Refinement Reflection: Complete Verification with SMT

NIKI VAZOU, University of Maryland, USA
ANISH TONDWALKAR, University of California, San Diego, USA
VIKRAMAN CHOUDHURY, Indiana University, USA
RYAN G. SCOTT, Indiana University, USA
RYAN R. NEWTON, Indiana University, USA
PHILIP WADLER, University of Edinburgh and Input Output HK, UK
RANJIT JHALA, University of California, San Diego, USA

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 the SMT logic in a precise fashion that permits decidable verification. 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. Finally, we introduce a proof-search algorithm called *Proof by Logical Evaluation* that uses techniques from model checking and 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 have used reflection to build the first library that actually verifies assumptions about associativity and ordering that are crucial for safe deterministic parallelism.

CCS Concepts: • **Software and its engineering → General programming languages**; **General programming languages**;

Additional Key Words and Phrases: refinement types, theorem proving, Haskell, automatic verification

ACM Reference Format:
Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.* 2, POPL, Article 53 (January 2018), 31 pages. https://doi.org/10.1145/3158141

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 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

Authors' addresses: Niki Vazou, University of Maryland, USA, nvazou@cs.umd.edu; Anish Tondwalkar, University of California, San Diego, USA, atondwal@cs.ucsd.edu; Vikraman Choudhury, Indiana University, USA; Ryan G. Scott, Indiana University, USA; Ryan R. Newton, Indiana University, USA; Philip Wadler, University of Edinburgh and Input Output HK, UK, wadler@inf.ed.ac.uk; Ranjit Jhala, University of California, San Diego, USA, jhala@cs.ucsd.edu.

2475-1421/2018/1-ART53
https://doi.org/10.1145/3158141

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:Integer | 0 ≤ v && 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]. In this work, we extend refinement types with refinement reflection, leading to the following three contributions.

1. Refinement Reflection Our first contribution is the notion of *refinement reflection*. To reason about user-defined functions, the function's implementation is *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 the Curry-Howard isomorphism [Howard 1980]) to show that SMT-based verifiers can express any natural deduction proof, thus providing 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 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, since 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 ≤ v` from the quantifier-free logic of linear arithmetic and uninterpreted functions (QF-UFLIA [Barrett et al. 2010]):

```
type Nat = { v:Integer | 0 ≤ 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 → 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 automatically verifies that if `fib` is invoked with a non-negative `Integer`, then it terminates and yields a non-negative `Integer`.

Propositions We 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 the Curry-Howard isomorphism, refinements encode universally-quantified propositions as *dependent function* types of the form:

```
type Plus_comm = x:Integer → y:Integer → { 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 → (m::Integer, {n < m})
```

The notation `(m :: t, t')` describes dependent pairs where the name `m` of the first element can appear inside refinements of 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 $\wedge$ and $\vee$). In § 3, we describe how to encode logical connectives using data types, *e.g.* conjunction as a product and disjunction as a union, how to specify arbitrary, quantified propositions using refinement types, *i.e.* have complete specifications, and how to verify those propositions using refinement type checking.

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

```
plus_2_2 :: Plus_2_2          plus_comm :: Plus_comm          int_up :: Int_up
plus_2_2 = ()                 plus_comm = \x y → ()           int_up = \n → (n+1,())
```

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

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

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

Soundness and Lazy Evaluation Readers familiar with Haskell's lazy semantics may be concerned that "bottom", which inhabits all types, makes our proofs suspect. Fortunately, as described in Vazou et al. [2014], Liquid Haskell, by default, checks that user defined functions provably terminate and are total (*i.e.* return non-bottom values, and do not throw any exceptions), which makes our proofs sound. Liquid Haskell checks that each function is terminating using a *termination metric i.e.* a natural number that decreases at each recursive call. For instance, if we generalize the signature of `fib` to `Integers`, as shown below, then Liquid Haskell reports a termination error:

```
fib :: n:Integer → Integer
```

Liquid 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.

```
fib :: n:Nat → Nat / [n]
```

The above type signature is explicitly annotated with the user specified termination metric `/ [n]` declaring that `n` is the decreasing natural number. Liquid 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.

Not all Haskell functions terminate. The `lazy` annotation deactivates termination checking, *e.g.* for the the `diverge` function shown below. Haskell terms marked as `lazy` could *unsoundly* be used as proof terms, much like Coq's unsoundness with respect to `Admitted`. For example, the following is accepted by Liquid Haskell

```
lazy diverge
diverge :: x:Integer → { x = 0 }
diverge x = diverge x
```

Totality Checking Liquid Haskell further checks that all user-specified functions are totally defined. For instance the below definition

```
fibPartial 0 = 0
fibPartial 1 = 1
```

generates a totality error. The above definition is completed, by GHC, with an `error` invocation

```
fibPartial 0 = 0
fibPartial 1 = 1
fibPartial _ = error "undefined"
```

Thus, to check totality, Liquid Haskell simply ascribes the precondition `false` to `error`

```
error :: { v:String | false } → a
```

Thus, to typecheck `fibPartial`, Liquid Haskell needs to prove that the call to `error` is dead-code, *i.e.* happens under an inconsistent environment. As this check fails, Liquid Haskell generates a totality error. As with termination checking, Liquid Haskell provides an *unsound* error function `unsoundError` with no false precondition.

In the sequel (and in our evaluation) we assume that proof terms are generated without any unsound uses of `lazy` or `unsafeError`. However, note that lazy, diverging functions can be soundly verified, and diverging code can soundly co-exist with terminating proof terms: we refer the reader to Vazou et al. [2014] for details.

2.2 Refinement Reflection

Suppose we wish to prove properties about the `fib` function, *e.g.* that `{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 `Nat → Nat` which is too weak to verify `{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* `fib :: Integer → 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* $\forall n, m.\ n = m \implies \texttt{fib}\ n = \texttt{fib}\ m$. On its own, the uninterpreted function is not terribly useful: we cannot check `{fib 2 = 1}` as the SMT solver cannot prove the VC $\mathit{true} \Rightarrow \texttt{fib}\ 2 = 1$ 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:

```
fib :: n:Nat → { v:Nat | v = fib n && fibP n }
```

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

```
fibP n ≐ n == 0 ⇒ fib n = 0
       ∧ n == 1 ⇒ fib n = 1
       ∧ n  > 1 ⇒ fib n = fib (n-1) + fib (n-2)
```

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 `{fib 2 = 1}` by:

```
pf_fib2 :: { fib 2 = 1 }
pf_fib2 = let { t0 = fib 0; t1 = fib 1; t2 = fib 2 } 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:

$$(\texttt{fibP}\ 0 \ \wedge\ \texttt{fibP}\ 1 \ \wedge\ \texttt{fibP}\ 2) \ \Rightarrow\ (\texttt{fib}\ 2 = 1)$$

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

```
pf_fib2' :: {v:[Nat] | fib 2 = 1 }
pf_fib2' = [ fib 0, fib 1, fib 2 ]
```

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

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, $\odot$, for logical operators in the theory QF-UFLIA, $\odot \in \{=, \neq, \leq, <, \geq, >\}$. In Haskell code, to avoid collisions with existing operators, we further append a colon ":" to these operators, so that "=" becomes the Haskell operator `(=:)`. The refinement type of $\odot$ *requires* that $x \odot y$ holds and then *ensures* that the returned value is equal to `x`. For example, we define `(=:)` as:

```
(=:) :: x:a → y:{ a | x = y } → { v:a | v = x }
x =: _ = x
```

and use it to write the following equational proof:

```
fib2_1 :: { fib 2 = 1 }
fib2_1 =  fib 2 =: fib 1 + fib 0 =: 1 ** QED
```

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

```
data QED = QED              (**) :: a → QED → Prop
                            _ ** QED = ()
```

Proof Arguments Often, we need to compose lemmas into larger theorems. For example, to prove `fib 3 = 2` we may wish to reuse `fib2_1` as a lemma. We do so by defining a variant of `(=:)`, written as `(=?)`, that takes an explicit proof argument:

```
(=?) :: x:a → y:a → { Prop | x = y } → { v:a | v = x }
x =? _ _ = x
```

We use the `(=?)` combinator to prove that `fib 3 = 2`.

```
fib3_2 :: { fib 3 = 2 }
fib3_2 = (fib 3 =: fib 2 + fib 1 =? 2 $ fib2_1) ** QED
```

Here `fib 2` is not important to unfold, because `fib2_1` already provides the same information.

"Because" Combinators Observe that the proof term `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 `(=?)`.

```
(∵) :: (Prop → a) → Prop → a
f ∵ y = f y
```

We use the "because" combinator to remove the parentheses from the proof term `fib3_2`.

```
fib3_2 :: { fib 3 = 2 }
fib3_2 = fib 3 =: fib 2 + fib 1 =? 2 ∵ fib2_1 ** 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 $\odot$. We define the class `OptEq` to have one method `(=.)` that takes two arguments of type `a`, to be compared for equality, and returns a value of type `r`.

```
class OptEq a r where
  (=.) :: a → a → r
```

When instantiating the result type to be the same as the argument type `a`, `(=.)` behaves 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 → { Prop | 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

```
fib3_2 :: { fib 3 = 2 }
fib3_2 = fib 3 =. fib 2 + fib 1 =. 2 ∵ fib2_1 ** QED
```

Arithmetic and Ordering Next, lets see how we can use arithmetic and ordering to prove that `fib` is (locally) increasing, *i.e.* for all n, $\texttt{fib}\ n \leq \texttt{fib}\ (n+1)$.

