Towards Complete Specification and Verification with SMT

ANONYMOUS AUTHOR(S)

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

ACM Reference format:
Anonymous Author(s). 2017. Towards Complete Specification and Verification with SMT. *Proc. ACM Program. Lang.* 1, 1, Article 1 (July 2017), 29 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

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

2017. 2475-1421/2017/7-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1. Refinement Reflection Our first contribution is the notion of *refinement reflection*. To reason about user-defined functions, the function's implementation can be *reflected* into its (output) refinement-type specification, thus converting the function's type signature into a precise description of the function's behavior. This simple idea has a profound consequence: at *uses* of the function, the standard rule for (dependent) function application yields a precise means of reasoning about the function (§ 4).

2. Complete Specification NV ♣ CHECK ♣ Reflection lets us represent expressive specifications, that fall well outside of SMT-decidable logics, simply as unit-values refined by logical propositions. 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 transcriptions of their pencil-and-paper analogues (§ 2). Furthermore, since proofs are literally just programs, we demonstrate, via the Curry-Howard Isomorphism, that Refinement Reflection yields a proof system that encodes natural deduction (§ 3).

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

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

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 equational 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 can automatically verify that if `fib` is invoked with a non-negative `Integer`, then it terminates and yields a non-negative `Integer`.

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

```
type Prop = ()
```

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

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

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

Universal & Existential Propositions Refinements encode universally-quantified propositions as *dependent function* types of the form:

```
type Plus_com = 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. Following the standard encoding of [Howard 1980], refinements encode existential quantification via *dependent pairs* of the form:

```
type Nat_up = n:Nat → (m::Integer, {n < m})
```

The notation `(m :: t, t')` describes dependent pairs where the name `m` for the first element can appear inside refinements for the second element. Thus, `Nat_up` states the proposition that for every natural number `n` *there exists* value 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 logical connectives (like ∧ and ∨). In this paper, we present how to encode logical connectives using data types, *e.g.* conjunction as product and disjunction as a union, and show how to specify arbitrary higher-order logic (HOL) propositions using refinement types and how to verify those propositions using refinement type checking (§ 3).

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

```
plus_2_2 :: Plus_2_2
plus_2_2 = ()

plus_com :: Plus_com
plus_com = \x y → ()

nat_up :: Nat_up
nat_up = \n → (n+1,())
```

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

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

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

RN ♣ On the call we discussed showing or hinting at non-arithmetic examples above, instead of all arithmetic. ♣

A Note on Bottom: Readers familiar with Haskell's semantics may be concerned that "bottom", which inhabits all types, makes our proofs suspect. Fortunately, as described in Vazou et al. [2014], Liquid Haskell ensures that all terms with non-trivial refinements provably terminate and evaluate to (non-bottom) values, which makes our proofs sound.

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* $\forall n, m.\ n = m \Rightarrow$ `fib` n = `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 following VC (which requires reasoning about the *definition* of `fib`)

$$true \Rightarrow \texttt{fib}\ 2 = 1$$

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:

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

Note that 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. 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 introduce additional tools for proof composition. Then, in § 2.4 we will eliminate unnecessary details in such proofs, using *Proof by Logical Evaluation* (PLE) for automation.

2.3 Equational Proofs

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

"Equation" Combinators We equip Liquid Haskell with a family of equation combinators, $\odot$, 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 period "`.`" to these operators, so that "`=`" becomes "`=.`" instead.) 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 = ()
```

"Because" Combinators 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 with a "because" combinator:

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

The operator is simply an alias for function application that lets us write `x ⊙ y ∵ p`. We use the because 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.

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, `fib` $n \leq$ `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 *base* cases `0` and 1, we simply assert the relevant inequalities. These are verified as the reflected refinement unfolds the definition of `fib` at those inputs. The derived VCs are (automatically) proved as the SMT solver concludes $0 < 1$ and $1 + 0 \leq 1$ respectively. When `n` is greater than two, `fib n` is unfolded to `fib (n-1) + fib (n-2)`, which, as `fib (n-2)` is non-negative, completes the proof.

Induction & Higher Order Reasoning Refinement reflection smoothly accomodates induction and higher-order reasoning. For example, lets 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:_ → 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 `fUp 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 elegant, 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]:

- ***Step 1: Guard Normal Form*** First, as shown in the definition of `fibP` above, each reflected function is transformed into a *guard normal form* $\wedge_i p_i \Rightarrow f(\overline{x}) = b_i$ *i.e.* as a collection of *guards* p_i and their corresponding definition b_i.
- ***Step 2: Unfolding*** Second, given a VC of the form $\Phi \Rightarrow p$, we iteratively *unfold* function application terms in 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 a VC *true* $\Rightarrow$ `fib` 3 = 2, the guard $3 \geq 1$ is trivially *enabled*, *i.e.* is true, and 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.
- ***Step 3: Fixpoint*** Third, we repeat the above process until either the goal is proved or we have reached a fixpoint, *i.e.* no further unfolding is enabled. For example, the above fixpoint computation unfolds the definition of `fib` at 2 and 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 exists. Second, that under certain conditions readily met in practice, it is guaranteed to terminate. These two properties allow us to use PLE to predictably automate proofs: the programmer needs *only* supply the relevant induction hypotheses or helper lemma applications.

```
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.

The remaining long chains of calculations are performed automatically via SMT-based PLE. (That is, they must provide case statements and recursive structure, but *not* chains of =. applications.) To wit, with complete proof search, the above proofs shrink to:

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

```
fibUp  :: Up fib
fibUp 0 = ()
fibUp 1 = ()
fibUp n = ()
```

```
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 = ()` adds inserts 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 verifying assertions over user-defined functions `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 `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 instantation can easily lead to infinite "matching loops". For example, the existence of a term `fib` n in a VC can trigger the above axiom, which may then produce the terms `fib` $(n-1)$ and `fib` $(n-2)$, which may then recursively give rise to further instantiations *ad infinitum*. To prevent matching loops an expert must carefully craft "triggers" or, alternatively provide a "fuel" parameter [Amin et al. 2014] that bounds the depth of instantiation. Both these approaches ensure termination, but can cause the axiom to not be instantiated at the 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
      | otherwise = 1 + pos (n-1)
```

