Towards Complete Specification and Verification with SMT

ANONYMOUS AUTHOR(S)

ACM Reference format:
Anonymous Author(s). 2017. Towards Complete Specification and Verification with SMT. *Proc. ACM Program. Lang.* 1, 1, Article 1 (July 2017), 30 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 query 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. Unfortunately, these verifiers require the user to supply lemmas or rewrite hints that add friction to the ubiquitous task of discharging simple queries over decidable theories.

RN ♣ As a reader, I don't have that firm a notion of what the claimed completeness is at this point. ♣

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].

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).

We show how to represent proofs simply as unit-values refined by the proposition that they prove, and we develop a *library of combinators* that lets programmers compose sophisticated *equational proofs* from basic refinements and function definitions. Our proof combinators let programmers use standard 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).

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

2. Complete Specification Furthermore, since proofs are literally just programs, our second contribution is to demonstrate, via the Curry-Howard Isomorphism, that Refinement Reflection yields a proof system that is equivalent to (§ 3) RJ ♣ SAY MORE. We show that ZZZ. ♣

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* (PBE) NV:♣ replace pbe with ple?♣ a new proof-search algorithm that completely automates equational reasoning. The key idea in PBE 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* abstraction of the concrete semantics of the user-defined functions. Thus, as those functions are total and terminating, we obtain the pleasant guarantee that the PBE proof search will terminate.

4. Evaluation We evaluate our approach by implementing refinement reflection and PBE in Liquid Haskell [Vazou et al. 2014], thereby turning Haskell into a theorem prover. 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 — in a set of benchmarks totalling RJ ♣ XXX ♣ lines — how reflection and PBE can be used to verify that widely used instances of the Monoid, Applicative, Functor, and Monad typeclasses satisfy the respective laws, making the use of these typeclasses safe (§ 7).

Taken together, 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, permits principled and automated specification and verification over both built-in theories and user-defined code.

2 OVERVIEW

We start with an overview of how SMT-based refinement reflection lets us write equational proofs as plain functions and how PBE 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`, drawn 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, we will use the textbook Fibonacci function to demonstrate the proof features we add to Liquid Haskell, 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, 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. 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 $\wedge$ and $\vee$). 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)

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

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

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 :: Int → Int` 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* $\forall n, m.\ n = m \Rightarrow \texttt{fib}\ n = \texttt{fib}\ m$. On its own, the uninterpreted function is not terribly useful, as it does not let us prove `{fib 2 = 1}` which requires reasoning about the *definition* of `fib`.

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 `f` to denote 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' :: { 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* (PBE) 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 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.

"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
```

RN ♣ Why are some fib calls highlighted and others not? Should they all be? ♣

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
```

NV:♣ This does not explain inductive proofs, either update the text of the proof♣ RN ♣ Umm, maybe we could make this a literate document where the code runs? The above looks wrong. It says that `fib n = fib (n+1)` on the last line. ♣ ***Case Splitting and Induction*** The proof `fibUp` works by induction on `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. In the *inductive* case, `fib n` is unfolded to `fib (n-1) + fib (n-2)`, which, because of the induction hypothesis (applied by invoking `fibUp` at `n-1` and `n-2`) and the SMT solver's arithmetic reasoning, completes the proof.

Higher Order Theorems Refinement reflection smoothly accomodates 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 a proof search algorithm PBE that is inspired by model checking [Clarke et al. 1992] and works as follows.

- ***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. 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.

RN ♣ At this point not sure if this fixpoint iteration is an outer loop repeating SMT invocation on each iteration. ♣

Automatic Equational Reasoning In § 6 we formalize a notion of *equational proof* and show that the proof search procedure PBE 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 PBE to predictably automate proofs: the programmer needs only supply the relevant induction hypotheses or helper lemma applications. The remaining long chains of calculations are performed automatically via SMT-based PBE. To wit, with complete proof search, the above proofs shrink to:

```
fib3_2 :: {fib 3 = 2}   fibUp  :: Up fib   fMono :: Mono / [y]
fib3_2 = ()             fibUp 0 = ()       fMono f up x y
                        fibUp 1 = ()         | x+1 == y = up x
                        fibUp n = ()         | x+1  < y = up (y-1) &&& fMono up x (y-1)
```

where the proof combinators `x &&& y = ()` simply inserts the two proof arguments in the VC environment.

