quad | \ \text{SmtValid}(\Phi_i, p) \\
&\quad \text{false} \quad | \ \Phi_{i+1} \subseteq \Phi_i \\
&\quad \text{otherwise} \quad | \ \text{true} \quad | \ \text{loop} (i + 1, \Phi_{i+1}) \\
&\quad \text{where} \quad \Phi_{i+1} = \ \Phi \cup \text{Unfold}(\Psi, \Phi_i) \\
\end{align*}

Fig. 8. Syntax of Predicates, Terms and Reflected Functions

Fig. 9. Algorithm PLE: Proof by Logical Evaluation

6 \textbf{COMPLETE VERIFICATION: PROOF BY LOGICAL EVALUATION}

Next, we formalize our Proof By Logical Evaluation algorithm PLE and show that it is sound (§ 6.1), that it is complete with respect to equational proofs (§ 6.2), and that it terminates (§ 6.3).

6.1 Algorithm

Figure 8 describes the input environments for PLE. The logical environment \( \Phi \) contains a set of hypotheses \( p \), described in Figure 6. The definitional environment \( \Psi \) maps function symbols \( f \) to their definitions \( \lambda \chi.\langle p \Rightarrow b \rangle \), written as \( \lambda \)-abstractions over guarded bodies. Moreover, the body \( b \) and the guard \( p \) contain neither \( \lambda \) nor if. These restrictions do not impact expressiveness: \( \lambda \)-s can be named and reflected, and if-expressions can be pulled out into top-level guards using DeIf(\( \cdot \)), found in Appendix [Vazou et al. 2017]. A definitional environment \( \Psi \) can be constructed from \( R \) as

\[ [R] \equiv \{ f \mapsto \lambda \chi.\text{DeIf}((\langle e \rangle)|f \mapsto \lambda \chi.e) \in R \} \]

\textbf{Notation} We write \( f(\overline{t}) \prec \Phi \) if the \( \lambda^2 \) term \( \text{app} \ldots \text{app} f t_1 \ldots t_n \) is a syntactic subterm of some \( t' \in \Phi \). We abuse notation to write \( f(\overline{t}) \prec t' \) for \( f(\overline{t}) \prec \{ t' \} \). We write SmtValid(\( \Phi, p \)) for SMT validity of the implication \( \Phi \Rightarrow p \).

\textbf{Instantiation \& Unfolding} A term \( q \) is a \((\Psi, \Phi)\)-instance if there exists \( f(\overline{t}) \prec \Phi \) such that:
A set of terms $Q$ is a $(\Psi, \Phi)$-instance if every $q \in Q$ is an $(\Psi, \Phi)$-instance. The unfolding of $\Psi, \Phi$ is the (finite) set of all $(\Psi, \Phi)$-instances. Procedure $\text{Unfold}(\Psi, \Phi)$ shown in Figure 9 computes and returns the conjunction of $\Phi$ and the unfolding of $\Psi, \Phi$. The following properties relate $(\Psi, \Phi)$-instances to the semantics of $\Psi^R$ and SMT validity. Let $R[e]$ denote the evaluation of $e$ under the reflection environment $R$, i.e. $\theta[e] \equiv e$ and $(R, f : e_f)[e] \equiv R[\text{let rec } f = e_f \text{ in } e]$.

**Lemma 6.1.** For every $\Gamma \models R$, and $\theta \in \langle \Gamma \rangle$, 

- **Sat-Inst** If $[e]$ is a $(\langle R \rangle, \langle \Gamma \rangle)$-instance, then $\theta \cdot R[e] \leftrightarrow * \text{ True}$.
- **SMT-Approx** If $\text{SmtValid}(\langle \Gamma \rangle, [e])$ then $\theta \cdot R[e] \leftrightarrow * \text{ True}$.
- **SMT-Inst** If $q$ is a $(\langle R \rangle, \langle \Gamma \rangle)$-instance and $\text{SmtValid}(\langle \Gamma \rangle \cup \{q\}, [e])$ then $\theta \cdot R[e] \leftrightarrow * \text{ True}$.