```
test   :: y:{y > 5} → {pos n = 3 + pos (n-3)}
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]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

```
reflect map :: (a → b) → [a] → [b]
map f []     = []
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 ++ into the refinement: `if isNil` *xs* `then` *ys* `else sel1` *xs* `:` (`sel2` *xs* ++ *ys*) Note that Liquid Haskell automatically checks that ++ and map are total [Vazou et al. 2014], which lets us safely `reflect` them into the refinement logic.

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 }
```

Proofs On the right in Figure 1 we show the proofs of these laws using PLE, which should be compared to the classical equational proof *e.g.* [Wadler 1987] shown on the left. With PLE, the user need only provide the high-level structure — the case splits and invocations of the induction hypotheses — after which PLE automatically completes the rest of the equational proof. Thus using SMT-based PLE, `app_assoc` shrinks down to its essence: an induction over the list `xs`. The difference is even more stark with `map_fusion` whose full equational proof we don't show, 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 PLE and (decidable) SMT-based theory reasoning can greatly simplify proofs. For example, consider the `swap` function from a Coq textbook [Appel 2016]:

```
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 three proofs that `swap` is idempotent: Appel's proof using Coq (simplified by the use of a hint database and the arithmetic tacic `omega`), its variant in Agda, and finally the PLE proof. It is readily apparent that PLE's combination of proof search working hand-in-glove with SMT-based theory reasoning makes proving the result relatively trivial. Of course, proof assistants like Agda, Coq, and Isabelle emit easily checkable certificates and have decades-worth of tactics, libraries and proof scripts that enable large scale proof engineering. We merely use this example it to illustrate that reflection and SMT-based proof search bring powerful and complete new tools to simplify specification and verification ; and defer a longer discussion to § 8.

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 = ()
```

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

3 COMPLETE SPECIFICATION: EMBEDDING NATURAL DEDUCTION WITH REFINEMENT TYPES

In this section, we show that the restriction for the refinement language to be quantifier free—crucial for SMT-decidable type checking—does not pose any expressiveness restrictions to the expressiveness of the specifications. Instead, *quantified* specifications can be naturally encoded using λ-abstractions and dependent pairs to encode universal and existential quantification respectively, while proof terms can be generated using the standard natural deduction derivation rules. Our encoding is completely standard [Wadler 2015], also known as the Curry-Howard isomorphism [Howard 1980]. What is new is that we exploit this encoding so that Liquid Haskell 1. can represent any proof in Gentzen's natural deduction [Gentzen 1935] while it 2. it still takes advantage of SMT-automation to prove propositions and thus discuss how 3.the quantifier-free portion of natural deduction proofs can be deferred to the SMTs.

NV ♣ CHECK ♣

NV ♣ TO remove? ♣ NV ♣ In this section we assume the requirement that all the expressions are total. ♣

NV ♣ extract totality definition from pbe? ♣ NV ♣ Phil suggests to make all types downarrow and thus delete this restriction. This cannot happen in Haskell: diverging programs (like unify in eval) would not type check ♣

```
If you had polymorphism, you can encode existentials into \lambda^R as follows:
  (x::\tau, \phi) = \forall \alpha. (x : \tau -> \phi -> \alpha) -> \alpha
But you don't have polymorphism in \lambda^R, do you?
```

3.1 The specification language

Figure 3 maps logical predicates to refinement types using type constructs to represent quntification, while the refinements remain quantifier-free. NV ♣ Check? ♣

Native terms Native terms consist of all (quantifier-free) expressions of the refinement languages. For example in § 4 we formalize Liquid Haskell as 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 (ϕ_1, ϕ_2). and disjunction $\phi_1 \vee \phi_2$ is encoded with the sum type `Either` that contains the proofs of one of the two disjuncts.

	Refinement Type	Logical Formula
Native Terms	$\{e\}$	e
Implication	$\phi_1 \rightarrow \phi_2$	$\phi_1 \Rightarrow \phi_2$
Negation	$\phi \rightarrow \{\texttt{False}\}$	$\neg\phi$
Conjunction	(ϕ_1, ϕ_2)	$\phi_1 \wedge \phi_2$
Disjunction	$\texttt{Either}\ \phi_1\ \phi_2$	$\phi_1 \vee \phi_2$
Forall	$x : \tau \rightarrow \phi$	$\forall x.\phi$
Exists	$(x :: \tau, \phi)$	$\exists x.\phi$

Fig. 3. Correspondence between quantifier-free refinement types and logical properties. $\{e\}$ simplifies $\{v : \texttt{Prop} \mid e\}$. Function binders are not represented in negation and implication where they are not relevant.

```
data Either a b = Left a | Right b
```

Quantifiers Universal quantification $\forall x.\phi$ is encoded and introduced as lambda abstraction $x : a \rightarrow \phi$ and eliminated by function application. Existential quantification $\exists x.\phi$ is encoded as a dependent pair `(x::a, ϕ)` that contains 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]. and do not add extra complexity to the system. In Liquid Haskell we added the syntax for dependent pairs in Figure 3 as a syntactic sugar for abstract refinements. NV ♣ CHECK ♣

3.2 Proof Terms: Proofs via Natural Deduction

We defined ϕ to be both a proposition and a refinement type. We connect these two meanings of ϕ by proving that, due to Curry-Howard isomorphism [Howard 1980], if there exist an expression with refinement type ϕ, then the proposition ϕ is valid.

But how does one construct a proof term e for a formula ϕ? For this construction we can use Gentzen's natural deduction system. Since each natural deduction derivation rule from [Gentzen 1935] maps to a derivation of λ^R. To get the rules for natural deduction one should read $\Gamma \vdash e : \phi$ as "ϕ is provable under the assumptions of Γ". Let $\Gamma \vdash_{ND} \phi$ stand for the logical judgement under assumption Γ, proposition ϕ holds in Gentzen's natural deduction. Then each of Gentzen's logical rules can be recovered from those in Figure **??** by rewriting each judgement $\Gamma \vdash e : \phi$ of λ^R as $\Gamma \vdash_{ND} \phi$. For example, conjunction and universal elimination derive as follows.