PBE ***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]

```
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) PBE proof, also of append-id and map-fusion.

```
pos n | n < 0     = 0
      | otherwise = 1 + pos (n-1)
```

```
test   :: y:{y > 5} → {pos n = 3 + pos (n-3)}
test _ = ()
```

RN ♣ Ugh, let's make sure the above 2x2 matrix of code doesn't get page-broken in the final version; its terrible. ♣ 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, PBE (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 PBE are not limited to integers. We end the overview by showing how to they verify textbook properties of lists equipped with append (++) and `map` functions:

```
(++)    :: [a] → [a] → [a]
[]      ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

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

In § 5.1 we describe how the reflection mechanism illustrated via `fibP` is extended to account for ADTs. NV:♣ Thus the case translation cannot go away♣ 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 PBE, which should be compared to the classical equational proof *e.g.* [Wadler 1987] shown on the left. With PBE, the user need only provide the high-level structure – the case splits and invocations of the induction hypotheses – after which PBE automatically completes the rest of the equational proof. Thus using SMT-based PBE, `app_assoc` shrinks down to its essence: an induction over the list `xs`. The

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
```

PBE

```
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 PBE.

difference is even more stark with `map_fusion` whose full equational proof we don't show, as it is twice as long.

NV:♣ This paragraph is related work maybe move there?♣ NV:♣ I suggest moving it there because it is not accurate♣ NV:♣ Agda does not have any tactics/ automation and I bet this proof can be scimplified in Coq using a Presburger Arithmetic solver omega♣ NV:♣ Lets instead cite my string matcher for a comparison♣ PBE ***vs. Normalization*** RJ ♣ Vikraman read carefully! ♣

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 PBE 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 PBE algorithm shows how to *emulate* (the spirit!) of those ideas by asserting equalities corresponding to function definitions. Second, from a *practical* perspective, the combination of PBE 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) | x1 > x2   = x2:x1:x2
                | otherwise = 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 `omega`), and variant in Agda and finally the PBE proof. It is readily apparent that PBE's combination of proof search working hand-in-glove with SMT-based theory reasoning makes proving the result relatively trivial. Of course, the decades-worth of tactics, libraries and proof scripts available in Agda, Coq, Isabelle etc. enable large scale proof engineering that is well beyond what is currently feasible with our approach. We merely use this example it to illustrate that reflection and SMT-based proof search bring powerful, complete new tools for specification and verification.

3 COMPLETE SPECIFICATION

In this section we present that the restriction of the refinement language to be quantifier free, crucial for SMT-decidable type checking, does not pose any expressiveness restrictions to our

ϕ	::=		*Formulas:*
		$\{e\}$	*native terms, with* `True`, `False` $\in e$
	|	$\phi_1 \rightarrow \phi_2$	*implication:* $\phi_1 \Rightarrow \phi_2$
	|	$\phi \rightarrow \{\texttt{False}\}$	*negation:* $!\phi$
	|	`PAnd` ϕ_1 ϕ_2	*conjunction:* $\phi_1 \wedge \phi_2$
	|	`POr` ϕ_1 ϕ_2	*disjunction:* $\phi_1 \vee \phi_2$
	|	$x : a \rightarrow \phi$	*forall:* $\forall x.\phi$
	|	$(x :: a, \phi)$	*exists:* $\exists x.\phi$

Fig. 3. Encoding of higher-order specifications in using quantifier-free refinement types. $\{e\}$ simplifies $\{v : \texttt{Prop} \mid e\}$. Function binders are not represented in negation and implication where they are not relevant.

specifications. Instead higher-order specifications can be naturally encoded using λ-abstractions and dependent pairs to encode universal and existential quantification [Howard 1980] respectively.

3.1 The specification language

Figure 3 summarizes the encoding of higher order specifications using first-order (*i.e.* quantifier-free refinements). This encoding is syntactically equivalent to the specifications of Isabelle/HOL [Wenzel 2016].