**The Algorithm** Figure 9 shows our proof search algorithm $\text{PLE}(\Psi, \Phi, p)$ which takes as input a set of reflected definitions $\Psi$, an hypothesis $\Phi$, and a goal $p$. The PLE procedure recursively unfolds function application terms by invoking Unfold until either the goal can be proved using the unfolded instances (in which case the search returns $\text{true}$) or no new instances are generated by the unfolding (in which case the search returns $\text{false}$).

**Soundness** First, we prove the soundness of PLE.

**Theorem 6.2 (Soundness).** If $\text{PLE}(\langle R \rangle, \langle \Gamma \rangle, [e])$ then $\forall \theta \in \langle \Gamma \rangle, \theta \cdot R[e] \leftrightarrow * \text{ True}$.

We prove Theorem 6.2 using the Lemma 6.1 that relates instantiation, SMT validity, and the exact semantics. Intuitively, PLE is sound as it reasons about a finite set of instances by conservatively treating all function applications as uninterpreted [Nelson 1980].

### 6.2 Completeness

Next, we show that our proof search is complete with respect to equational reasoning. We define a notion of equational proof $\Psi, \Phi \vdash t \rightarrow t'$ and prove that if there exists such a proof, then $\text{PLE}(\Psi, \Phi, t = t')$ is guaranteed to return $\text{true}$. To prove this theorem, we introduce the notion of bounded unfolding which corresponds to unfolding definitions $n$ times. We will show that unfolding preserves congruences, and hence, that an equational proof exists if the goal can be proved with some bounded unfolding. Thus, completeness follows by showing that the proof search procedure computes the limit (i.e. fixpoint) of the bounded unfolding. In §6.3 we will show that the fixpoint is computable: there is an unfolding depth at which PLE reaches a fixpoint and hence terminates.

**Bounded Unfolding** For every $\Psi, \Phi$ and $0 \leq n$, the bounded unfolding of depth $n$ is defined by:

\[
\begin{align*}
\text{Unfold}^*(\Psi, \Phi, 0) & \equiv \Phi \\
\text{Unfold}^*(\Psi, \Phi, n + 1) & \equiv \Phi_n \cup \text{Unfold}(\Psi, \Phi_n) \quad \text{where } \Phi_n = \text{Unfold}^*(\Psi, \Phi, n)
\end{align*}
\]

That is, the unfolding at depth $n$ essentially performs Unfold up to $n$ times. The bounded-unfoldings yield a monotonically non-decreasing sequence of formulas such that if two consecutive bounded unfoldings coincide, then all subsequent unfoldings are the same.

**Lemma 6.3 (Monotonicity).** $\forall 0 \leq n. \text{Unfold}^*(\Psi, \Phi, n) \subseteq \text{Unfold}^*(\Psi, \Phi, n + 1)$.

**Lemma 6.4 (Fixpoint).** Let $\Phi_i \equiv \text{Unfold}^*(\Psi, \Phi, i)$. If $\Phi_n = \Phi_{n+1}$ then $\forall n < m. \Phi_m = \Phi_n$.

**Uncovering** Next we prove that every function application term that is uncovered by unfolding to depth $n$ is congruent to a term in the $n$-depth unfolding.
That is, we show that every possible link in a possible equational chain can be proved equal to the source expression via bounded unfolding.

**Lemma 6.7 (Link).** If SmtValid(Unfold⁺(Ψ, Φ ∪ \{ ν = t \}, \nu = t'), \nu = t') then SmtValid(Unfold⁺(Ψ, Φ ∪ \{ ν = t \}, \nu = t'), \nu = t') for every \( \nu \) in the expression. Conversely, if \( \nu \) is not in the expression, then \( \nu \) is not in the proof. In the SMT-realm, the explicit notion of a normal or canonical form is converted to the implicit notion of the equivalence classes of the SMT solver’s congruence closure procedure.

**Completeness of Bounded Unfolding** Finally, we use the fact that unfolding preserves equational links to show that bounded unfolding is complete for equational proofs. That is, we prove by induction on the structure of the equational proof that whenever there is an equational proof of \( t = t' \), there exists some bounded unfolding that suffices to prove the equality.