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

Since Figure **??** maps directly natural deduction rules to derivations that are accepted by λ^R, we conclude that if there is a natural deduction derivation for a proposition ϕ, then there exists a λ^R term that proves this formula.

Theorem 3.1. *If $\Gamma \vdash_{ND} \phi$ then we can construct an e so that $\Gamma; \emptyset \vdash e : \phi$.*

3.3 Examples

Next we illustrate examples of proofs for quantified propositions. These propositions range from textbook logical properties to properties over user-defined domains (here lists), and even induction on integers.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p{:}\phi_p, y{:}\tau_y \vdash p : \phi_p} \quad \dfrac{\dfrac{}{p{:}\phi_p, y{:}\tau_y, x{:}t_x, p_x{:}\phi_x \vdash p_x : \phi_x} \quad \dfrac{}{p{:}\phi_p, y{:}\tau_y, x{:}t_x, p_x{:}\phi_x \vdash y : \tau_y}}{p{:}\phi_p, y{:}\tau_y, x{:}t_x, p_x{:}\phi_x \vdash p_x\ y : f\ x\ y}\,\forall\text{-E}}{p{:}\phi_p, y{:}\tau_y \vdash \mathsf{case}\ p\ \mathsf{of}\ \{(x, p_x) \to (x, p_x\ y)\} : \exists x.(f\ x\ y)}\,\exists\text{-E}}{p{:}\phi_p \vdash \lambda y.\mathsf{case}\ p\ \mathsf{of}\ \{(x, p_x) \to (x, p_x\ y)\} : \forall y.\exists x.(f\ x\ y)}\,\forall\text{-I}}{\emptyset \vdash \lambda p\ y.\mathsf{case}\ p\ \mathsf{of}\ \{(x, p_x) \to (x, p_x\ y)\} : (\exists x.\forall y.(f\ x\ y)) \Rightarrow (\forall y.\exists x.(f\ x\ y))}\,\Rightarrow\text{-I}$$

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

3.3.1 Natural Deduction as Type Derivation. To see the mapping from natural deduction to typing rules in action, Figure 4 is using typing judgments to express the Gentzen's proof of the proposition

$$\phi \equiv (\exists x.\forall y.(f\ x\ y)) \Rightarrow (\forall y.\exists x.(f\ x\ y))$$

Reading bottom up the derivation provides a proof of ϕ, while reading top down it constructs the proof term of the formula to be $\lambda p\ y.\mathsf{case}\ p\ \mathsf{of}\ \{(x, p_x) \to (x, p_x\ y)\}$. This proof term corresponds directly to the following Haskell expression that Liquid Haskell type checks to have the refinement type representing ϕ.

```
existsForall :: f:(a → a → Bool)
             → (x::a,y:a → {f x y}) → y:a → (x::a,{f x y})
existsForall f = \p y → case p of {(x,px) → (x,px y)}
```

Using Haskell's pattern matching we can further simplify the proof term to

```
existsForall f (x,px) y = (x, px y)
```

In the rest of the examples we prove logical properties by constructing the Haskell proof terms.

3.3.2 Logical Properties. Next, we prove distribution of existentials over disjunction:

$$\phi \equiv \exists x.(p\ x \vee q\ x)) \Rightarrow ((\exists x.p\ x) \vee (\exists x.q\ x))$$

The proof is a λ^R program (here expressed using Haskell). The specification of this property requires nesting existentials inside disjunctions and vice versa. The proof proceeds by existential case splitting and introduction:

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

PW ♣ Phil suggests to show the proof derivation tree here too ♣

Similarly, we distribute foralls over conjunction:

$$\phi \equiv \forall x.(p\ x \wedge q\ x)) \Rightarrow ((\forall x.p\ x) \wedge (\forall x.q\ x))$$

The specification of the conclusion now requires nesting universal quantification over conjunctions. This requirement leads to a proof term that performs λ-abstraction and case spitting inside the conjunction pair.

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

3.3.3 Properties on user specified domains. Since native terms are drawn from user-defined decidable logics, the terms in ϕ can talk about properties of data types, like lists. As an example we prove that forall lists `xs` if there exists a list `ys` so that `xs == ys ++ ys` then `xs` has even length.

$$\phi \equiv (\forall xs.\exists ys.xs = ys{+}{+}ys) \Rightarrow (\exists n.\texttt{length}\ xs = n + n)$$

The proof proceeds by existential elimination and introduction and by invocation of the `lenAppend` lemma.

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

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

The `lenAppend` lemma is proven by induction on the input list, while PLE is used to simplify the trivial unfoldings.

3.3.4 Induction on Natural Numbers. Finally, we use Liquid Haskell to specify and verify induction on natural numbers.

$$\phi \equiv (p\ 0 \wedge (\forall n.p\ (n-1) \Rightarrow p\ n)) \Rightarrow \forall n.p\ n$$

```
natInd :: p:(Int→ Bool) → ({p 0}, (n:Int → {p (n-1)} → {p n}))
       → n:Nat → {p n}
natInd p (p0, pn) n
  | n == 0    = p0
  | otherwise = pn n (natInd p (p0, pn) (n-1))
```

The proof proceeds by induction (*e.g.* case splitting). In the base case, `n == 0`, the proof calls the left conjunct. Otherwise, `0 < n`, the proof calls the right conjuct instantiated on the correct argument `n` and assuming the inductive hypothesis.

RN ♣ What's the punchline of this section? ♣

RN ♣ BTW, still not much intuition at this point about how the implementation works – e.g. how times does it call the SMT solver, and when? ♣

3.4 Consequences

The discussion is this section is not novel, it is merely an application of Curry-Howard correspondence to our system. What is novel is the evidence that Curry-Howard correspondence actually applies to λ^R which leads to two major contributions.