```
type Up f = n:Nat → { f n ≤ f (n + 1) }

fibUp   :: Up fib
fibUp 0 =  fib 0 <. fib 1                        ** QED
fibUp 1 =  fib 1 ≤. fib 1 + fib 0      =. fib 2      ** QED
fibUp n =  fib n ≤. fib n + fib (n-1) =. fib (n+1) ** QED
```

Case Splitting The proof `fibUp` works by splitting cases on the value of `n`. In the 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 VCs are (automatically) proved as the SMT solver concludes $0 < 1$ and $1 + 0 \leq 1$ respectively. When `n` is greater than one, `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`)

```
type Mono = f:(Nat → Integer) → Up f → x:Nat → y:{ x < y } → { f x ≤ f y }

fMono :: Mono / [y]
fMono f up x y
  | x+1 == y = f x ≤. f (x+1) ∵ up x ≤. f y                          ** QED
  | x+1  < y = f 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:

```
fibMono :: n:Nat → m:{ n < m } → { fib n ≤ fib m }
fibMono = fMono fib fibUp
```

2.4 Complete Verification: Automating Equational Reasoning

While equational proofs can be very easy, writing them out can quickly get exhausting. Lets 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]

- ***Guard Normal Form:*** First, as shown in the definition of `fibP` in § 2.2, each reflected function is transformed into a *guard normal form* $\wedge_i(p_i \Rightarrow f(\overline{x}) = b_i)$ *i.e.* a collection of *guards* p_i and their corresponding definition b_i.
- ***Unfolding:*** Second, given a VC of the form $\Phi \Rightarrow p$, we iteratively *unfold* function application terms in Φ 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 the VC $true \Rightarrow$ `fib` $3 = 2$, the guard $3 > 1$ of the application `fib` 3 is trivially enabled, *i.e.* is true, hence we strengthen the hypothesis Φ with the equality `fib` $3 =$ `fib` $(3-1) +$ `fib` $(3-2)$ corresponding to unfolding the definition of `fib` at `3`.
- ***Fixpoint:*** We repeat the unfolding process until either the VC is proved or we have reached a fixpoint, *i.e.* no further unfolding is enabled. For example, the fixpoint computation of `fib` 3 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. (The user must still provide instantiations of lemmas and induction hypotheses.) 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 to supply the relevant induction hypotheses or helper lemma applications. The remaining long chains of calculations are performed automatically via SMT-based PLE. That is, the user must provide case statements and the recursive structure, but can elide the long chains of =. applications. To wit, with complete proof search, the proofs of § 2.3 shrink to:

```
fib3_2 :: {fib 3 = 2}
fib3_2 = ()
```

```
fMono :: Mono / [y]
fMono f up x y
  | x+1 == y = up x
  | x+1  < y = up (y-1) &&& fMono up x (y-1)
```

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): $\forall n.$ `fibP` n after which the SMT solver may instantiate the above axiom at `3`, `2`, `1` and `0` in order to automatically prove `{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

```
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.

instantiations *ad infinitum*. To prevent matching loops an expert must carefully craft "triggers" or alternatively, provide a "fuel" parameter that bounds the depth of instantiation [Amin et al. 2014]. Both these approaches ensure termination, but can cause the axiom to not be instantiated at the right places, thereby rendering the VC checking *incomplete*. The incompleteness is illustrated by the following example from the Dafny benchmark suite [Leino 2016]

```
pos n | n < 0     = 0                 test   :: y:{y > 5} → {pos n = 3 + pos (n-3)}
      | otherwise = 1 + pos (n-1)     test _ = ()
```

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:

```
reflect (++) :: [a] → [a] → [a]        reflect map :: (a → b) → [a] → [b]
[]     ++ ys = ys                       map f []     = []
(x:xs) ++ ys = x : (xs ++ ys)           map f (x:xs) = f x : map f xs
```

In § 5.1 we will describe how the reflection mechanism illustrated via `fibP` is extended to account for ADTs using SMT-decidable selection and projection operations, which reflect the definition of `xs ++ ys` into the refinement as: `if isNil` *xs* `then` *ys* `else sel1` *xs* `:` `(sel2` *xs* `++` *ys*`)`. We require an explicit **`reflect`** annotation 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. Recall that Liquid Haskell verifies that all reflected functions, like `(++)` and `map` here, are total [Vazou et al. 2014] and rejects the code otherwise.

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 *identity* operation, (2) appending is an *associative* operation, and (3) map *distributes* over function composition:

```
type AppendNilId = xs:_ → { xs ++ [] = xs }
type AppendAssoc = xs:_ → ys:_ → zs:_ → { xs ++ (ys ++ zs) = (xs ++ ys) ++ zs }
type MapFusion   = f:_  → g:_  → xs:_ → { map (f . g) xs = map f (map g xs) }
```

Coq

```
Theorem swap_idemp:
  ∀ al, swap (swap al) = swap al.
Proof. intros.
      destruct al as [ | a al].
      simpl. reflexivity.
      destruct al as [ | b al].
      simpl. reflexivity.
      simpl. bdestruct (b <? a).
    *
      simpl. bdestruct (a <? b).
      omega. reflexivity.
    *
      simpl. bdestruct (b <? a).
      omega. reflexivity.    Qed.
```

Agda

```
swap_idemp : ∀ xs → swap (swap xs) ≡ swap xs
swap_idemp [] = P.refl
swap_idemp (x₁ :: []) = P.refl
swap_idemp (x₁ :: x₂ :: xs) with x₁ ≤? x₂
swap_idemp (x₁ :: x₂ :: xs) | yes p with x₂ ≤? x₁
... | yes q rewrite antisym p q = P.refl
... | no ¬q = P.refl
swap_idemp (x₁ :: x₂ :: xs) | no ¬p with x₁ ≤? x₂
... | yes q = ⊥-elim (¬p q)
... | no ¬q = P.refl
```

PLE

```
swap_idemp ::
  xs:_ -> {swap (swap xs) = swap xs}

swap_idemp (x1:x2:xs)
  | x1 > x2    = ()
  | otherwise = ()
swap_idemp xs = ()
```

Dafny

```
lemma swap_idempotent2(xs: List<int>)
  ensures swap(swap(xs)) == swap(xs);
  { }
```

Fig. 2. Proofs that swap is idempotent with Coq, Agda, Dafny and PLE.

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.* by 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 Appel [2016]'s Coq textbook:

```
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 trivial in comparison to Coq or Agda. 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. 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*

	Logical Formula	**Refinement Type**
Native Terms	e	$\{ e \}$
Implication	$\phi_1 \Rightarrow \phi_2$	$\phi_1 \rightarrow \phi_2$
Negation	$\neg\phi$	$\phi \rightarrow \{ \texttt{False} \}$
Conjunction	$\phi_1 \wedge \phi_2$	(ϕ_1, ϕ_2)
Disjunction	$\phi_1 \vee \phi_2$	$\texttt{Either}\ \phi_1\ \phi_2$
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 $\{ v : \texttt{Prop} \mid e \}$. Function binders are not relevant for negation and implication, and hence, elided.

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.

3 EMBEDDING NATURAL DEDUCTION WITH REFINEMENT TYPES

In this section we show how user-provided *quantified* specifications can be naturally encoded using λ-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.

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 λ^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 ϕ_1 to the proof of ϕ_2. Negation is encoded as an implication where the consequent is `False`. Conjunction $\phi_1 \wedge \phi_2$ is encoded as the *pair* $(\phi_1,\ \phi_2)$ that contains the proofs of *both* conjuncts and disjunction $\phi_1 \vee \phi_2$ is encoded as the *sum* type `Either` that contains the proofs of *one of* the disjuncts, *i.e.* where `data Either a b = Left a | Right 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 ϕ to be both a proposition and a refinement type. We connect these two meanings of ϕ by using the Propositions as Types [Wadler 2015], to prove that if there exists an expression (or proof term) with refinement type ϕ, then the proposition ϕ 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 "ϕ is provable under the assumptions of Γ". We write $\Gamma \vdash_{ND} \phi$ for Gentzen's natural deduction judgment "under assumption Γ, proposition ϕ holds". Then, each of Gentzen's logical rules can be recovered from the rules in Figure 5 by rewriting each judgment $\Gamma \vdash e : \phi$ of λ^R as $\Gamma \vdash_{ND} \phi$. For example, conjunction and universal elimination can be derived as:

$$\frac{\Gamma \vdash_{ND} \phi_1 \vee \phi_2 \qquad \Gamma, \phi_1 \vdash_{ND} \phi \qquad \Gamma, \phi_2 \vdash_{ND} \phi}{\Gamma \vdash_{ND} \phi} \vee\text{-E} \qquad \frac{\Gamma \vdash_{ND} e_x \text{ term} \qquad \Gamma \vdash_{ND} \forall x.\phi}{\Gamma \vdash_{ND} \phi[x/e_x]} \forall\text{-E}$$

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 ϕ, then there exists an expression that has the refinement type ϕ.