**Lemma 6.8.** If \( \Psi, \Phi \vdash t \rightarrow t' \) then \( \exists n \leq n. \text{SmtValid(Unfold}^\ast(\Psi, \Phi \cup \{ \nu = t \}, \nu = t') \).}

**PLE is a Fixpoint of Bounded Unfolding** Next, we show that the proof search procedure PLE computes the least-fixpoint of the bounded unfolding and hence, returns true iff there exists some unfolding depth \( n \) at which the goal can be proved.

**Lemma 6.9 (Fixpoint).** PLE(\( \Psi, \Phi, t = t' \)) iff \( \exists n. \text{SmtValid(Unfold}^\ast(\Psi, \Phi \cup \{ \nu = t \}, \nu = t') \).
The proof follows by observing that PLE(Ψ, Φ, t = t′) computes the least-fixpoint of the sequence Φ_i ⊆ Unfold*(Ψ, Φ, i). Specifically, we can prove by induction on i that at each invocation of loop (i, Φ_i) in Figure 9, Φ_i is equal to Unfold*(Ψ, Φ ∪ {ν = t}, i), which then yields the result.

**Completeness of PLE** By combining Lemma 6.9 and Lemma 6.7 we can show that PLE is complete, i.e. if there is an equational proof that t ≈ t’ under Ψ, Φ, then PLE(Ψ, Φ, t ≈ t’) returns true.

**Theorem 6.10 (Completeness).** If Ψ, Φ ⊢ t ≈ t’ then PLE(Ψ, Φ, t ≈ t’) = true.

### 6.3 PLE Terminates

So far, we have shown that our proof search procedure PLE is both sound and complete. Both of these are easy to achieve simply by enumerating all possible instances and repeatedly querying the SMT solver. Such a monkeys-with-typewriters approach is rather impractical: it may never terminate. Fortunately, next, we show that in addition to being sound and complete with respect to equational proofs, if the hypotheses are transparent, then our proof search procedure always terminates. Next, we describe transparency and explain intuitively why PLE terminates. We then develop the formalism needed to prove the termination theorem 6.16.

**Transparency** An environment Γ is inconsistent if SmtValid(⟨Γ⟩, false). An environment Γ is inhabited if there exists some θ ∈ ⟨Γ⟩. We say Γ is transparent if it is either inhabited or inconsistent. As an example of a non-transparent Φ_0 consider the predicate 1enA xs = 1 + 1enB xs, where 1enA and 1enB are both identical definitions of the list length function. Clearly there is no θ that causes the above predicate to evaluate to true. At the same time, the SMT solver cannot (using the decidable, quantifier-free theories) prove a contradiction as that requires induction over xs. Thus, non-transparent environments are somewhat pathological, and in practice, we only invoke PLE on transparent environments. Either the environment is inconsistent, e.g. when doing a proof-by-contradiction, or e.g. when doing a proof-by-case-analysis we can easily find suitable concrete values via random [Claessen and Hughes 2000] or SMT-guided generation [Seidel et al. 2015].

**Challenge: Connect Concrete and Logical Semantics** As suggested by its name, the PLE algorithm aims to lift the notion of evaluation or computations into the level of the refinement logic. Thus, to prove termination, we must connect the two different notions of evaluation, the concrete (operational) semantics and the logical semantics being used by PLE. This connection is trickier than appears at first glance. In the concrete realm totality ensures that every reflected function f will terminate when run on any individual value v. However, in the logical realm, we are working with infinite sets of values, compactly represented via logical constraints. In other words, the logical realm can be viewed (informally) as an abstract interpretation, of the concrete semantics. We must carefully argue that despite the approximation introduced by the logical abstraction, the abstract interpretation will also terminate.