Encoding of native terms The logical terms in Liquid Haskell are Haskell expressions e as presented in Figure 1. AT: broken external refs here? These expressions include constants, boolean operations, lambda abstractions, applications, and in practice are extended to include decidable SMT theories, including quantifier-free linear arithmetic and set theory (hereafter referred to as the 'native' logic) In the absence of reflected functions, reasoning over terms from this native (SMT-decidable) logic is automatically performed by the SMT solver. Otherwise, when terms include reflected functions, reasoning is performed by reflection of function definitions into type level computation, either in equational proofs by the user, or by our algorithm outlined in Section 6. These reflected functions are encoded inside the native logic as uninterpred functions with axioms, as described in sections 4.4 and 5.

Encoding of first-order 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 simply `False`. Conjunction $\phi_1 \wedge \phi_2$ is encoded with the data type `PAnd` that contains the proofs of the two conjuncts.

```
data PAnd a b = PAnd a b
```

Disjunction $\phi_1 \vee \phi_2$ is encoded with the sum type `POr` that contains the proofs of one of the two disjuncts.

```
data POr a b = POrLeft a | POrLeft b
```

Encoding of quantifiers Universal quantification $\forall x.\phi$ is encoded and introduced as a lambda abstraction $x : a \rightarrow \phi$ and instantiation 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. Dependent pairs can be implemented using abstract refinement types [Vazou et al. 2013]. without adding complexity to type checking.

Refinement & Reification of first order connectives. For first-order specifications, potentially derived after instantiations of the higher-order specification of Figure 3 we define, for each connective, a refinement and a reification operator that maps speficiations of Figure 3 to a quantifier-free refinement type and back. For instance, we refine conjunction by performing case analysis on the `PAnd {b1} {b2}` constructor therefore bringing both the conjuncts into the environment leading to the VC $b1 \wedge b2 \Rightarrow b1 \wedge b2$ that can be discharged by the SMT-solver.

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

We reify conjuction by using $\phi_1 \wedge \phi_2$ as a proof for each conjunct ϕ_1 and ϕ_2.

```
andReify :: b1:Bool → b2:Bool → {b1 && b2} → PAnd {b1} {b2}
andReify _ _ b = PAnd b b
```

3.2 Proof Terms: Encoding Natural Deduction

Due to Curry-Howard isomorphism there is an "1-1" correspondence between any higher-order natural deduction proof [Gentzen 1935] and terms of simply typed lambda calculus. Thus, we can naturally use Haskell terms to encode natural deduction proofs for the higher-order specifications ϕ. Since proofs are encoded as Haskell terms, specifications as refinement types we use Liquid Haskell to ensure that the Haskell terms indeed prove the specifications. We need to ensure that Liquid Haskell type checking is precise enough to accept the natural deduction rules. We do so by representing each natural deduction rule as a specification ϕ and providing a Haskell term that satisfies ϕ.

Modus Ponens We prove modus ponens (or implication elimination) as function application.

```
implElim :: p → (p → q) → q
implElim p f = f p
```

Conjunction The conjunction introduction rule proves that assuming two proofs for `b1` and `b2` we deduce a proof for `PAnd {b1} {b2}`.

```
andIntro :: b1:{Bool | b1} → b2:{Bool | b2} → PAnd {b1} {b2}
andIntro b1 b2 = PAnd () ()
```

We eliminate conjuction by returning the left or the right conjuncts.

```
andElimL :: b1:Bool → b2:Bool
         → PAnd {b1} {b2} → {b1}
andElimL _ _ (PAnd b1 b2) = b1
```

```
andElimR :: b1:Bool → b2:Bool
         → PAnd {b1} {b2} → {b2}
andElimR _ _ (PAnd b1 b2) = b2
```

Universal Quantification Universal elimination is encoded as function application while introduction in an identity.

```
forallElim :: p:(a → Bool) → (x:a → {p x} ) → y:a → {p y}
forallElim _ f y = f y

forallIntro :: p:(a → Bool) → (t:a → {p t}) → (x:a → {p x})
forallIntro _ f = f
```

AT: let's be careful with the language here. Have to make sure not to mix up the language for univ elimination with implication refinement

Existential Quantification Existential introduction and elimination are encoded as packing and unpacking the existential pair.

```
existsIntro :: a → b → (a,b)
existsIntro x px = (x, px)

existsElim :: (a,b) → (a → b → t) → t
existsElim (t, pt) f = f t pt
```

3.3 Examples

Next we present how the quantified formulas can be used to express textbook higher-order logical properties, properties over user-defined domains (here lists), and conclude by encoding and proving induction on integers.