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 can be proved in our system (*i.e.* we can construct an `Either` term e, so that p:{ `Bool` | `True` } $\vdash e : p \vee \neg p$), but cannot be proved using natural deduction (*i.e.* { `True` } $\nvdash_{ND} p \vee \neg p$). The reason for that is that our system is using the classical logic of the SMTs, which includes the law of the excluded middle. On the contrary, in intuitionistic systems that also encode natural deduction (*e.g.* Coq, Idris, NuPRL) the law of the excluded middle 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 ϕ. Read top-down, it constructs a *proof* of the formula as the *term* $\lambda e\ y.$`case` e `of` $\{(x, e_x) \rightarrow (x, e_x\ y)\}$. This proof term corresponds directly to the following Haskell expression that typechecks with type ϕ.

```
exAll :: p:(a → a → Bool) → (x::a, y:a → {p x y}) → y:a → (x::a, {p x y})
exAll _ = \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 ϕ, you can convert between $\{\phi\}$, where ϕ is treated as an SMT-proposition and ϕ, where ϕ is treated as a type; and this conversion goes *both* ways. For example, let $\phi \equiv p \wedge (q \vee r)$. Then `flatten` converts from ϕ 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
```

$$
\cfrac{
  \cfrac{
    \cfrac{
      e{:}\phi_e, y{:}\tau_y \vdash e : \phi_e
      \qquad
      \cfrac{\begin{array}{c} e{:}\phi_e, y{:}\tau_y, x{:}t_x, e_x{:}\phi_x \vdash e_x : \phi_x \\ e{:}\phi_e, y{:}\tau_y, x{:}t_x, e_x{:}\phi_x \vdash y : \tau_y \end{array}}{e{:}\phi_e, y{:}\tau_y, x{:}t_x, e_x{:}\phi_x \vdash e_x\ y : p\ x\ y}\ \forall\text{-E}
    }{e{:}\phi_e, y{:}\tau_y \vdash \mathtt{case}\ e\ \mathtt{of}\ \{(x, e_x) \to (x, e_x\ y)\} : \exists x.(p\ x\ y)}\ \exists\text{-E}
  }{e{:}\phi_e \vdash \lambda y.\mathtt{case}\ e\ \mathtt{of}\ \{(x, e_x) \to (x, e_x\ y)\} : \forall y.\exists x.(p\ x\ y)}\ \forall\text{-I}
}{\emptyset \vdash \lambda e\ y.\mathtt{case}\ e\ \mathtt{of}\ \{(x, e_x) \to (x, e_x\ y)\} : (\exists x.\forall y.(p\ x\ y)) \Rightarrow (\forall y.\exists x.(p\ x\ y))}\ \Rightarrow\text{-I}
$$

Fig. 4. Proof of $(\exists x.\forall y.(p\ x\ y)) \Rightarrow (\forall y.\exists x.(p\ x\ y))$ where $\phi_e \equiv \exists x.\forall y.(p\ x\ y)$, $\phi_x \equiv \forall y.(p\ x\ y)$.

```
expand :: p:_ → q:_ → r:_ → {p && (q || r)} → ({p}, Either {q} {r})
expand proof | q = (proof, Left  proof)
expand proof | r = (proof, Right proof)
```

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.*

$$
\begin{aligned}
\phi_\exists &\equiv (\exists x.p\ x \lor q\ x) \Rightarrow ((\exists x.p\ x) \lor (\exists x.q\ x)) && (1)\\
\phi_\forall &\equiv (\forall x.p\ x \land q\ x) \Rightarrow ((\forall x.p\ x) \land (\forall x.q\ x)) && (2)
\end{aligned}
$$

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 λ-abstraction and case spitting inside the conjunction pair:

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

The above proof term exactly corresponds to its natural deduction proof derivation but using SMT-aided verification can get simplified to the following

```
allDistAnd _ _ andx = (pf, pf)
  where pf x = case andx x of (px, py) → px &&& py
```

Properties of User Defined Datatypes As ϕ 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.

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

The proof (`evenLen`) proceeds by existential elimination and introduction and uses the `lenAppend` lemma, which in turn uses induction on the input list and PLE to automate equational reasoning.

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

$$\frac{\Gamma \vdash \texttt{fst}\ e : \phi_1 \quad \Gamma \vdash \texttt{snd}\ e : \phi_2}{\Gamma \vdash e : (\phi_1, \phi_2)}\ \wedge\text{-I} \qquad \frac{\Gamma \vdash e : (\phi_1, \phi_2)}{\Gamma \vdash \texttt{fst}\ e : \phi_1}\ \wedge\text{-L-E} \qquad \frac{\Gamma \vdash e : (\phi_1, \phi_2)}{\Gamma \vdash \texttt{snd}\ e : \phi_2}\ \wedge\text{-R-E}$$

$$\frac{\Gamma \vdash e_1 : \phi_1}{\Gamma \vdash \texttt{Left}\ e_1 : \texttt{Either}\ \phi_1\ \phi_2}\ \vee\text{-L-I} \qquad \frac{\Gamma \vdash e_1 : \phi_2}{\Gamma \vdash \texttt{Right}\ e_2 : \texttt{Either}\ \phi_1\ \phi_2}\ \vee\text{-R-I}$$

$$\frac{\Gamma \vdash e : \texttt{Either}\ \phi_1\ \phi_2 \quad \Gamma, x_1{:}\phi_1 \vdash e_1 : \phi \quad \Gamma, x_2{:}\phi_2 \vdash e_2 : \phi}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{\texttt{Left}\ x_1 \rightarrow e_1;\ \texttt{Right}\ x_2 \rightarrow e_2\} : \phi}\ \vee\text{-E}$$

$$\frac{\Gamma, x{:}\phi_x \vdash e : \phi}{\Gamma \vdash \lambda x.e : \phi_x \rightarrow \phi}\ \Rightarrow\text{-I} \qquad \frac{\Gamma \vdash e : \phi_x \rightarrow \phi \quad \Gamma \vdash e_x : \phi_x}{\Gamma \vdash e\ e_x : \phi}\ \Rightarrow\text{-E}$$

$$\frac{\Gamma, x{:}\tau \vdash e : \phi}{\Gamma \vdash \lambda x.e : (x : \tau \rightarrow \phi)}\ \forall\text{-I} \qquad \frac{\Gamma \vdash e_x : \tau \quad \Gamma \vdash e : (x : \tau \rightarrow \phi)}{\Gamma \vdash e\ e_x : \phi[x/e_x]}\ \forall\text{-E}$$

$$\frac{\Gamma \vdash \texttt{fst}\ e : \tau \quad \Gamma, x{:}\tau \vdash \texttt{snd}\ e : \phi}{\Gamma \vdash e : (x :: \tau, \phi[x/\texttt{fst}\ e])}\ \exists\text{I} \qquad \frac{\Gamma \vdash e : (x :: \tau, \phi_x) \quad \Gamma, x{:}\tau, y{:}\phi_x \vdash e' : \phi}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{(x, y) \rightarrow e'\} : \phi}\ \exists\text{E}$$

Fig. 5. Natural deduction rules for refinement types. With $[\texttt{fst}|\texttt{snd}]\ e \equiv \texttt{case}\ e\ \texttt{of}\ \{(x_1, x_2) \rightarrow [x_1|x_2]\}$.

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

Induction on Natural Numbers Finally, we specify and verify *induction* on natural numbers:

$$\phi_{ind} \equiv (p\ 0 \wedge (\forall n.p\ (n-1) \Rightarrow p\ n) \Rightarrow \forall n.p\ n)$$

The proof proceeds by induction (*e.g.* case splitting). In the base case, `n == 0`, the proof calls the left conjunct, which contains a proof of the base case. Otherwise, `0 < n`, and the proof *applies* the induction hypothesis to the right conjunct instantiated at `n-1`.

```
ind :: p:_ → ({p 0},(n:Nat → {p (n-1)} → {p n})) → n:Nat → {p n}
ind p (p0, pn) 0 = p0
ind p (p0, pn) n = pn n (ind p (p0, pn) (n-1))
```

3.4 Consequences

To summarize, we use the propositions-as-types principle to make two important contributions. First, we show that natural deduction reasoning can smoothly co-exist with SMT-based verification to automate the decidable, quantifier-free portions of the proof.

Second, we show for the first time how natural deduction proofs can be encoded in refinement type systems like Liquid Haskell and we expect this encoding to extend, in a straightforward manner to other SMT-based deductive verifiers (*e.g.* Dafny and F*). This encoding shows that refinement type systems are expressive enough to encode any intuitionistic natural deduction proof, gives a guideline for encoding proofs with nested quantifiers, and provides a pleasant implementation of natural deduction that is pedagogically useful.

4 REFINEMENT REFLECTION: λ^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 λ^R with an *undecidable* type system based

$$
\begin{array}{rrcl}
\textit{\textbf{Ops}} & \odot & ::= & = \mid < \\
\textit{\textbf{Consts}} & c & ::= & \wedge \mid ! \mid \odot \mid +, -, \ldots \\
 & & \mid & \texttt{True} \mid \texttt{False} \mid 0, -1, 1, \ldots \\
\textit{\textbf{Vals}} & w & ::= & c \mid \lambda x.e \mid D\,\overline{w} \\
\textit{\textbf{Exprs}} & e & ::= & w \mid x \mid e\ e \\
 & & \mid & \texttt{case}\ x = e\ \texttt{of}\ \{D\,\overline{x} \to e\} \\
\textit{\textbf{Binds}} & b & ::= & e \mid \texttt{let rec}\ x : \tau = b\ \texttt{in}\ b \\
\textit{\textbf{Progs}} & p & ::= & b \mid \texttt{reflect}\ x : \tau = e\ \texttt{in}\ p \\
\textit{\textbf{Bas. Types}} & B & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \\
\textit{\textbf{Ref. Types}} & \tau & ::= & \{v : B^{[\Downarrow]} \mid e\} \mid x : \tau_x \to \tau
\end{array}
\qquad
\begin{array}{rrcl}
\textit{\textbf{Terms}} & p & ::= & p \bowtie p \mid \oplus_1 p \mid D \\
 & & \mid & n \mid b \mid x \mid x\,\overline{p} \\
 & & \mid & \texttt{if}\ p\ \texttt{then}\ p\ \texttt{else}\ p \\
\textit{\textbf{Ints}} & n & ::= & 0, -1, 1, \ldots \\
\textit{\textbf{Bools}} & b & ::= & \texttt{True} \mid \texttt{False} \\
\textit{\textbf{Bin-Ops}} & \bowtie & ::= & = \mid < \mid \wedge \mid +, -, \ldots \\
\textit{\textbf{Un-Ops}} & \oplus_1 & ::= & ! \mid \ldots \\
\textit{\textbf{Args}} & s_a & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{U} \\
 & & \mid & \texttt{Fun}\ s_a\ s_a \\
\textit{\textbf{Sorts}} & s & ::= & s_a \mid s_a \to s
\end{array}
$$

Fig. 6. **(L)** Syntax of λ^R: Denotational Typing. **(R)** Syntax of λ^S: Algorithmic Typing.

on denotational semantics and prove it sound. Next, in § 5 we define a language λ^S that soundly approximates λ^R while enabling decidable, SMT-based type checking. Finally, in § 6 we develop a complete proof search algorithm to automate equational reasoning.