**Solution: Universal Abstract Interpretation** We make this argument in three parts. First, we formalize how PLE performs computation at the logical level via logical steps and logical traces. We show (Lemma 6.13) that the logical steps form a so-called universal (or “must”) abstraction of the concrete semantics [Clarke et al. 1992; Cousot and Cousot 1977]. Second, we show that if PLE diverges, it is because it creates a strictly increasing infinite chain, Unfold*(Ψ, Φ, 0) ⊂ Unfold*(Ψ, Φ, 1) . . . which corresponds to an infinite logical trace. Third, as the logical computation is universal abstraction we use inhabitation to connect the two realms, i.e. to show that an infinite logical trace corresponds to an infinite concrete trace. The impossibility of the latter must imply the impossibility of the former, i.e. PLE terminates. Next, we formalize the above to obtain Theorem 6.16.
Logical Evaluation Terminates

Finally, we prove that the proof search procedure terminates if and only if every logical step corresponds to a finite logical trace. We prove this result by contradiction. Specifically, we show by Lemma 6.13 that an infinite (∥R, [Γ]∥)-trace combined with fact that Γ is inhabited yields an infinite strictly ascending chain of unfoldings, which contradicts totality. Hence, all the ([R], [Γ]) logical traces must be finite.

Theorem 6.14 (Finite-Trace). If Γ is inhabited then every ([R], [Γ])-trace is finite.

Ascending Chains and Traces

If unfolding Ψ, Φ yields an infinite chain Φ₀ ⊆ . . . ⊆ Φₙ . . . , then Ψ, Φ has an infinite logical trace. We construct the trace by selecting, at level i, (i.e. in Φᵢ), an application term fᵢ(Γᵢ) that was created by unfolding an application term at level i − 1 (i.e. in Φᵢ₋₁).

Lemma 6.15 (Ascending Chains). Let Φᵢ = Unfold*(Ψ, Φ, i). If there exists an (infinite) ascending chain Φ₀ ⊆ . . . ⊆ Φₙ . . . then there exists an (infinite) logical trace f₀(Γ₀), . . . , fₙ(Γₙ), . . .

Logical Evaluation Terminates

Finally, we prove that the proof search procedure PLE terminates. If PLE loops forever, there must be an infinite strictly ascending chain of unfoldings Φᵢ, and hence, by Lemma 6.15, an infinite logical trace, which, by Theorem 6.14, is impossible.

Theorem 6.16 (Termination). If Γ is transparent then PLE([R], [Γ], p) terminates.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Common</th>
<th>Without PLE Search</th>
<th>With PLE Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Impl (l)</td>
<td>Spec (l)</td>
<td>Proof (l)</td>
</tr>
<tr>
<td>Arithmetic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci</td>
<td>7</td>
<td>10</td>
<td>38</td>
</tr>
<tr>
<td>Ackermann</td>
<td>20</td>
<td>73</td>
<td>196</td>
</tr>
<tr>
<td>Class Laws</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monoid</td>
<td>33</td>
<td>50</td>
<td>109</td>
</tr>
<tr>
<td>Functor</td>
<td>48</td>
<td>44</td>
<td>93</td>
</tr>
<tr>
<td>Applicative</td>
<td>62</td>
<td>110</td>
<td>241</td>
</tr>
<tr>
<td>Monad</td>
<td>63</td>
<td>42</td>
<td>122</td>
</tr>
<tr>
<td>Higher-Order Properties</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical Properties</td>
<td>0</td>
<td>20</td>
<td>33</td>
</tr>
<tr>
<td>Fold Universal</td>
<td>10</td>
<td>44</td>
<td>43</td>
</tr>
<tr>
<td>Functional Correctness</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAT-solver</td>
<td>92</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>Unification</td>
<td>51</td>
<td>60</td>
<td>85</td>
</tr>
<tr>
<td>Deterministic Parallelism</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conc. Sets</td>
<td>597</td>
<td>329</td>
<td>339</td>
</tr>
<tr>
<td>n-body</td>
<td>163</td>
<td>251</td>
<td>101</td>
</tr>
<tr>
<td>Par. Reducers</td>
<td>30</td>
<td>212</td>
<td>124</td>
</tr>
<tr>
<td>Total</td>
<td>1176</td>
<td>1279</td>
<td>1524</td>
</tr>
</tbody>
</table>

Table 1. We report verification Time (in seconds, on a 2.3GHz Intel® Xeon® CPU E5-2699 v3 with 18 physical cores and 64GiB RAM.), the number of SMT queries and size of Proofs (in lines). The Common columns show sizes of common Implementations and Specifications. We separately consider proofs Without and With PLE Search.