Fist, we show that natural deduction reasoning can smoothly co-exists with SMT-based verification to automate both the quantifier-free portions and the user-defined domains portions (*e.g.*

properties of lists) of the proof. To use the SMT-solver to discharge proofs, for each (quantifier-free) propositional connective of ϕ, we can define a refinement operator that maps propositions to refinements. For instance, we refine conjunction by performing case analysis on the `({b1}, {b2})` constructor therefore bringing both the conjuncts into the environment leading to the trivial VC $b1 \wedge b2 \Rightarrow b1 \wedge b2$ that can be trivially discharged by the SMT-solver.

```
andRefine :: b1:Bool → b2:Bool → ({b1}, {b2}) → {b1 && b2}
andRefine _ _ (b1, b2) = ()
```

Using `andRefine` the user does not have to explicitly eliminate `PAnd`. For instance the `forallAndDistr` proof from § 3.3.2 simplifies to the following.

```
forallAndDistr :: p:(a → Bool) → q:(a → Bool)
               → (x:a → ({p x}, {q x}))
               → ((x:a → {p x}), (x:a → {q x}))
forallAndDistr _ _ andx = (fAndG, fAndG)
  where fAndG = andRefine (f x) (g x) (andx x)
```

On the other direction, we reify SMT conjunction by using $\phi_1 \wedge \phi_2$ as a proof for each conjunct ϕ_1 and ϕ_2 that can later be used in natural deduction derivations.

```
andReify :: b1:Bool → b2:Bool → {b1 && b2} → ({b1}, {b2})
andReify _ _ b = (b, b)
```

Second, since λ^R is implemented in Liquid Haskell we show for first time how natural deduction proofs are encoded in Liquid Haskell which sets clearer bounds for the expressiveness of the language, gives a guideline for encoding proofs with nested quantifies, and provides a pleasant implementation of natural deduction that can be used for pedagogical purposes.

4 REFINEMENT REFLECTION: λ^R

We formalize refinement reflection in three steps. First, we develop a core calculus λ^R with an *undecidable* type system based on denotational semantics. We show how the soundness of the type system allows us to *prove theorems* using λ^R. 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 5 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 $\oplus$, here, the set $\{=, <\}$. Moreover, they include the booleans `True`, `False`, integers $-1, 0, 1$, *etc.*, and logical operators $\wedge$, $\vee$, `!`, *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

$$
\begin{array}{rrcl}
\textbf{\textit{Operators}} & \odot & ::= & = \mid < \\
\textbf{\textit{Constants}} & c & ::= & \wedge \mid \texttt{!} \mid \odot \mid +, -, \ldots \\
 & & \mid & \texttt{True} \mid \texttt{False} \mid 0, \pm 1, \ldots \\
\textbf{\textit{Values}} & w & ::= & c \mid \lambda x.e \mid D\ \overline{w} \\
\textbf{\textit{Expressions}} & e & ::= & w \mid x \mid e\ e \\
 & & \mid & \texttt{case}\ x = e\ \texttt{of}\ \{D\ \overline{x} \rightarrow e\} \\
\textbf{\textit{Binders}} & b & ::= & e \mid \texttt{let rec}\ x : \tau = b\ \texttt{in}\ b \\
\textbf{\textit{Program}} & p & ::= & b \mid \texttt{reflect}\ x : \tau = e\ \texttt{in}\ p \\
\textbf{\textit{Basic Types}} & B & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \\
\textbf{\textit{Ref. Types}} & \tau & ::= & \{v : B^{[\Downarrow]} \mid e\} \mid x{:}\tau_x \rightarrow \tau
\end{array}
$$

Fig. 5. **Syntax of λ^R**: a calculus with an undecidable type system

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', $(f = f') \hookrightarrow \texttt{True}$ 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 can be checked using refinement types and are used to ensure that refinements cannot diverge as required for soundness of type checking under lazy evaluation. Finally, we have dependent *function types* $x{:}\tau_x \rightarrow \tau$ 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 $\tau_x \rightarrow \tau$ to abbreviate $x{:}\tau_x \rightarrow \tau$ 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 $\texttt{shape}(\tau)$ 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,

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

4.4 Refinement Reflection

The key idea in our work is to *strengthen* the output type of functions 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 6).

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 $\lambda n.e_{\texttt{fib}}$.

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

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

$$\mathsf{Reflect}(t_{\texttt{fib}}, e_{\texttt{fib}}) = n{:}\{v : \texttt{Int}^{\Downarrow} \mid 0 \leq v\} \to \{v : \texttt{Int}^{\Downarrow} \mid 0 \leq v \wedge v = e_{\texttt{fib}}\}$$

Termination Checking We defined $\mathsf{Reflect}(\cdot, \cdot)$ 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 6 is used to type check the reflected function by strengthening the f's result via $\mathsf{Reflect}(\cdot, \cdot)$. Finally, the rule T-Let is used to check that the automatically strengthened type of f satisfies f's implementation.

RN ♣ Hmm, the ordering is a bit weird here where there are a couple references to fig 5, but then the next subsec starts with an introductory tone. ♣

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

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

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

$$\frac{\Gamma; R \vdash e_1 : (x{:}\tau_x \to \tau) \quad \Gamma; R \vdash e_2 : \tau_x}{\Gamma; R \vdash e_1\ e_2 : \tau}\ \textsc{T-App} \qquad \frac{\Gamma, x : \tau_x; R \vdash b_x : \tau_x \quad \Gamma, x : \tau_x \vdash \tau_x \quad \Gamma, x : \tau_x; R \vdash b : \tau \quad \Gamma \vdash \tau}{\Gamma; R \vdash \mathtt{let\ rec}\ x : \tau_x = b_x\ \mathtt{in}\ b : \tau}\ \textsc{T-Let}$$

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

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

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. 6. Typing of λ^R

4.5 Typing Rules

Next, we present the type-checking rules of λ^R, as found in Figure 6.