4.1 Syntax

Figure 6 summarizes the syntax of λ^R, which is essentially the calculus λ^U [Vazou et al. 2014] with explicit recursion and a special `reflect` binding to denote terms that are reflected into the refinement logic. The elements of λ^R are constants, values, expressions, binders, and programs.

Constants The constants of λ^R include primitive relations $\odot$, here, the set $\{=, <\}$. Moreover, they include the booleans `True`, `False`, integers `-1`, `0`, `1`, *etc.*, and logical operators $\wedge$, `!`, *etc.*.

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

Values & Expressions The values of λ^R include constants, λ-abstractions $\lambda x.e$, and fully applied data constructors D that wrap values. The expressions of λ^R include values, variables x, applications $e\ e$, and `case` expressions.

Binders & Programs A *binder* b is a series of possibly recursive `let` definitions, followed by an expression. A *program* p is a series of `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 `let` binders in expressions e, but omit them for simplicity.)

4.2 Operational Semantics

We define $\hookrightarrow$ to be the small step, call-by-name β-reduction semantics for λ^R. We evaluate reflected terms as recursive `let` bindings, with termination constraints imposed by the type system:

$$\texttt{reflect}\ x : \tau = e\ \texttt{in}\ p \hookrightarrow \texttt{let rec}\ x : \tau = e\ \texttt{in}\ p$$

We define $\hookrightarrow^\star$ to be the reflexive, transitive closure of $\hookrightarrow$. Moreover, we define $\approx_\beta$ to be the reflexive, symmetric, and transitive closure of $\hookrightarrow$.

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, *i.e.* $c\ v \hookrightarrow \delta(c, v)$. For example, consider $=$, the primitive equality operator on integers. We have $\delta(=, n) \doteq =_n$ where $\delta(=_n, m)$ equals `True` iff m is the same as n.

Equality We assume that the equality operator is defined *for all* values, and, for functions, is defined as extensional equality. That is, for all f and f', $(f = f') \hookrightarrow$ `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

λ^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 $[Int]$ 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 $e{:}\{v : B^{\Downarrow} \mid e'\}$, then e terminates. As discussed by Vazou et al. [2014] termination labels are 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 checks for exhaustive definitions (§ 2.1), it ensures totality (well-formedness) of expressions as required by propositions-as-types (§ 3) and termination of PLE (§ 6). Finally, we have dependent *function types* $x{:}\tau_x \to \tau$ where the input x has the type τ_x and the output τ may refer to the input binder x. We write B to abbreviate $\{v : B \mid \texttt{True}\}$ and $\tau_x \to \tau$ to abbreviate $x{:}\tau_x \to \tau$, if x does not appear in τ.

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

$$
\begin{aligned}
[\![\{x : B \mid r\}]\!] &\doteq \{e \mid e : B,\ \text{if } e \hookrightarrow^\star w \text{ then } r[x/w] \hookrightarrow^\star \texttt{True}\}\\
[\![\{x : B^{\Downarrow} \mid r\}]\!] &\doteq [\![\{x : B \mid r\}]\!] \cap \{e \mid \exists\, w.e \hookrightarrow^\star w\}\\
[\![x{:}\tau_x \to \tau]\!] &\doteq \{e \mid e : \mathsf{shape}(\tau_x \to \tau), \forall e_x \in [\![\tau_x]\!].\ (e\ e_x) \in [\![\tau[x/e_x]]\!]\}
\end{aligned}
$$

Constants For each constant c we define its type $\mathsf{prim}(c)$ such that $c \in [\![\mathsf{prim}(c)]\!]$. For example,

$$
\begin{aligned}
\mathsf{prim}(3) &\doteq \{v : \texttt{Int}^{\Downarrow} \mid v = 3\}\\
\mathsf{prim}(+) &\doteq \texttt{x}{:}\texttt{Int}^{\Downarrow} \to \texttt{y}{:}\texttt{Int}^{\Downarrow} \to \{v : \texttt{Int}^{\Downarrow} \mid v = x + y\}\\
\mathsf{prim}(\leq) &\doteq \texttt{x}{:}\texttt{Int}^{\Downarrow} \to \texttt{y}{:}\texttt{Int}^{\Downarrow} \to \{v : \texttt{Bool}^{\Downarrow} \mid v \Leftrightarrow x \leq y\}
\end{aligned}
$$

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 (`reflect` $f : \tau = e$ `in` p) as a `let rec`-binder (`let rec` f : $\mathsf{Reflect}(\tau, e) = e$ `in` p) during type checking (rule T-Refl in Figure 7).

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

$$
\begin{aligned}
\mathsf{Reflect}(\{v : B^{\Downarrow} \mid r\}, e) &\doteq \{v : B^{\Downarrow} \mid r \wedge v = e\}\\
\mathsf{Reflect}(x{:}\tau_x \to \tau, \lambda x.e) &\doteq x{:}\tau_x \to \mathsf{Reflect}(\tau, e)
\end{aligned}
$$

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

$$\begin{aligned} t_{\texttt{fib}} &\doteq \{v : \texttt{Int}^{\Downarrow} \mid 0 \leq v\} \rightarrow \{v : \texttt{Int}^{\Downarrow} \mid 0 \leq v\} \\ e_{\texttt{fib}} &\doteq \texttt{case } x = n \leq 1 \texttt{ of } \{\texttt{True} \rightarrow n; \texttt{False} \rightarrow \texttt{fib}(n-1) + \texttt{fib}(n-2)\} \end{aligned}$$

Then, the reflected type of `fib` will be:

$$\text{Reflect}(t_{\texttt{fib}}, e_{\texttt{fib}}) = n{:}\{v : \texttt{Int}^{\Downarrow} \mid 0 \leq v\} \rightarrow \{v : \texttt{Int}^{\Downarrow} \mid 0 \leq v \land v = e_{\texttt{fib}}\}$$

Termination Checking We defined 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 λ^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 x = 1 + f x` into the logic leads to an inconsistent system that is able to prove `0 = 1`.

Automatic Reflection Reflection of λ^R expressions into the refinements happens automatically by the type system, not manually by the user. The user simply annotates a function f as `reflect` f. Then, the rule T-Refl in Figure 7 is used to type check the reflected function by strengthening f's result via Reflect$(\cdot, \cdot)$. Finally, the rule 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 λ^R, as found in Figure 7.

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

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

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:

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

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

Rules All but two of the rules are standard [Knowles and Flanagan 2010; Vazou et al. 2014] except for the addition of the reflection environment R at each rule. First, rule 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 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 `fib 1` is typed as $\{v : \texttt{Int}^{\Downarrow} \mid \texttt{fibP}\ 1 \land v = \texttt{fib}\ \ 1\}$ where the first conjunct comes from the (reflection-strengthened) output refinement of `fib` (§ 2) and the second comes from rule T-Exact.

Typing $\boxed{\Gamma; R \vdash p : \tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma; R \vdash x : \tau}\ \textsc{T-Var} \qquad \frac{}{\Gamma; R \vdash c : \mathsf{prim}(c)}\ \textsc{T-Con} \qquad \frac{\Gamma; R \vdash p : \tau' \quad \Gamma; R \vdash \tau' \preceq \tau}{\Gamma; R \vdash p : \tau}\ \textsc{T-Sub}$$

$$\frac{\Gamma; R \vdash e : \{v : B \mid e_r\}}{\Gamma; R \vdash e : \{v : B \mid e_r \wedge v = e\}}\ \textsc{T-Exact} \qquad \frac{\Gamma, x : \tau_x; R \vdash e : \tau}{\Gamma; R \vdash \lambda x.e : (x{:}\tau_x \to \tau)}\ \textsc{T-Fun}$$

$$\frac{\Gamma; R \vdash e : (x{:}\tau_x \to \tau) \quad \Gamma; R \vdash e_x : \tau_x}{\Gamma; R \vdash e\ e_x : \tau[x/e_x]}\ \textsc{T-App} \qquad \frac{\begin{array}{cc}\Gamma, x : \tau_x; R \vdash b_x : \tau_x & \Gamma, x : \tau_x \vdash \tau_x \\ \Gamma, x : \tau_x; R \vdash b : \tau & \Gamma \vdash \tau\end{array}}{\Gamma; R \vdash \mathtt{let\ rec}\ x : \tau_x = b_x\ \mathtt{in}\ b : \tau}\ \textsc{T-Let}$$

$$\frac{\begin{array}{cc}\Gamma; R \vdash e : \{v : T \mid e_r\} & \Gamma \vdash \tau \\ \forall i.\mathsf{prim}(D_i) = \overline{y_j : \tau_j} \to \{v : T \mid e_{r_i}\} & \Gamma, \overline{y_j : \tau_j}, x : \{v : T \mid e_r \wedge e_{r_i}\}; R \vdash e_i : \tau\end{array}}{\Gamma; R \vdash \mathtt{case}\ x = e\ \mathtt{of}\ \{D_i\ \overline{y_i} \to e_i\} : \tau}\ \textsc{T-Case}$$

$$\frac{\Gamma; R, f \mapsto e \vdash \mathtt{let\ rec}\ f : \mathsf{Reflect}(\tau_f, e) = e\ \mathtt{in}\ p : \tau}{\Gamma; R \vdash \mathtt{reflect}\ f : \tau_f = e\ \mathtt{in}\ p : \tau}\ \textsc{T-Refl}$$

Well Formedness $\boxed{\Gamma \vdash \tau}$

$$\frac{\Gamma, v : B; \emptyset \vdash e : \mathsf{Bool}^{\Downarrow}}{\Gamma \vdash \{v : B \mid e\}}\ \textsc{WF-Base} \qquad \frac{\Gamma \vdash \tau_x \quad \Gamma, x : \tau_x \vdash \tau}{\Gamma \vdash x{:}\tau_x \to \tau}\ \textsc{WF-Fun}$$

Subtyping $\boxed{\Gamma; R \vdash \tau_1 \preceq \tau_2}$

$$\frac{\forall \theta \in [\![\Gamma]\!].[\![\theta \cdot \{v : B \mid e_1\}]\!] \subseteq [\![\theta \cdot \{v : B \mid e_2\}]\!]}{\Gamma; R \vdash \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}}\ \preceq\textsc{-Base-}\lambda^R$$

$$\frac{\Gamma; R \vdash \tau_x' \preceq \tau_x \quad \Gamma, x : \tau_x'; R \vdash \tau \preceq \tau'}{\Gamma; R \vdash x{:}\tau_x \to \tau \preceq x{:}\tau_x' \to \tau'}\ \preceq\textsc{-Fun}$$

Fig. 7. Typing of λ^R.