7 EVALUATION

We have implemented reflection and PLE in LIQUID HASKELL [Vazou et al. 2014]. Table 1 summarizes our evaluation which aims to determine (1) the kinds of programs and properties that can be verified, (2) how PLE simplifies writing proofs, and (3) how PLE affects the verification time.

Benchmarks We summarize our benchmarks below, see [Vazou et al. 2017] for details.

- **Arithmetic** We proved arithmetic properties for the textbook Fibonacci function (c.f. § 2) and the 12 properties of the Ackermann function from [Tourlakis 2008].
- **Class Laws** We proved the monoid laws for the Peano, Maybe and List data types and the Functor, Applicative, and Monad laws, summarized in Figure 11, for the Maybe, List and Identity monads.
- **Higher Order Properties** We used natural deduction to prove textbook logical properties as in § 3. We combined natural deduction principles with PLE-search to prove universality of right-folds, as described in [Hutton 1999] and formalized in AGDA [Mu et al. 2009].
- **Functional Correctness** We proved correctness of a SAT solver and a unification algorithm as implemented in Zombie [Casinghino et al. 2014]. We proved that the SAT solver takes as input a formula $f$ and either returns Nothing or an assignment that satisfies $f$, by reflecting the notion of satisfaction. Then, we proved that if the unification $\text{unify } s \ t$ of two terms $s$ and $t$ returns a substitution $su$, then applying $su$ to $s$ and $t$ yields identical terms. Note that, while the unification function can itself diverge, and hence, cannot be reflected, our method allows terminating and diverging functions to soundly coexist.
• **Deterministic Parallelism** Retrofitting verification onto an existing language with a mature parallel run-time allows us to create three deterministic parallelism libraries that, for the first time, verify implicit assumptions about associativity and ordering that are critical for determinism (c.f. [Vazou et al. 2017] for extended description). First, we proved that the ordering laws hold for keys inserted into LVar-style concurrent sets [Kuper et al. 2014]. Second, we used monad-par [Marlow et al. 2011] to implement an n-body simulation, whose correctness relied upon proving that a triple of Real (implementing) 3-d acceleration was a Monoid. Third, we built a DPJ-style [Bocchino et al. 2009] parallel-reducers library whose correctness relied upon verifying that the reduced arguments form a CommutativeMonoid, and which was the basis of a parallel array sum.

**Proof Effort** We split the total lines of code of our benchmarks into three categories: Spec represents the refinement types that encode theorems, lemmas, and function specifications; Impl represents the rest of the Haskell code that defines executable functions; Proofs represent the sizes of the Haskell proof terms (i.e. functions returning Prop). Reflection and PLE are optionally enabled using pragmas; the latter is enabled either for a whole file/module, or per top-level function.

**Runtime Overhead** Proof terms have no runtime overhead as they will never be evaluated. When verification of an executable term depends on theorems, we use the below withTheorem function

```haskell
withTheorem :: x:a -> Proof -> {v:a | v == x}
withTheorem x _ = x
```

that inserts the proof argument into the static verification environment, relying upon laziness to not actually evaluate the proof. For example, when verification depends on the associativity of append on the lists xs, ys, and zs, the invocation withTheorem xs (app_assoc xs ys zs) extends the (static) SMT verification environment with the instantiation of the associativity theorem of Figure 1. This invocation adds no runtime overhead, since even though app_assoc xs ys zs is an expensive recursive function, it will never actually get evaluated. To ensure that proof terms are not evaluated in runtime, without using laziness, one can add one rewrite rule for each proof term that replaces the term with unit. For example, the rewrite rule for app_assoc is

```haskell
{-# RULES " assoc/runtime" forall xs ys zs. app_assoc xs ys zs = () #-}
```

Such rules are sound, since each proof term is total, thus provably reduces to unit.