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:

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

Typing A judgment $\Gamma; R \vdash p : \tau$ states that the program p has the type τ in the environment Γ under 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 the standard refinement typing rules [Knowles and Flanagan 2010; Vazou et al. 2014] except for the addition of the reflection environment R at each rule. First, rule T-Refl is used to extend the reflection environment with the binding of the function name with its definition ($f \mapsto e$) and moreover 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 $\{v : \texttt{Int}^{\Downarrow} \mid \texttt{fibP}\ 1 \wedge v = \texttt{fib}\ 1\}$ where the first conjunct comes from the (reflection-strengthened) output refinement of `fib` § 2 and the second comes from rule T-Exact.

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 the denotation of Γ, $[\![\theta \cdot \tau_1]\!]$ is contained in $[\![\theta \cdot \tau_2]\!]$.

Soundness Following λ^U [Vazou et al. 2014], in Supplementary-Material [2017] we prove that evaluation preserves typing and typing implies denotational inclusion.

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* $\emptyset; \emptyset \vdash w : \tau$.

Theorem 4.1 lets us interpret well typed programs as proofs of propositions. For example, in § 2 we verified that the term `fibUp` proves $n{:}\texttt{Nat} \rightarrow \{\texttt{fib}\ n \leq \texttt{fib}\ (n+1)\}$. 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

Next, we describe λ^S, a conservative, 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 $\preceq$-Base-PBE, yielding an SMT-based algorithmic type system that is both sound and decidable.

Syntax: Terms & Sorts Figure 7 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 1981]. 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 built-in integer `Int` and `Bool` for representing integers and booleans. 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`. The universal sort `U` represents all other values.

$$
\begin{array}{rrcl}
\textbf{\textit{Predicates}} & p & ::= & p \bowtie p \mid \oplus_1 p \\
 & & \mid & n \mid b \mid x \mid D \mid x\,\overline{p} \\
 & & \mid & \texttt{if}\ p\ \texttt{then}\ p\ \texttt{else}\ p \\
\textbf{\textit{Integers}} & n & ::= & 0, -1, 1, \ldots \\
\textbf{\textit{Booleans}} & b & ::= & \texttt{True} \mid \texttt{False} \\
\textbf{\textit{Binary Ops}} & \bowtie & ::= & = \mid < \mid \wedge \mid + \mid - \mid \ldots \\
\textbf{\textit{Unary Ops}} & \oplus_1 & ::= & \texttt{!} \mid \ldots \\
\textbf{\textit{Sort Args}} & s_a & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{U} \mid \texttt{Fun}\ s_a\ s_a \\
\textbf{\textit{Sorts}} & s & ::= & s_a \mid s_a \rightarrow s
\end{array}
$$

Fig. 7. **Syntax of λ^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 [Supplementary-Material 2017]. Here we discuss the non-identity ones.

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

$$
\begin{array}{llll}
(\!|\texttt{Int}|\!) \doteq \texttt{Int} & (\!|T|\!) \doteq \texttt{U} & (\!|\{v : B^{[\Downarrow]} \mid e\}|\!) \doteq (\!|B|\!) \\
(\!|\texttt{Bool}|\!) \doteq \texttt{Bool} & & (\!|x{:}\tau_x \rightarrow \tau|\!) \doteq \texttt{Fun}\ (\!|\tau_x|\!)\ (\!|\tau|\!)
\end{array}
$$

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`.

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

The term $\lambda x.e$ of type $\tau_x \rightarrow \tau$ is transformed to $\texttt{lam}^{s_x}_{s}\ x\ p$ of sort $\texttt{Fun}\ s_x\ s$, where s_x and s are respectively $(\!|\tau_x|\!)$ and $(\!|\tau|\!)$, $\texttt{lam}^{s_x}_{s}$ is a special uninterpreted function of sort $s_x \rightarrow s \rightarrow \texttt{Fun}\ s_x\ s$, and x of sort s_x and r of sort s are the embedding of the binder and body, respectively. As `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 embed applications via defunctionalization [Reynolds 1972] using the uninterpreted app:

$$
\frac{\Gamma \vdash e' \rightsquigarrow p' \qquad \Gamma \vdash e \rightsquigarrow p \qquad \Gamma; \Psi \vdash e : \tau_x \rightarrow \tau}{\Gamma \vdash e\ e' \rightsquigarrow \texttt{app}^{(\!|\tau_x|\!)}_{(\!|\tau|\!)}\ p\ p'}
$$