3.4 Logical Properties

Existentials over disjunction We start by distributing existentials over disjunction:

$$\forall p\ q\ .(\exists x.(p\ x \vee q\ x)) \Rightarrow ((\exists x.p\ x) \vee (\exists x.q\ x)))$$

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, POr {p x} {q x})
              → POr (x::a, {p x}) (x::a, {q x})
existsOrDistr _ _ (x,POrLeft  px) = POrLeft  (x,px)
existsOrDistr _ _ (x,POrRight qx) = POrRight (x,qx)
```

Foralls over conjunction Similarly, we distribute foralls over conjunction:

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

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 PAnd data constructor.

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

3.4.1 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 we prove that forall lists `xs` if there exists a list `ys` so that `xs == ys ++ ys` then `xs` has even length.

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

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)
```

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

Fig. 4. **Syntax of λ^R**

```
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 PBE is used to simplify the trivial unfoldings.

3.4.2 Induction on Natural Numbers. As the last example, we use Liquid Haskell to specify and verify induction on natural numbers.

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

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

4 REFINEMENT REFLECTION: λ^R

NV:♣ Add exact reflected definition of fib ♣ 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 strengthen proof obligations with complete unfoldings of the reflected functions, to automate equational reasoning.

4.1 Syntax

Figure 4 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 layered into primitive constants, values, expressions, binders and programs.

Constants The primitive constants of λ^R include primitive relations $\oplus$, here, the set $\{=, <\}$. Moreover, they include the primitive 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 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 extra termination-check constraints imposed by the type system:

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

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

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

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

4.3 Types

λ^R types include basic types, which are *refined* with predicates, and dependent function types. *Basic types* B comprise integers, booleans, and a family of data-types T (representing lists, trees *etc.*). For example, the data type $[Int]$ represents lists of integers. We refine basic types with predicates (boolean-valued expressions e) to obtain *basic refinement types* $\{v : B \mid e\}$. We use $\Downarrow$ to mark provably terminating computations and use refinements to ensure that if e:$\{v : B^\Downarrow \mid e'\}$, then e terminates. As discussed by Vazou et al. [2014] termination labels 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:

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

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

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

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 5).

Reflection We write $\mathsf{Reflect}(\tau, e)$ for the *reflection* of the term e into the type τ, defined by strengthening τ 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 $\lambda n.e_{\texttt{fib}}$.

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

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

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

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 5 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.

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

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

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

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

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

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

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

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

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

$$\frac{\forall \theta \in [\![\Gamma]\!].[\![\theta \cdot \{v : B \mid e_1\}]\!] \subseteq [\![\theta \cdot \{v : B \mid e_2\}]\!]}{\Gamma; R \vdash \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}}\ \preceq\textsc{-Base-}\lambda^R \qquad \frac{\Gamma; R \vdash \tau_x' \preceq \tau_x \qquad \Gamma, x : \tau_x'; R \vdash \tau \preceq \tau'}{\Gamma; R \vdash x{:}\tau_x \to \tau \preceq x{:}\tau_x' \to \tau'}\ \preceq\textsc{-Fun}$$

Fig. 5. Typing of λ^R

4.5 Typing Rules

Next, we present the type-checking rules of λ^R.

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:

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

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:

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

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 [Knowles and Flanagan 2010; Vazou et al. 2014] refinement typing rules that moreover preserve the reflection environment $RRenv$. 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 $\{v : \texttt{Int}^{\Downarrow} \mid \texttt{fibP}\ v\ 1 \wedge v = \texttt{fib}\ 1\}$ 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

RJ ♣ TODO: Shrink this section and move to APPENDIX. ♣

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-λ^S, yielding an SMT-based algorithmic type system that is both sound and decidable.

Syntax: Terms & Sorts Figure 6 summarizes the syntax of λ^S, the *sorted* (SMT-) decidable logic of quantifier-free equality, uninterpreted functions and linear arithmetic (QF-EUFLIA) [Barrett et al. 2010; Nelson 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 $\oplus_2$ 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.

$$
\begin{array}{rrcl}
\textbf{\textit{Predicates}} & p & ::= & p \oplus_2 p \mid \oplus_1 p \\
 & & \mid & n \mid b \mid x \mid D \mid x\,\overline{p} \\
 & & \mid & \texttt{if}\ p\ \texttt{then}\ p\ \texttt{else}\ p \\
\textbf{\textit{Integers}} & n & ::= & 0, -1, 1, \ldots \\
\textbf{\textit{Booleans}} & b & ::= & \texttt{True} \mid \texttt{False} \\
\textbf{\textit{Binary Ops}} & \oplus_2 & ::= & = \mid < \mid \wedge \mid + \mid - \mid \ldots \\
\textbf{\textit{Unary Ops}} & \oplus_1 & ::= & \texttt{!} \mid \ldots \\
\textbf{\textit{Sort Args}} & s_a & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{U} \mid \texttt{Fun}\ s_a\ s_a \\
\textbf{\textit{Sorts}} & s & ::= & s_a \mid s_a \rightarrow s
\end{array}
$$

Fig. 6. **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. 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:

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

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 λ-abstraction using the uninterpreted function `lam`.

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

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* name pre-declared in the SMT logic.

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 translate each data constructor 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 assume the existence of reflected functions that *check* the top-level constructor and *select* their individual fields. For example, for lists, we assume the existence of measures:

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

Due to the simplicity of their syntax the above checkers and selectors can be automatically instantiated in the logic (*i.e.* without actual calls to the reflected functions at source level) using the measure mechanism of Vazou et al. [2015].

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 the *check function* $\mathtt{is}_{D_i}$ has the sort $\mathtt{Fun}\ (\!|\tau|\!)\ \mathtt{Bool}$ and the *select function* $\mathtt{sel}_{D_{i,j}}$ 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.

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

For example, the body of the list append function