Well-formedness A judgment $\Gamma \vdash \tau$ states that the refinement type τ is well-formed in the environment Γ. Following Vazou et al. [2014], τ is well-formed if all the refinements in τ are `Bool`-typed, provably terminating expressions in Γ.

Subtyping A judgment $\Gamma; R \vdash \tau_1 \preceq \tau_2$ states that the type τ_1 is a subtype of τ_2 in the environments Γ and R. Informally, τ_1 is a subtype of τ_2 if, when the free variables of τ_1 and τ_2 are bound to expressions described by Γ, the denotation of τ_1 is *contained in* the denotation of τ_2. Subtyping of basic types reduces to denotational containment checking, shown in rule $\preceq$-Base-λ^R. That is, τ_1 is a subtype of τ_2 under Γ if for any closing substitution θ in $[\![\Gamma]\!]$, $[\![\theta \cdot \tau_1]\!]$ is contained in $[\![\theta \cdot \tau_2]\!]$.

Soundness We prove that typing implies denotational inclusion and evaluation preserves typing.

Theorem 4.1. *[Soundness of λ^R]*

- ***Denotations*** *If* $\Gamma; R \vdash p : \tau$, *then* $\forall \theta \in [\![\Gamma]\!].\theta \cdot p \in [\![\theta \cdot \tau]\!]$.
- ***Preservation*** *If* $\emptyset; \emptyset \vdash p : \tau$ *and* $p \hookrightarrow^{\star} w$, *then* $\emptyset; \emptyset \vdash w : \tau$.

The proofs can be found in [Vazou et al. 2017b]. Theorem 4.1 lets us prove that if ϕ is a λ^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 ϕ is valid. For example, in § 2 we verified that the term `fibUp` proves n:`Nat` $\rightarrow \{\texttt{fib}\ n \leq \texttt{fib}\ (n+1)\}$. Via soundness of λ^R, we get that for each valid input `n`, the result refinement is valid.

$$\forall n. 0 \leq n \hookrightarrow^\star \texttt{True} \Rightarrow \texttt{fib}\ n \leq \texttt{fib}\ (n+1) \hookrightarrow^\star \texttt{True}$$

5 ALGORITHMIC CHECKING: λ^S

λ^S is a first order approximation of λ^R where higher-order features are approximated with uninterpreted functions and the undecidable type subsumption rule $\preceq$-Base-λ^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 λ^S, the *sorted* (SMT-) decidable logic of quantifier-free equality, uninterpreted functions and linear arithmetic (QF-EUFLIA) [Barrett et al. 2010; Nelson 1980]. The *terms* of λ^S include integers n, booleans b, variables x, data constructors D (encoded as constants), fully applied unary $\oplus_1$ and binary $\bowtie$ operators, and application $x\ \overline{p}$ of an uninterpreted function x. The *sorts* of λ^S include the built-in `Int` and `Bool` to represent integers and booleans and the uninterpreted `U` to represent data types. The interpreted functions of λ^S, *e.g.* the logical constants $=$ and $<$, have the function sort $s \rightarrow s$. Other functional values in λ^R, *e.g.* reflected λ^R functions and λ-expressions, are represented as first-order values with the uninterpreted sort `Fun` $s\ s$.

5.1 Transforming λ^R into λ^S

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

Embedding Types We embed λ^R types into λ^S sorts as:

$$\begin{array}{llllll}
\lfloor \texttt{Int} \rfloor & \doteq \texttt{Int} & \lfloor T \rfloor & \doteq \texttt{U} & \lfloor \{v : B^{[\Downarrow]} \mid e\} \rfloor & \doteq \lfloor B \rfloor \\
\lfloor \texttt{Bool} \rfloor & \doteq \texttt{Bool} & & & \lfloor x : \tau_x \rightarrow \tau \rfloor & \doteq \texttt{Fun}\ \lfloor \tau_x \rfloor\ \lfloor \tau \rfloor
\end{array}$$

Embedding Constants Elements shared on both λ^R and λ^S translate to themselves. These elements include booleans, integers, variables, binary and unary operators. SMT solvers do not support currying, and so in λ^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 `(\z → z + 1)`.

Embedding Functions As λ^S is first-order, we embed λs using the uninterpreted function `lam`.

$$\frac{\Gamma, x : \tau_x \vdash e \rightsquigarrow p \qquad \Gamma; \emptyset \vdash (\lambda x.e) : (x : \tau_x \rightarrow \tau)}{\Gamma \vdash \lambda x.e \rightsquigarrow \texttt{lam}_{\lfloor \tau \rfloor}^{\lfloor \tau_x \rfloor}\ x\ p}$$

The term $\lambda x.e$ of type $\tau_x \rightarrow \tau$ is transformed to $\texttt{lam}_s^{s_x}\ x\ p$ of sort `Fun` $s_x\ s$, where s_x and s are respectively $\lfloor \tau_x \rfloor$ and $\lfloor \tau \rfloor$, $\texttt{lam}_s^{s_x}$ is a special uninterpreted function of sort $s_x \rightarrow s \rightarrow \texttt{Fun}\ s_x\ s$, and x of sort s_x and p of sort s are the embeddings of the binder and body, respectively. As `lam` is an SMT-function, it *does not* create a binding for x. Instead, x is renamed to a *fresh* SMT name.

Embedding Applications We use defunctionalization [Reynolds 1972] to embed applications.

$$\frac{\Gamma \vdash e_x \rightsquigarrow p_x \qquad \Gamma \vdash e \rightsquigarrow p \qquad \Gamma; \emptyset \vdash e : \tau_x \to \tau}{\Gamma \vdash e\ e_x \rightsquigarrow \mathsf{app}_{\lfloor \tau \rfloor}^{\lfloor \tau_x \rfloor}\ p\ p_x}$$

The term $e\ e_x$, where e and e_x have types $\tau_x \to \tau$ and τ_x, is transformed to $\mathsf{app}_s^{s_x}\ p\ p_x : s$ where s and s_x are $\lfloor \tau \rfloor$ and $\lfloor \tau_x \rfloor$, the $\mathsf{app}_s^{s_x}$ is a special uninterpreted function of sort $\mathsf{Fun}\ s_x\ s \to s_x \to s$, and p and p_x are the respective translations of e and e_x.

Embedding Data Types We embed data constructors to a predefined λ^S constant s_D of sort $\lfloor \mathsf{prim}(D) \rfloor$: $\Gamma \vdash D \rightsquigarrow \mathsf{s}_D$. For each datatype, we create reflected functions that *check* the top-level constructor and *select* their individual fields. For example, for lists, we create the functions

```
isNil []     = True         isCons (x:xs) = True         sel1 (x:xs) = x
isNil (x:xs) = False        isCons []     = False        sel2 (x:xs) = xs
```

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 $\mathsf{prim}(D_i) \doteq \tau_{i,1} \to \cdots \to \tau_{i,n} \to \tau$. Then *check* $\mathtt{is}_{D_i}$ has the sort $\mathsf{Fun}\ \lfloor \tau \rfloor\ \mathsf{Bool}$ and *select* $\mathtt{sel}_{D_{i,j}}$ has the sort $\mathsf{Fun}\ \lfloor \tau \rfloor\ \lfloor \tau_{i,j} \rfloor$.

Embedding Case Expressions We translate case-expressions of λ^R into nested `if` terms in λ^S, by using the check functions in the guards and the select functions for the binders of each case.

$$\frac{\Gamma \vdash e \rightsquigarrow p \qquad \Gamma \vdash e_i[\overline{y_i}/\overline{\mathtt{sel}_{D_i}\ x}][x/e] \rightsquigarrow p_i}{\Gamma \vdash \mathtt{case}\ x = e\ \mathtt{of}\ \{D_i\ \overline{y_i} \to e_i\} \rightsquigarrow \mathtt{if}\ \mathtt{app}\ \mathtt{is}_{D_1}\ p\ \mathtt{then}\ p_1\ \mathtt{else}\ \ldots\ \mathtt{else}\ p_n}$$

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 θ and every logical model σ that respects the environment Γ if $\theta \cdot e \hookrightarrow^\star v$ then $\sigma \models p = \lfloor v \rfloor$.