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

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

```
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 1981]. 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 $\mathtt{Fun}\ (\!|\tau|\!)\ \mathtt{Bool}$ and *select* $\mathtt{sel}_{D_{i,j}}$ has the sort $\mathtt{Fun}\ (\!|\tau|\!)\ (\!|\tau_{i,j}|\!)$.

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 6 algorithmic by checking subtyping via our novel, SMT-based *Proof by Logical Evaluation*(PLE). Next, we formalize how PLE makes checking algorithmic, and then describe it in detail in § 6.

Verification Conditions The implication or *verification condition* (VC) $\lfloor \Gamma \rfloor \Rightarrow p$ is *valid* only if the set of values described by Γ is subsumed by the set of values described by p. Γ is embedded into logic by conjoining the refinements of provably terminating binders [Vazou et al. 2014]:

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

Validity Checking Given a reflection environment R, type environment Γ, and expression e, the procedure $\mathsf{PLE}(\lfloor R \rfloor, \lfloor \Gamma \rfloor, \lfloor e \rfloor)$, 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 version $\preceq$-Base-PBE that uses PLE to check the validity of the subtyping VC.

$$\frac{\mathsf{PLE}(\lfloor R \rfloor, \lfloor \Gamma, v : \{v : B^{\Downarrow} \mid e\} \rfloor, \lfloor e' \rfloor)}{\Gamma; R \vdash_{PBE} \{v : B \mid e\} \preceq \{v : B \mid e'\}}\ \preceq\text{-Base-PBE}$$

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

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

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

Theorem 5.2 (Soundness of λ^S). *If* $\Gamma; R \vdash_S e : \tau$ *then* $\Gamma; R \vdash e : \tau$.

$$
\begin{array}{rrcl}
\textbf{\textit{Terms}} & p, t, b & ::= & \lambda^S \texttt{ if}\text{-free predicates from Figure 7} \\
\textbf{\textit{Functions}} & F & ::= & \lambda \overline{x}.\langle \overline{p \Rightarrow b} \rangle \\
\textbf{\textit{Definitional Environment}} & \Psi & ::= & \emptyset \mid f \mapsto F, \Psi \\
\textbf{\textit{Logical Environment}} & \Phi & ::= & \emptyset \mid p, \Phi
\end{array}
$$

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

Unfold	:	$(\Psi, \Phi) \rightarrow \Phi$
$\mathsf{Unfold}(\Psi,\ \Phi)$	=	$\Phi \cup \bigcup_{f(\overline{t}) < \Phi} \mathsf{Instantiate}\left(\Psi, \Phi, f, \overline{t}\right)$
$\mathsf{Instantiate}\left(\Psi, \Phi, f, \overline{t}\right)$	=	$\left\{ (\lfloor f(\overline{x}) \rfloor = b_i)\left[\overline{t}/\overline{x}\right] \;\middle|\; (p_i \Rightarrow b_i) \in \overline{d},\ \ \mathsf{SmtValid}(\Phi, p_i\left[\overline{t}/\overline{x}\right]) \right\}$
where		
$\lambda \overline{x}.\langle \overline{d} \rangle$	=	$\Psi(f)$
PLE	:	$(\Psi, \Phi, p) \rightarrow \mathsf{Bool}$
$\mathsf{PLE}(\Psi, \Phi, p)$	=	$\mathsf{loop}\left(0,\ \Phi \cup \bigcup_{f(\overline{t}) < p} \mathsf{Instantiate}\left(\Psi, \Phi, f, \overline{t}\right)\right)$
where		
$\mathsf{loop}\ (i, \Phi_i)$		
$\mid\ \mathsf{SmtValid}(\Phi_i, p)$	=	*true*
$\mid\ \Phi_{i+1} \subseteq \Phi$	=	*false*
$\mid$ otherwise	=	$\mathsf{loop}\ (i+1, \Phi_{i+1})$
where		
Φ_{i+1}	=	$\Phi \cup \mathsf{Unfold}(\Psi,\ \Phi_i)$

Fig. 9. Algorithm PLE: Proof by Logical Evaluation

6 COMPLETE VERIFICATION: PROOF BY LOGICAL EVALUATION

Next, we formalize our Proof By Logical Evaluation PLE algorithm 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 PBE. The logical environment Φ contains a set of hypotheses p, described in Figure 7. 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 `DeIf`$(\cdot)$, found in Appendix **??**. AT ♣ broken reference ♣ A definitional environment Ψ can be constructed from R as

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

Notation We write $f(\overline{t}) \prec \Phi$ if the λ^S term $(\text{app} \ldots (\text{app } f\ t_1) \ldots t_n)$ is a syntactic subterm of some $t' \in \Phi$. We abuse notation to write $f(\overline{t}) \prec t'$ for $f(\overline{t}) \prec \{t'\}$. We write $\mathsf{SmtValid}(\Phi, p)$ 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:

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

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.

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

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

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 PBE 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. Let $R[e]$ denote the evaluation of e under the reflection environment R, *i.e.* $\emptyset[e] \doteq e$ and $(R, f : e_f)[e] \doteq R[\text{let rec } f = e_f \text{ in } e]$.

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

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 1981].

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 will 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 will show that the fixpoint is computable: there is an unfolding depth at which PLE reaches a fixpoint and hence terminates.

Bounded Unfolding For every 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 and 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* $\forall n < m.\ \Phi_m = \Phi_n$.

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

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

$$\frac{\Psi, \Phi \vdash t_1 \twoheadrightarrow t_1' \quad \Psi, \Phi \vdash t_2 \twoheadrightarrow t_2' \quad \mathsf{SmtValid}(\Phi, t_1' \bowtie t_2')}{\Psi, \Phi \vdash t \bowtie t'} \text{ Eq-Proof}$$

Fig. 10. Equational Proofs

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. AT ♣ I think this is unclear just because of the word "uncovered". Maybe $\cdots$ that is *uncovered* (ie found) by or even more explanation ♣

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* $\mathsf{SmtValid}(\Phi, v = t, v = t')$ *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 Next, 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 a possible 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 verifiers 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 Next, 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*** Next, we show that the proof search procedure PLE computes the least-fixpoint of the bounded unfolding and hence, returns *true* iff there exists *some* unfolding depth n at which the goal can be proved.

Lemma 6.9 (**Fixpoint**). $\mathsf{PLE}(\Psi, \Phi, t = t')$ *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 can 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 By combining Lemma 6.9 and Lemma 6.7 we can show that PLE is complete, *i.e.* if there is an equational proof that $t \bowtie t'$ 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, next, we show that in addition to being sound and complete with respect to equational proofs, if the hypotheses are transparent, then our proof search procedure always terminates. Next, we describe transparency and explain intuitively why PLE terminates. We then develop the formalism needed to prove the termination theorem 6.16.