```
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

is reflected into the λ^S refinement: `if isNil` *xs* `then` *ys* `else sel1` *xs* `:` (`sel2` *xs* `++` *ys*). We favor selectors to the axiomatic translation of HALO [Vytiniotis et al. 2013] to avoid universally quantified formulas and the resulting unpredictability.

Notation 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.

Semantic Preservation We can prove that 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 Decidable Type Checking

We make the type checking from Figure 5 decidable by checking subtyping via an SMT solver.

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 embeddings of) the refinements of provably terminating binders [Vazou et al. 2014]:

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

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

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

Fig. 7. Syntax of Predicates, Terms and Reflected Functions

version $\preceq$-Base-λ^S that uses an SMT solver to check the validity of the subtyping VC.

$$\frac{\Gamma' \doteq \Gamma, v : \{v : B^{\Downarrow} \mid e\} \qquad \text{SmtValid}(\lfloor \Gamma' \rfloor \Rightarrow \lfloor e' \rfloor)}{\Gamma; R \vdash_S \{v : B \mid e\} \preceq \{v : B \mid e'\}} \; \preceq\text{-Base-}\lambda^S$$

We rely on the semantic preservation of the translation relation to prove soundness of subtyping.

Lemma 5.1. *If* $\Gamma; R \vdash_S \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}$ *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 using the algorithmic subtyping rule $\preceq$-Base-λ^S instead of the denotational rule $\preceq$-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* $\Gamma; R \vdash e : \tau$.

6 COMPLETE VERIFICATION

Next, we make the type checking from Figure 5 algorithmic by checking subtyping using our new PBE algorithm for validity checking under a given reflection environment. First, we describe the input-output behavior of PBE and show how it makes refinement checking *algorithmic* (§ 6.1).Next, we formalize the algorithm and show that it is sound (§ 6.2), it is complete with respect to equational proofs (§ 6.3), and, that it terminates (§ 6.4).

6.1 Algorithmic Type Checking

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]:

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

Validity Checking Given a reflection environment R, type environment Γ, and expression e, the procedure $\text{PBE}(\lfloor R \rfloor, \lfloor \Gamma \rfloor, \lfloor e \rfloor)$ 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 PBE to check the validity of the subtyping VC.

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

This typing rule is sound as functions reflected in R always respect the typing environment Γ (by construction), and because PBE is sound (Theorem 6.3).

Lemma 6.1. *If* $\Gamma; R \vdash_{PBE} \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}$ *then* $\Gamma; R \vdash \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}$.