**Results** The main highlights of our evaluation are the following. (1) Reflection allows for the specification and verification of a wide variety of important properties of programs. (2) PLE drastically reduces the proof effort: by a factor of 2 − 5× — shrinking the total lines of proof from 1524 to 638— making it quite modest, about the size of the specifications of the theorems. Since PLE searches for equational properties, there are some proofs, that rarely occur in practice, that PLE cannot simplify, e.g. the logical properties from § 3. (3) PLE does not impose a performance penalty: even though proof search can make an order of magnitude many more SMT queries — increasing the total SMT queries from 1626 without PLE to 4068 with PLE— most of these queries are simple and it is typically faster to type-check the compact proofs enabled by PLE than it is to type-check the 2 − 5× longer explicit proofs written by a human.

8 RELATED WORK

**SMT-Based Verification** SMT-solvers have been extensively used to automate program verification via Floyd-Hoare logics [Nelson 1980]. LEON introduces an SMT-based algorithm that is complete for catamorphisms (folds) over ADTs [Suter et al. 2010], and a semi-decision procedure that is guaranteed to find satisfying assignments (models) for queries over arbitrary recursive functions, if they exist [Suter et al. 2011]. Our work is inspired by DAFNY’s Verified Calculations [Leino Proc. ACM Program. Lang., Vol. 1, No. 1, Article 1. Publication date: October 2017.
<table>
<thead>
<tr>
<th>Monoid (for Peano, Maybe, List)</th>
<th>Functor (for Maybe, List, Id)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left Id. mempty $x \circ \equiv x$</td>
<td>Id. $fmap\ id\ xs \equiv id\ xs$</td>
</tr>
<tr>
<td>Right Id. $x \circ mempty \equiv x$</td>
<td>Distr. $fmap\ (g \circ h)\ xs \equiv (fmap\ g \circ fmap\ h)\ xs$</td>
</tr>
<tr>
<td>Assoc $\left((x \circ y) \circ z \equiv x \circ (y \circ z)\right)$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Applicative (for Maybe, List, Id)</th>
<th>Monad (for Maybe, List, Id)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id. $pure\ id \otimes v \equiv v$</td>
<td>Left Id. $return\ a \gg f \equiv f\ a$</td>
</tr>
<tr>
<td>Comp. $pure\ (\circ) \otimes u \otimes v \otimes w \equiv u \otimes (v \otimes w)$</td>
<td>Right Id. $m \gg return \equiv m$</td>
</tr>
<tr>
<td>Hom. $pure\ f \otimes pure\ x \equiv pure\ (f\ x)$</td>
<td>Assoc $\left((m \gg f) \gg g \equiv m \gg (\lambda x \to f\ x \gg g)\right)$</td>
</tr>
<tr>
<td>Inter. $u \otimes pure\ y \equiv pure\ ($_\ y) \otimes u$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ord (for Int, Double, Either, (,) )</th>
<th>Commutative Monoid (for Int, Double, (,) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refl. $x \leq x$</td>
<td>Comm. $x \circ y \equiv y \circ x$</td>
</tr>
<tr>
<td>Antisym. $x \leq y \land y \leq x \Rightarrow x \equiv y$</td>
<td></td>
</tr>
<tr>
<td>Trans. $x \leq y \land y \leq z \Rightarrow x \leq z$</td>
<td>(including Monoid laws)</td>
</tr>
<tr>
<td>Total. $x \leq y \lor y \leq x$</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 11. Summary of Verified Typeclass Laws

and Polikarpova 2016] but differs in (1) our use of reflection instead of axiomatization, (2) our use of refinements to compose proofs, and (3) our use of PLE to automate reasoning about user-defined functions. DAFNY (and F* [Swamy et al. 2016]) encode user-functions as axioms and use a fixed fuel to instantiate functions up to some fixed unfolding depth [Amin et al. 2014]. While the fuel-based approach is incomplete, even for equational or calculational reasoning, it may, although rare in practice, quickly time out after a fixed, small number of instantiations rather than perform an exhaustive proof search like PLE. Nevertheless, PLE demonstrates that it is possible to develop complete and practical algorithms for reasoning about user-defined functions.