Transparency An environment Γ is *inconsistent* if $\mathsf{SmtValid}(\lfloor \Gamma \rfloor, \mathit{false})$. 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 argument in three parts. First, we formalize how PLE performs computation at the logical level via *logical steps* and *logical traces*. We show (Lemma 6.13) that the logical steps form a so-called *universal* (or *"must"*) abstraction of the concrete semantics [Clarke et al. 1992; Cousot and Cousot 1977]. Second, we show that if PLE diverges, it is because it creates a strictly increasing infinite chain, $\mathsf{Unfold}^*(\Psi, \Phi, 0) \subset \mathsf{Unfold}^*(\Psi, \Phi, 1) \ldots$ which corresponds to an *infinite logical trace*. Third, as the logical computation is 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, and hence, PLE must terminate. Next, we formalize the above intuition 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 [Vazou et al. 2014]. Hence, for brevity, in the sequel we will *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 $\theta \cdot R[p_i] \hookrightarrow^\star \texttt{True}$ then $\theta \cdot R[e_i] \hookrightarrow^\star v$.
(2) If $\theta \cdot R[p_i] \hookrightarrow^\star \texttt{True}$ and $\theta \cdot \Psi[p_j] \hookrightarrow^\star \texttt{True}$, then $i = j$.
(3) There exists an i so that $\theta \cdot R[p_i] \hookrightarrow^\star \texttt{True}$.

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 $\texttt{DeIf}(\cdot)$ 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* $b \equiv \lambda\overline{x}.\langle\overline{\lfloor p \rfloor \Rightarrow \lfloor e \rfloor}\rangle$. *For every* Γ, R, *and* $\theta \in (\!|\Gamma|\!)$, *if* $\theta \cdot R[p_i] \hookrightarrow^\star \texttt{True}$ *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

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

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 $\overline{t_0} \equiv \overline{x_0}$ for some variables $\overline{x_0}$, and $f_i(\overline{t_i}) \rightsquigarrow f_{i+1}(\overline{t_{i+1}})$ 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 fact that Γ is inhabited yields *at least one infinite concrete trace*, which contradicts totality. 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.*

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* $\Phi_i \doteq \text{Unfold}^*(\Psi, \Phi, i)$. *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* R *is transparent then* $\text{PLE}(\lfloor R \rfloor, \lfloor \Gamma \rfloor, p)$ *terminates.*

Benchmark	Common		Equational Proof			PLE Proof		
	Impl	Spec	Time	SMT	Proof	Time	SMT	Proof
Arithmetic								
Fibonacci	7	10	2.74	129	38	1.92	79	16
Ackermann	20	73	5.40	566	196	13.77	846	119
Class Laws Fig 11								
Monoid	33	50	4.47	34	109	4.22	209	33
Functor	48	44	4.97	26	93	3.68	68	14
Applicative	62	110	11.98	69	241	10.00	1090	74
Monad	63	42	5.39	49	122	4.89	250	39
Higher-Order Properties								
Logical Properties	0	20	2.71	32	33	2.74	32	33
Fold Universal	10	44	2.17	24	43	1.46	48	14
Functional Correctness								
SAT-solver	92	34	50.00	50	0	50.00	50	0
Unification	51	60	4.77	195	85	5.64	422	21
Deterministic Parallelism								
Conc. Sets	0	0	0	0	0	0	0	0
n-body	0	0	0	0	0	0	0	0
Par. Reducers	0	0	0	0	0	0	0	0
Total	333	442	38.13	566	876	34.62	561	339

Table 1. Quantitative evaluation. We report verification **Time** (in seconds), the number of **SMT-queries** and LoC required to prove the respective properties. We split proofs Liquid Haskell (843 LoC in total) and Liquid Haskell with PLE (306 LoC in total) into **specifications**, **proof terms** and **executable** code. All benchmarks were run on a 2.3GHz Xeon E5-2699.

7 EVALUATION

We have implemented reflection and PLE in Liquid Haskell [Vazou et al. 2014]. Table 1 summarizes our empirical evaluation which seeks to (1) describe the programs and properties that can be verified, (2) measure how PLE simplifies *writing* proofs, (3) measure the PLE affects the time taken to *verify* proofs.

Benchmarks To evaluate our approach we use it to implement and verify a wide variety of programs.

- **Arithmetic** We proved arithmetic properties for the textbook Fibonacci function (c.f. § 2), and 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 text-book propositional properties from [Gentzen 1935] as described in 3. We combined natural deduction principles with PLE automatic equational reasoning to prove universality of right-folds, as described in [Hutton 1999] and formalized in Adga [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]. By reflecting the notion of `sat`isfaction, we prove that the SAT `solver` takes as input a formula `f` and either returns `Nothing` or an assignment that `sat`isfies `f`. While the unification function can itself diverge, and hence, cannot be reflected, our method allows terminating and diverging functions to soundly coexist. Consequently, we prove 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.
- **Deterministic Parallelism** Retrofitting complete specification and verification onto an existing language with a mature parallel run-time, reflection allows us to create three deterministic parallelism libraries that, for the first time, actually verify assumptions about associativity and ordering that are critical for determinism. First, we proved that the *ordering laws* hold for keys being inserted into LV-ish style concurrent sets [Kuper

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	$(m \gg\!= f) \gg\!= g \equiv m \gg\!= (\lambda x \rightarrow f\ x \gg\!= g)$
Inter.	$u \circledast \texttt{pure}\ y \equiv \texttt{pure}\ (\$\ y) \circledast u$		
Ord (for Int, Double, Either, (,))		**Abelian Monoid** (for Int, Double, (,))	
Refl.	$x \leq x$	Comm.	$x \diamond y \equiv y \diamond x$
Antisym.	$x \leq y \wedge y \leq x \implies x \equiv y$		
Trans.	$x \leq y \wedge y \leq z \implies x \leq z$		
Total.	$x \leq y \vee y \leq x$		

Fig. 11. **Summary of Verified Typeclass Laws**

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 arguments being reduced formed a `CommutativeMonoid`, and we used this to implement 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`). We further compare the lines needed for explicit *equational* proofs (*i.e. without* proof-search), as well as with PLE-proofs (*i.e. with* proof-search.)

Results The main highlights of our evaluation are the following. (1) Reflection allows for the specification and verification of a wide variety of important properties of programs. (2) PLE drastically reduces the proof effort: by a factor of $2-5\times$, making it quite modest, about the size of the specifications of the theorems. (3) PLE does not impose a performance penalty: even though proof search can make an order of magnitude many more SMT queries, most of these queries are simple and it typically *faster* 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 1981]. Leon introduces an SMT-based algorithm that is complete for catamorphisms (folds) over ADTs [Suter et al. 2010]. Our work is inspired by Dafny's Verified Calculations [Leino and Polikarpova 2016], and a framework for proving theorems in Dafny [Leino 2010], but differs in (1) our use of reflection instead of axiomatization and (2) our use of refinements to compose proofs. (3) our use of PLE to automate reasoning about user-defined functions. Dafny (and the related 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, in practice be more expedient to quickly time out after a fixed, small number of instantiations rather than to 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.