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 $\lfloor \cdot \rfloor$ as the translation from λ^R to λ^S. Informally, the implication or *verification condition* (VC) $\lfloor \Gamma \rfloor \Rightarrow p_1 \Rightarrow p_2$ 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 Γ. Γ is embedded into logic by conjoining the refinements of terminating binders [Vazou et al. 2014]:

$$\lfloor \Gamma \rfloor \ \doteq\ \bigcup_{x \in \Gamma} \lfloor \Gamma, x \rfloor \quad \text{where we embed each binder as} \quad \lfloor \Gamma, x \rfloor \ \doteq\ \begin{cases} \lfloor e \rfloor & \text{if } \Gamma(x) = \{x : B^{\Downarrow} \mid e\} \\ \mathsf{True} & \text{otherwise.} \end{cases}$$

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

Subtyping via VC Validity Checking We make subtyping, and hence, typing decidable, by replacing the denotational base subtyping rule $\preceq$-Base-λ^R with the conservative, algorithmic

$$\begin{array}{rrcl}
\textbf{\textit{Terms}} & p, t, b & ::= & \lambda^S \; \texttt{if}\text{-free terms from Figure 6} \\
\textbf{\textit{Functions}} & F & ::= & \lambda \overline{x}.\langle \overline{p \Rightarrow b} \rangle \\
\textbf{\textit{Definitional Environment}} & \Psi & ::= & \emptyset \mid f \mapsto F, \Psi \\
\textbf{\textit{Logical Environment}} & \Phi & ::= & \emptyset \mid p, \Phi
\end{array}$$

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

version $\preceq$-Base-PLE that uses PLE to check the validity of the subtyping.

$$\frac{\mathsf{PLE}(\lfloor R \rfloor, \lfloor \Gamma, v : \{v : B^{\Downarrow} \mid e_1\} \rfloor, \lfloor e_2 \rfloor)}{\Gamma; R \vdash_{PLE} \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}} \preceq\text{-BASE-PLE}$$

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

Lemma 5.1. *If* $\Gamma; R \vdash_{PLE} \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}$ *then* $\Gamma; R \vdash \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}$.

Soundness of λ^S We write $\Gamma; R \vdash_{\mathsf{PLE}} e : \tau$ for the judgments that can be derived by the algorithmic subtyping rule $\preceq$-Base-PLE (instead of $\preceq$-Base-λ^R.) Lemma 5.1 implies the soundness of λ^S.

Theorem 5.2 (Soundness of λ^S). *If* $\Gamma; R \vdash_{\mathsf{PLE}} e : \tau$, *then* $\Gamma; R \vdash e : \tau$.

6 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 Φ contains a set of hypotheses p, described in Figure 6. The definitional environment Ψ maps function symbols f to their definitions $\lambda \overline{x}.\langle \overline{p \Rightarrow b} \rangle$, written as λ-abstractions over guarded bodies. Moreover, the body b and the guard p contain neither λ nor `if`. These restrictions do not impact expressiveness: λs can be named and reflected, and `if`-expressions can be pulled out into top-level guards using $\mathtt{DeIf}(\cdot)$, defined in [Vazou et al. 2017b]. A definitional environment Ψ can be constructed from R as

$$\lfloor R \rfloor \doteq \{f \mapsto \lambda \overline{x}.\mathtt{DeIf}(\lfloor e \rfloor) | (f \mapsto \lambda \overline{x}.e) \in R\}$$

Notation We write $f\ (t_1, \ldots, t_n) \prec \Phi$ if the λ^S term $(\mathsf{app} \ldots (\mathsf{app}\ f\ t_1) \ldots t_n)$ 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 $\mathsf{SmtValid}(\Phi, p)$ for SMT validity of the implication $\Phi \Rightarrow p$.

Instantiation & Unfolding A term q is a (Ψ, Φ)-*instance* if there exists $f(\overline{t}) \prec \Phi$ such that:

- $\Psi(f) \equiv \lambda \overline{x}.\langle \overline{p_i \Rightarrow b_i} \rangle$,
- $\mathsf{SmtValid}(\Phi, p_i \left[\overline{t}/\overline{x}\right])$, and
- $q \equiv (f(\overline{x}) = b_i) \left[\overline{t}/\overline{x}\right]$.

A set of terms Q is a (Ψ, Φ)-*instance* if every $q \in Q$ is an (Ψ, Φ)-instance. The *unfolding* of Ψ, Φ is the (finite) set of all (Ψ, Φ)-instances. Procedure $\mathsf{Unfold}(\Psi,\ \Phi)$ shown in Figure 9 computes and returns the conjunction of Φ and the unfolding of Ψ, Φ. The following properties relate (Ψ, Φ)-instances to the semantics of λ^R and SMT validity. Let $R[e]$ denote the evaluation of e under the reflection environment R, *i.e.* $\emptyset[e] \doteq e$ and $(R, f : e_f)[e] \doteq R[\text{let rec } f = e_f \text{ in } e]$.

Lemma 6.1. *For every* $\Gamma \models R$ *and* $\theta \in (\!|\Gamma|\!)$,

$$
\begin{array}{lcl}
\hline
\mathsf{Unfold} & : & (\Psi, \Phi) \to \Phi \\
\hline
\mathsf{Unfold}(\Psi,\ \Phi) & = & \Phi \cup \bigcup_{f(\overline{t}) \prec \Phi} \mathsf{Instantiate}\left(\Psi, \Phi, f, \overline{t}\right) \\
\mathsf{Instantiate}\left(\Psi, \Phi, f, \overline{t}\right) & = & \left\{ (\lfloor f(\overline{x}) \rfloor = b_i) \left[\overline{t}/\overline{x}\right] \;\middle|\; (p_i \Rightarrow b_i) \in \overline{d},\ \ \mathsf{SmtValid}(\Phi, p_i \left[\overline{t}/\overline{x}\right]) \right\} \\
\quad \text{where} & & \\
\qquad \lambda \overline{x}.\langle \overline{d} \rangle & = & \Psi(f) \\
\hline
\mathsf{PLE} & : & (\Psi, \Phi, p) \to \mathtt{Bool} \\
\hline
\mathsf{PLE}(\Psi, \Phi, p) & = & \mathsf{loop}\left(0,\ \Phi \cup \bigcup_{f(\overline{t}) \prec p} \mathsf{Instantiate}\left(\Psi, \Phi, f, \overline{t}\right)\right) \\
\quad \text{where} & & \\
\qquad \mathsf{loop}\,(i, \Phi_i) & & \\
\qquad \mid\ \mathsf{SmtValid}(\Phi_i, p) & = & \mathit{true} \\
\qquad \mid\ \Phi_{i+1} \subseteq \Phi_i & = & \mathit{false} \\
\qquad \mid\ \text{otherwise} & = & \mathsf{loop}\,(i+1, \Phi_{i+1}) \\
\qquad\quad \text{where} & & \\
\qquad\qquad \Phi_{i+1} & = & \Phi \cup \mathsf{Unfold}(\Psi,\ \Phi_i) \\
\hline
\end{array}
$$

Fig. 9. **Algorithm** PLE: Proof by Logical Evaluation.

- ***Sat-Inst*** *If* $\lfloor e \rfloor$ *is a* $(\lfloor R \rfloor, \lfloor \Gamma \rfloor)$*-instance, then* $\theta \cdot R[e] \hookrightarrow^\star \mathtt{True}$.
- ***SMT-Approx*** *If* $\mathsf{SmtValid}(\lfloor \Gamma \rfloor, \lfloor e \rfloor)$*, then* $\theta \cdot R[e] \hookrightarrow^\star \mathtt{True}$.
- ***SMT-Inst*** *If* q *is a* $(\lfloor R \rfloor, \lfloor \Gamma \rfloor)$*-instance and* $\mathsf{SmtValid}(\lfloor \Gamma \rfloor \cup \{q\}, \lfloor e \rfloor)$*, then* $\theta \cdot R[e] \hookrightarrow^\star \mathtt{True}$.

The Algorithm Figure 9 shows our proof search algorithm $\mathsf{PLE}(\Psi, \Phi, p)$ which takes as input a set of *reflected definitions* Ψ, an *hypothesis* Φ, 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 *true*) *or* no new instances are generated by the unfolding (in which case the search returns *false*).

Soundness First, we prove the soundness of PLE.

Theorem 6.2 (**Soundness**). *If* $\mathsf{PLE}(\lfloor R \rfloor, \lfloor \Gamma \rfloor, \lfloor e \rfloor)$ *then* $\forall \theta \in (\!|\Gamma|\!),\ \theta \cdot R[e] \hookrightarrow^\star \mathtt{True}$.

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 $\Psi, \Phi \vdash t \twoheadrightarrow t'$ and prove that if there exists such a proof, then $\mathsf{PLE}(\Psi, \Phi, t = t')$ is guaranteed to return *true*. To prove this theorem, we introduce the notion of *bounded unfolding* which corresponds to unfolding definitions n times. We show that unfolding preserves congruences, and hence, that an equational proof exists *iff* 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 show that the fixpoint is computable: there exists an unfolding depth at which PLE reaches a fixpoint and hence terminates.

$$\frac{}{\Psi, \Phi \vdash t \twoheadrightarrow t} \ \textsc{Eq-Refl}$$

$$\frac{\Psi, \Phi \vdash t \twoheadrightarrow t'' \quad \Phi' = \mathsf{Unfold}(\Psi,\ \Phi \cup \{v = t''\}) \quad \mathsf{SmtValid}(\Phi', v = t')}{\Psi, \Phi \vdash t \twoheadrightarrow t'} \ \textsc{Eq-Trans}$$

$$\frac{\Psi, \Phi \vdash t_1 \twoheadrightarrow t_1' \quad \Psi, \Phi \vdash t_2 \twoheadrightarrow t_2' \quad \mathsf{SmtValid}(\Phi, t_1' \bowtie t_2')}{\Psi, \Phi \vdash t_1 \bowtie t_2} \ \textsc{Eq-Proof}$$

Fig. 10. **Equational Proofs:** rules for equational reasoning.