**Proving Equational Properties** Several authors have proposed tools for proving (equational) properties of (functional) programs. Systems of Sousa and Dillig [2016] and Asada et al. [2015] extend classical safety verification algorithms, respectively based on Floyd-Hoare logic and refinement types, to the setting of relational or $k$-safety properties that are assertions over $k$-traces of a program. Thus, these methods can automatically prove that certain functions are associative, commutative etc. but are restricted to first-order properties and are not programmer-extensible. Zeno [Sonnex et al. 2012] generates proofs by term rewriting and Halo [Vytiniotis et al. 2013] uses an axiomatic encoding to verify contracts. Both the above are automatic, but unpredictable and not programmer-extensible, hence, have been limited to far simpler properties than the ones checked here. HERMIT [Farmer et al. 2015] proves equalities by rewriting the GHC core language, guided by user specified scripts. Our proofs are Haskell programs, SMT solvers automate reasoning, and, importantly, we connect the validity of proofs with the semantics of the programs.

**Dependent Types in Programming** Integration of dependent types into Haskell has been a long standing goal [Eisenberg and Stolarek 2014] that dates back to Cayenne [Augustsson 1998], a Haskell-like, fully dependent type language with undecidable type checking. Our approach differs significantly in that reflection and PLE use SMT-solvers to drastically simplify proofs over decidable theories. Zombie [Sjöberg and Weirich 2015] investigates the design of a dependently typed language where SMT-style congruence closure is used to reason about the equality of terms. However, Zombie explicitly eschews type-level computation as the authors write "equalities that follow from $\beta$-reduction" are "incompatible with congruence closure". Due to this incompleteness,
the programmer must use explicit join terms to indicate where normalization should be triggered, even so, equality checking is based on fuel, hence, is incomplete.

**Theorem Provers** Reflection shows how to retrofit deep specification and verification in the style of **Agda** [Norell 2007], **Coq** [Bertot and Castéran 2004] and **ISABELLE** [Nipkow et al. 2002] into existing languages via refinement typing and **PLE** shows how type-level computation can be made compatible with SMT solvers' native theory reasoning yielding a powerful new way to automate proofs (§ 2.5). An extensive comparison [Vazou et al. 2017] between our approach and mature theorem provers like Coq, Agda, and ISABELLE reveals that these provers have two clear advantages over our approach: they emit certificates, so they rely on a small trusted computing base, and they have decades-worth of tactics, libraries and proof scripts that enable large scale proof engineering. Some tactics even enable embedding of SMT-based proof search heuristics, e.g. SLEDGEHAMMER [Blanchette et al. 2011], that is widely used in ISABELLE. However, this search does not have the completeness guarantees of PLE. The issue of extracting checkable certificates from SMT solvers is well understood [Chen et al. 2010; Necula 1997] and easy to extend to our setting. However, the question of extending SMT-based verifiers with tactics and scriptable proof search, and more generally, incorporating interactivity in the style of proof-assistants, perhaps enhanced by proof-completion hints remains an interesting direction for future work.

9 CONCLUSIONS AND FUTURE WORK

Our results identify a new design for deductive verifiers wherein: (1) via Refinement Reflection, we can encode natural deduction proofs as SMT-checkable refinement typed programs; (2) via Proof by Logical Evaluation we can combine the complementary strengths of SMT- (i.e., decision procedures) and TT-based approaches (i.e., type-level computation) to obtain completeness guarantees when verifying properties of user-defined functions. However, the increased automation of SMT and proof-search can sometimes make it harder for a user to debug failed proofs. In future work, it would be interesting to investigate how to add interactivity to SMT based verifiers, in the form of tactics and scripts or algorithms for synthesizing proof hints, and by designing new ways to explain and fix refinement type errors.

ACKNOWLEDGMENTS

We thank the anonymous referees, our shepherd Koen Claessen, and Rustan Leino for their feedback on earlier versions of this paper. Special thanks to George Karachalias and Valentin Robert for their input on proof combinators, and to Joachim Breitner and Eric Seidel for their suggestion on run-time proof elimination. This work was supported by the EPSRC programme grant EP/K034413/1, the National Science Foundation under Grant Numbers CCF-1618756, CNS-1518765, CCF-1518844, CCF-1422471, CCF-1223850, and CCF-1218344, and a generous gift from Microsoft Research.

REFERENCES


P. Vekris, B. Cosman, and R. Jhala. 2016. Refinement types for TypeScript. In PLDI.


P. Wadler. 2015. Propositions As Types. In Communications of the ACM.