Proof Assistants Reflection shows how deep specification and verification in the style of Coq [Bertot and Castéran 2004] and Agda [Norell 2007] can be retrofitted 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). Of course, mature proof assistants like Agda, Coq, and Isabelle have two advantages over our approach: they emit certificates and hence have a small trusted computing base, and they have decades-worth of tactics, libraries and proof scripts that enable large scale proof engineering. Tactics 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 integrating tactics and scriptable proof search within SMT-based verifiers remains an important direction for future work.

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". Unlike in our work, the programmer must put in explicit `join` terms to indicate where normalization should be triggered, and even so, equality checking is based on "fuel" and hence, incomplete.

Proving Equational Properties Several authors have proposed tools for proving (equational) properties of (functional) programs. Systems 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.

9 CONCLUSIONS AND FUTURE WORK

We have shown how refinement reflection—namely reflecting the definitions of functions in their output refinements—can be used to convert a legacy programming language, like Haskell, into a theorem prover. Reflection ensures that (refinement) type checking stays decidable and predictable via careful design of the logic and proof combinators. Reflection enables programmers working with the highly tuned libraries, compilers and run-times of legacy languages to specify and verify arbitrary properties of their code simply by writing programs in the legacy language. Our evaluation shows that refinement reflection lets us prove deep specifications of a variety of implementations, for example, the determinism of fast parallel programming libraries, and also identifies two important avenues for research.

First, while our proofs are often elegant and readable, they can sometimes be cumbersome. For example, in the proof of associativity of the monadic bind operator for the `Reader` monad three out of eight (extensional) equalities required explanations, some nested under multiple λ-abstractions. Thus, it would be valuable to explore how to extend the notions of tactics, proof search, and automation to the setting of legacy languages. Similarly, while our approach to α- and β-equivalence is sound, we do not know if it is *complete*. We conjecture it is, due to the fact that our refinement terms are from the simply typed lambda calculus (STLC). Thus, it would be interesting to use the normalization of STLC to develop a sound and complete system for SMT-based type-level computation and use it to automate proofs predictably.

Second, while Haskell's separation of pure and effectful code undoubtedly makes it easier to implement refinement reflection, we believe that the technique, like refinement typing in general, is orthogonal to purity. In particular, by carefully mediating between the interaction of pure and impure code using methods like permissions, uniqueness *etc.* refinement type systems have been developed for legacy languages like C [Rondon

et al. 2010], JavaScript [Chugh et al. 2012; Vekris et al. 2016], and Racket [Kent et al. 2016] and so it would be interesting to see how to extend refinement reflection to other legacy languages.

REFERENCES

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

Andrew 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.

Richard S. Bird. 1989. Algebraic Identities for Program Calculation. *Comput. J.* 32, 2 (1989), 122–126. https://doi.org/10.1093/comjnl/32.2.122

Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. 2011. Extending Sledgehammer with SMT Solvers. In *Proceedings of the 23rd International Conference on Automated Deduction (CADE'11)*. Springer-Verlag, Berlin, Heidelberg, 116–130. http://dl.acm.org/citation.cfm?id=2032266.2032277

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

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

Juan Chen, Ravi Chugh, and Nikhil Swamy. 2010. Type-preserving Compilation of End-to-end Verification of Security Enforcement. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 412–423. https://doi.org/10.1145/1806596.1806643

R. Chugh, D. Herman, and R. Jhala. 2012. Dependent Types for JavaScript. In *OOPLSA*.

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

E. M. Clarke, O. Grumberg, and D.E. Long. 1992. Model checking and abstraction. In *POPL 92: Principles of Programming Languages*. ACM, 343–354.

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 77*. ACM, 238–252.

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

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (Aug. 1975), 453–457. https://doi.org/10.1145/360933.360975

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 *(Haskell)*.

G. Gentzen. 1935. Inverstigations into Logical Deduction. (1935).

W. A. Howard. 1980. The formulae-as-types notion of construction. (1980). http://lecomte.al.free.fr/ressources/PARIS8_LSL/Howard80.pdf

Graham 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. *TOPLAS* (2010).

Lindsey Kuper, Aaron Turon, Neelakantan R Krishnaswami, and Ryan 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 *(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*.

Rustan 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. *J. Funct. Program.* (2009).

G.C. Necula. 1997. Proof carrying code. In *POPL 97: Principles of Programming Languages*. ACM, 106–119.
G. Nelson. 1981. *Techniques for program verification.* Technical Report CSL81-10. Xerox Palo Alto Research Center.
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 *25th 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. *IEEE TSE* (1998).
Eric L. Seidel, Niki Vazou, and Ranjit Jhala. 2015. Type Targeted Testing. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*. Springer-Verlag New York, Inc., New York, NY, USA, 812–836. https://doi.org/10.1007/978-3-662-46669-8_33
V. Sjöberg and S. Weirich. 2015. Programming Up to Congruence. *POPL* (2015).
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*.
Non-Anonymous Supplementary-Material. 2017. Supplementary Material. (2017).
Philippe Suter, Mirco Dotta, and Viktor Kuncak. 2010. Decision Procedures for Algebraic Data Types with Abstractions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 199–210. https://doi.org/10.1145/1706299.1706325
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. AckermannâĂŹs Function. (2008). http://www.cs.yorku.ca/~gt/papers/Ackermann-function.pdf.
N. Vazou, P. Rondon, and R. Jhala. 2013. Abstract Refinement Types. In *ESOP*.
N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. 2014. Refinement Types for Haskell. In *ICFP*.
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*.
Philip Wadler. 1987. A Critique of Abelson and Sussman or Why Calculating is Better Than Scheming. *SIGPLAN Not.* 22, 3 (March 1987), 83–94. https://doi.org/10.1145/24697.24706
Philip Wadler. 2015. Propositions As Types. *Commun. ACM* (2015).
H. Xi and F. Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types.. In *PLDI*.