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

$$\begin{array}{lcl} \mathsf{Unfold}^*(\Psi, \Phi, 0) & \doteq & \Phi \\ \mathsf{Unfold}^*(\Psi, \Phi, n+1) & \doteq & \Phi_n \cup \mathsf{Unfold}(\Psi,\ \Phi_n) \quad \text{where } \Phi_n = \mathsf{Unfold}^*(\Psi, \Phi, n) \end{array}$$

That is, the unfolding at depth n essentially performs Unfold upto 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**). $\forall 0 \leq n.\ \mathsf{Unfold}^*(\Psi, \Phi, n) \subseteq \mathsf{Unfold}^*(\Psi, \Phi, n+1)$.

Lemma 6.4 (**Fixpoint**). *Let* $\Phi_i \doteq \mathsf{Unfold}^*(\Psi, \Phi, i)$*. 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.

Lemma 6.5 (**Uncovering**). *Let* $\Phi_n \equiv \mathsf{Unfold}^*(\Psi, \Phi \cup \{v = t\}, n)$*. If* $\mathsf{SmtValid}(\Phi_n, v = t')$*, then for every* $f(\overline{t'}) \prec t'$ *there exists* $f(\overline{t}) \prec \Phi_n$ *such that* $\mathsf{SmtValid}(\Phi_n, t_i = t_i')$.

We prove the above lemma by induction on n where the inductive step uses the following property of congruence closure, which itself is proved by induction on the structure of t':

Lemma 6.6 (**Congruence**). *If* $\mathsf{SmtValid}(\Phi \cup \{v = t\}, v = t')$ *and* $v \notin \Phi, t, t'$*, then for every* $f(\overline{t'}) \prec t'$ *there exists* $f(\overline{t}) \prec \Phi, t$ *such that* $\mathsf{SmtValid}(\Phi, t_i = t_i')$.

Unfolding Preserves Equational Links We use the uncovering Lemma 6.5 and congruence to show that *every instantiation* that is valid after n steps is subsumed by the $n + 1$ depth unfolding. That is, we show that every possible *link* in any equational chain can be proved equal to the source expression via bounded unfolding.

Lemma 6.7 (**Link**). *If* $\mathsf{SmtValid}(\mathsf{Unfold}^*(\Psi, \Phi \cup \{v = t\}, n), v = t')$*, then* $\mathsf{SmtValid}(\mathsf{Unfold}^*(\Psi, \Phi \cup \{v = t\}, n + 1), \mathsf{Unfold}(\Psi,\ \Phi \cup \{v = t'\}))$.

Equational Proof Figure 10 formalizes our rules for equational reasoning. Intuitively, there is an *equational proof* that $t_1 \bowtie t_2$ under Ψ, Φ, written by the judgment $\Psi, \Phi \vdash t_1 \bowtie t_2$, if by some sequence of repeated function unfoldings, we can prove that t_1 and t_2 are respectively equal to t_1' and t_2' such that, $\mathsf{SmtValid}(\Phi, t_1' \bowtie t_2')$ holds. Our notion of equational proofs adapts the idea of type level computation used in TT-based proof assistants to the setting of SMT-based reasoning, via the directional unfolding judgment $\Psi, \Phi \vdash t \twoheadrightarrow t'$. 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 (post-unfolding).

Completeness of Bounded Unfolding 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* $\Psi, \Phi \vdash t \twoheadrightarrow t'$, *then* $\exists 0 \leq n.\ \mathsf{SmtValid}(\mathsf{Unfold}^*(\Psi, \Phi \cup \{v = t\}, n), v = t')$.

PLE ***is a Fixpoint of Bounded Unfolding*** 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**). $\mathsf{PLE}(\Psi, \Phi, t = t')$ *iff* $\exists n.\ \mathsf{SmtValid}(\mathsf{Unfold}^*(\Psi, \Phi \cup \{v = t\}, n), v = t')$.

The proof follows by observing that $\mathsf{PLE}(\Psi, \Phi, t = t')$ computes the *least-fixpoint* of the sequence $\Phi_i \doteq \mathsf{Unfold}^*(\Psi, \Phi, i)$. Specifically, we prove by induction on i that at each invocation of loop (i, Φ_i) in Figure 9, Φ_i is equal to $\mathsf{Unfold}^*(\Psi, \Phi \cup \{v = t\}, i)$, which then yields the result.

Completeness of PLE Finally, we combine Lemmas 6.9 and 6.7 to show that PLE is complete, *i.e.* if there is an equational proof that $t \bowtie t'$ under Ψ, Φ, then $\mathsf{PLE}(\Psi, \Phi, t \bowtie t')$ returns *true*.

Theorem 6.10 (**Completeness**). *If* $\Psi, \Phi \vdash t \bowtie t'$ *then* $\mathsf{PLE}(\Psi, \Phi, t \bowtie t') = \mathit{true}$.

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, 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 $\mathsf{SmtValid}(\lfloor\Gamma\rfloor, \mathit{false})$. An environment Γ is *inhabited* if there exists some $\theta \in (\!|\Gamma|\!)$. We say Γ is *transparent* if it is either inhabited *or* inconsistent. As an example of a *non-transparent* Φ_0 consider the predicate `lenA xs` $= 1 +$ `lenB xs`, where `lenA` and `lenB` 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 termination argument in three steps. 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, $\mathsf{Unfold}^*(\Psi, \Phi, 0) \subset \mathsf{Unfold}^*(\Psi, \Phi, 1) \ldots$ which corresponds to an *infinite logical trace*. Third, as the logical computation is a 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.

Totality A function is *total* when its evaluation reduces to exactly one value. The totality of R can and is checked by refinement types (§ 4). Hence, for brevity, in the sequel we *implicitly assume* that R is total under Γ.

*Definition 6.11 (**Total**).* Let $b \equiv \lambda \overline{x}.\langle \overline{\lfloor p \rfloor \Rightarrow \lfloor e \rfloor} \rangle$. b is *total* under Γ and R, if forall $\theta \in (\!|\Gamma|\!)$:

(1) if $\theta \cdot R[p_i] \hookrightarrow^\star \mathtt{True}$, then $\exists v.\ \theta \cdot R[e_i] \hookrightarrow^\star v$,
(2) if $\theta \cdot R[p_i] \hookrightarrow^\star \mathtt{True}$ and $\theta \cdot \Psi[p_j] \hookrightarrow^\star \mathtt{True}$, then $i = j$, and
(3) there exists an i so that $\theta \cdot R[p_i] \hookrightarrow^\star \mathtt{True}$.

R is *total* under Γ, if every $b \in \lfloor R \rfloor$ is total under Γ and R.

Subterm Evaluation As the reflected functions are total, the Church-Rosser theorem implies that evaluation order is not important. To prove termination, we require an evaluation strategy, *e.g.* CBV, in which if a reflected function's guard is satisfied, then the evaluation of the corresponding function body requires evaluating *every subterm* inside the body. As $\mathtt{DeIf}(\cdot)$ hoists `if`-expressions out of the body and into the top-level guards, the below fact follows from the properties of CBV:

Lemma 6.12. *Let* $b \equiv \lambda \overline{x}.\langle \overline{\lfloor p \rfloor \Rightarrow \lfloor e \rfloor} \rangle$ *and* $f \in R$. *For every* Γ, R, *and* $\theta \in (\!|\Gamma|\!)$, *if* $\theta \cdot R[p_i] \hookrightarrow^\star \mathtt{True}$ *and* $f(\overline{\lfloor e' \rfloor}) \prec \lfloor e_i \rfloor$, *then* $\theta \cdot R[e_i] \hookrightarrow^\star C[f(\theta \cdot R[\overline{e'}])]$.

Logical Step A pair $f(\overline{t}) \rightsquigarrow f'(\overline{t'})$ is a Ψ, Φ-*logical step* (abbrev. step), if

- $\Psi(f) \equiv \lambda \overline{x}.\langle \overline{p \Rightarrow b} \rangle$,
- $\mathsf{SmtValid}(\Phi \wedge Q, p_i)$ for some (Ψ, Φ)-instance Q, and
- $f'(\overline{t'}) \prec b_i \left[\overline{t}/\overline{x}\right]$.

Steps and Reductions Next, using Lemmas 6.12, 6.1, and the definition of logical steps, we show that every logical step corresponds to a *sequence* of steps in the concrete semantics:

Lemma 6.13 (**Step-Reductions**). *If* $f(\overline{\lfloor e \rfloor}) \rightsquigarrow f'(\overline{\lfloor e' \rfloor})$ *is a logical step under* $\lfloor R \rfloor, \lfloor \Gamma \rfloor$ *and* $\theta \in (\!|\Gamma|\!)$, *then* $f(\overline{\theta \cdot R[e]}) \hookrightarrow^\star C[f(\theta \cdot R[\overline{e'}])]$ *for some context* C.

Logical Trace A sequence $f_0(\overline{t_0}), f_1(\overline{t_1}), f_2(\overline{t_2}), \ldots$ is a Ψ, Φ-*logical trace* (abbrev. trace), if $f_i(\overline{t_i}) \rightsquigarrow f_{i+1}(\overline{t_{i+1}})$ is a Ψ, Φ-step, for each i. Our termination proof hinges upon the following key result: inhabited environments only have *finite* logical traces. We prove this result by contradiction. Specifically, we show by Lemma 6.13 that an infinite $(\lfloor R \rfloor, \lfloor \Gamma \rfloor)$-trace combined with the fact that Γ is inhabited yields *at least one infinite concrete trace*, which contradicts the totality assumption. Hence, all the $(\lfloor R \rfloor, \lfloor \Gamma \rfloor)$ logical traces must be finite.

Theorem 6.14 (**Finite-Trace**). *If* Γ *is inhabited, then every* $(\lfloor R \rfloor, \lfloor \Gamma \rfloor)$*-trace is finite.*

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**.

Benchmark	**Common**		**Without PLE Search**			**With PLE Search**		
	Impl (l)	Spec (l)	Proof (l)	Time (s)	SMT (q)	Proof (l)	Time (s)	SMT (q)
Arithmetic								
Fibonacci	7	10	38	2.74	129	16	1.92	79
Ackermann	20	73	196	5.40	566	119	13.80	846
Class Laws Fig 11								
Monoid	33	50	109	4.47	34	33	4.22	209
Functor	48	44	93	4.97	26	14	3.68	68
Applicative	62	110	241	12.00	69	74	10.00	1090
Monad	63	42	122	5.39	49	39	4.89	250
Higher-Order Properties								
Logical Properties	0	20	33	2.71	32	33	2.74	32
Fold Universal	10	44	43	2.17	24	14	1.46	48
Functional Correctness								
SAT-solver	92	34	0	50.00	50	0	50.00	50
Unification	51	60	85	4.77	195	21	5.64	422
Deterministic Parallelism								
Conc. Sets	597	329	339	40.10	339	229	40.70	861
n-body	163	251	101	7.41	61	21	6.27	61
Par. Reducers	30	212	124	6.63	52	25	5.56	52
Total	1176	1279	1524	148.76	1626	638	150.88	4068

Ascending Chains and Traces If unfolding Ψ, Φ yields an infinite chain $\Phi_0 \subset \ldots \subset \Phi_n \ldots$, then Ψ, Φ has an infinite logical trace. We construct the trace by selecting at level i (*i.e.* in Φ_i) an application term $f_i(\overline{t_i})$ that was created by unfolding an application term at level $i-1$ (*i.e.* in Φ_{i-1}).

LEMMA 6.15 (**ASCENDING CHAINS**). *Let* $\Phi_i \doteq \mathsf{Unfold}^*(\Psi, \Phi, i)$. *If there exists an (infinite) ascending chain* $\Phi_0 \subset \ldots \subset \Phi_n \ldots$, *then there exists an (infinite) logical trace* $f_0(\overline{t_0}), \ldots, f_n(\overline{t_n}), \ldots$.

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 Φ_i, 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* $\mathsf{PLE}(\lfloor R \rfloor, \lfloor \Gamma \rfloor, p)$ *terminates.*

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. 2017b] 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 `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. 2017b] 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

```
withTheorem :: x:a → Prop → { 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

```
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 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 $2-5\times$ — 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*

Monoid (for Peano, Maybe, List)		**Functor** (for Maybe, List, Id)	
Left Id.	$\texttt{mempty}\ x \diamond\ \equiv x$	Id.	$\texttt{fmap id}\ xs \equiv \texttt{id}\ xs$
Right Id.	$x \diamond \texttt{mempty} \equiv x$	Distr.	$\texttt{fmap}\ (g \circ h)\ xs \equiv (\texttt{fmap}\ g\ \circ\ \texttt{fmap}\ h)\ xs$
Assoc.	$(x \diamond y) \diamond z \equiv x \diamond (y \diamond z)$		
Applicative (for Maybe, List, Id)		**Monad** (for Maybe, List, Id)	
Id.	$\texttt{pure id}\ \circledast\ v \equiv v$	Left Id.	$\texttt{return}\ a \gg= f \equiv f\ a$
Comp.	$\texttt{pure}\ (\circ) \circledast u \circledast v \circledast w \equiv u \circledast (v \circledast w)$	Right Id.	$m \gg= \texttt{return}\ \equiv m$
Hom.	$\texttt{pure}\ f\ \circledast\ \texttt{pure}\ x \equiv \texttt{pure}\ (f\ x)$	Assoc.	$(m \gg= f) \gg= g \equiv m \gg= (\lambda x \to f\ x \gg= g)$
Inter.	$u\ \circledast\ \texttt{pure}\ y \equiv \texttt{pure}\ (\$\, y)\ \circledast\ u$		
Ord (for Int, Double, Either, (,))		**Commutative Monoid** (for Int, Double, (,))	
Refl.	$x \leq x$	Comm.	$x \diamond y \equiv y \diamond x$
Antisym.	$x \leq y \land y \leq x \implies x \equiv y$		
Trans.	$x \leq y \land y \leq z \implies x \leq z$		*(including Monoid laws)*
Total.	$x \leq y \lor y \leq x$		

Fig. 11. Summary of Verified Typeclass Laws.

to type-check the compact proofs enabled by PLE than it is to type-check the $2 - 5\times$ 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 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 upto 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 β-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. 2017a] 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 and (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 to design 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

N. Amin, K. R. M. L., and T. Rompf. 2014. Computing with an SMT Solver. In *TAP*.

A. Appel. 2016. *Verified Functional Algorithms*. https://www.cs.princeton.edu/~appel/vfa/Perm.html.

K. Asada, R. Sato, and N. Kobayashi. 2015. Verifying Relational Properties of Functional Programs by First-Order Refinement. In *PEPM*.

L. Augustsson. 1998. Cayenne - a Language with Dependent Types.. In *ICFP*.

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

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

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

R. S. Bird. 1989. Algebraic Identities for Program Calculation. In *The Computer Journal*.

J. C. Blanchette, S. Böhme, and L. C. Paulson. 2011. Extending Sledgehammer with SMT Solvers. In *CADE*.

Jr. Bocchino, L. Robert, V. S. Adve, S. V. Adve, and M. Snir. 2009. Parallel Programming Must Be Deterministic by Default. In *HotPar*.

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

J. Chen, R. Chugh, and N. Swamy. 2010. Type-preserving Compilation of End-to-end Verification of Security Enforcement. In *PLDI*.

K. Claessen and J. Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP*.

E. M. Clarke, O. Grumberg, and D.E. Long. 1992. Model checking and abstraction. In *POPL*.

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

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

P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for the Static Analysis of Programs. In *POPL*.

E. W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. In *Communications of the ACM*.

E. W. Dijkstra. 1976. *A Discipline of Programming.* Prentice-Hall.

R. A. Eisenberg and J. Stolarek. 2014. Promoting functions to type families in Haskell. In *Haskell*.

A. Farmer, N. Sculthorpe, and A. Gill. 2015. Reasoning with the HERMIT: Tool Support for Equational Reasoning on GHC Core Programs. In *Haskell*.

G. Gentzen. 1935. Inverstigations into Logical Deduction. In *American Philosophical Quarterly*.

W. A. Howard. 1980. The formulae-as-types notion of construction. In *Essays on Combinatory Logic, Lambda Calculus and Formalism*.

G. Hutton. 1999. A tutorial on the universality and expressiveness of fold. *J. Functional Programming* (1999).

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

K. W. Knowles and C. Flanagan. 2010. Hybrid type checking. In *TOPLAS*.

L. Kuper, A. Turon, N. R Krishnaswami, and R. R Newton. 2014. Freeze after writing: quasi-deterministic parallel programming with LVars. In *POPL*.

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

K. R. M. Leino and C. Pit-Claudel. 2016. Trigger selection strategies to stabilize program verifiers. In *CAV*.

K. R. M. Leino and N. Polikarpova. 2016. Verified Calculations. In *VSTTE*.

R. Leino. 2016. Dafny. (2016). https://github.com/Microsoft/dafny/blob/master/Test/dafny0/Fuel.dfy.

S. Marlow, R. Newton, and S. Peyton-Jones. 2011. A Monad for Deterministic Parallelism. In *Haskell*.

S. C. Mu, H. S. Ko, and P. Jansson. 2009. Algebra of Programming in Agda: Dependent Types for Relational Program Derivation. In *J. Funct. Program.*

G. C. Necula. 1997. Proof carrying code. In *POPL*.

C. G. Nelson. 1980. *Techniques for Program Verification.* Ph.D. Dissertation. Stanford University.

T. Nipkow, L.C. Paulson, and M. Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.*

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

X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. 2004. Dynamic Typing with Dependent Types. In *IFIP TCS*.

J. C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *ACM National Conference*.

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

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

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

E. L. Seidel, N. Vazou, and R. Jhala. 2015. Type Targeted Testing. In *ESOP*.

V. Sjöberg and S. Weirich. 2015. Programming Up to Congruence. In *POPL*.

W. Sonnex, S. Drossopoulou, and S. Eisenbach. 2012. Zeno: An Automated Prover for Properties of Recursive Data Structures. In *TACAS*.

M. Sousa and I. Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *PLDI*.

P. Suter, M. Dotta, and V. Kuncak. 2010. Decision Procedures for Algebraic Data Types with Abstractions. In *POPL*.

P. Suter, A. Sinan Köksal, and V. Kuncak. 2011. Satisfiability Modulo Recursive Programs. In *SAS*.

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

G. Tourlakis. 2008. Ackermann's Function. (2008). http://www.cs.yorku.ca/~gt/papers/Ackermann-function.pdf.

N. Vazou, L. Lampropoulos, and J. Polakow. 2017a. A Tale of Two Provers: Verifying Monoidal String Matching in Liquid Haskell and Coq. In *Haskell*.

N. Vazou, P. Rondon, and R. Jhala. 2013. Abstract Refinement Types. In *ESOP*.

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

N. Vazou, A. Tondwalkar, V. Choudhury, R. G. Scott, R. R. Newton, P. Wadler, and R. Jhala. 2017b. Extended Version: Refinement Reflection: Complete Verification with SMT. (2017). https://nikivazou.github.io/static/popl18/extended-refinement-reflection.pdf

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

D. Vytiniotis, S. L. Peyton-Jones, K. Claessen, and D. Rosén. 2013. HALO: Haskell to Logic through Denotational Semantics. In *POPL*.

P. Wadler. 1987. A Critique of Abelson and Sussman or Why Calculating is Better Than Scheming. In *SIGPLAN Not.*

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

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