Unfold	:	$(\Psi, \Phi) \to \Phi$
Unfold$(\Psi,\ \Phi)$	=	$\Phi \cup \bigcup_{f(\overline{t}) \prec \Phi} \text{Instantiate}\left(\Psi, \Phi, f, \overline{t}\right)$
Instantiate $\left(\Psi, \Phi, f, \overline{t}\right)$	=	$\left\{(\lfloor f(\overline{x}) \rfloor = b_i)\left[\overline{t}/\overline{x}\right] \;\middle|\; (p_i \Rightarrow b_i) \in \overline{d},\ \ \Phi \vdash_S p_i\left[\overline{t}/\overline{x}\right]\right\}$
where		
$\lambda\overline{x}.\langle\overline{d}\rangle$	=	$\Psi(f)$
PBE	:	$(\Psi, \Phi, p) \to \texttt{Bool}$
PBE(Ψ, Φ, p)	=	loop $\left(0,\ \Phi \cup \bigcup_{f(\overline{t}) \prec p} \text{Instantiate}\left(\Psi, \Phi, f, \overline{t}\right)\right)$
where		
loop (i, Φ_i)		
$\mid\ \Phi_i \vdash_S p$	=	*true*
$\mid\ \Phi_{i+1} \subseteq \Phi$	=	*false*
$\mid$ otherwise	=	loop $(i+1, \Phi_{i+1})$
where		
Φ_{i+1}	=	$\Phi \cup$ Unfold$(\Psi,\ \Phi_i)$

Fig. 8. Algorithm PBE: Proof by Symbolic Evaluation

6.2 Algorithm

Figure 7 describes the input environments for PBE. The logical environment Φ contains a set of hypotheses p, described in Figure 6. The definitional environment Ψ maps function symbols f to their definitions $\lambda\overline{x}.\langle\overline{p \Rightarrow b}\rangle$, written as λ-abstractions over guarded bodies. Moreover, the body b and the guard p contain neither λ nor `if`. These restrictions do not impact expressiveness: λs can be named and reflected, and `if`-expressions can be pulled out into top-level guards using `DeIf`$(\cdot)$, found in Appendix **??**. 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 $t' \in \Phi$. We abuse notation to write $f(\overline{t}) \prec t'$ for $f(\overline{t}) \prec \{t'\}$. We write $\Phi \vdash_S p$ 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:

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

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 Unfold$(\Psi,\ \Phi)$ shown in Figure 8 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 6.2. *For every* RJ ♣ what is $\models$? ♣ $\Gamma \models R$, *and* $\theta \in (\!|\Gamma|\!)$,

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

The Algorithm Figure 8 shows our proof search algorithm $\mathsf{PBE}(\Psi, \Phi, p)$ 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 PBE. 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 6.3 (**Soundness**). *If* $\mathsf{PBE}(\lfloor R \rfloor, \lfloor \Gamma \rfloor, \lfloor e \rfloor)$ *then* $\forall \theta \in (\!|\Gamma|\!),\ \theta \cdot R[e] \hookrightarrow^{\star} \mathtt{True}$

We prove Theorem 6.3 using the Lemma 6.2 that relates instantiation, SMT validity, and the exact semantics. Intuitively, PBE is sound as it reasons about a finite set of instances by *conservatively* treating all function applications as *uninterpreted* [Nelson 1981].

6.3 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 $\mathsf{PBE}(\Psi, \Phi, t = t')$ 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 § 6.4 we will show that the fixpoint is computable: there is an unfolding depth at which PBE 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 6.4 (**Monotonicity**). $\forall 0 \leq n.\ \mathsf{Unfold}^*(\Psi, \Phi, n) \subseteq \mathsf{Unfold}^*(\Psi, \Phi, n+1)$.

Lemma 6.5 (**Fixpoint**). *Let* $\Phi_i \doteq \mathsf{Unfold}^*(\Psi, \Phi, i)$. *If* $\Phi_n = \Phi_{n+1}$ *then* $\forall n < m.\ \Phi_m = \Phi_n$.

Uncovering Next we prove that every function application term that is *uncovered* by unfolding to depth n is congruent to a term in the n-depth unfolding.

Lemma 6.6 (**Uncovering**). *Let* $\Phi_n \equiv \mathsf{Unfold}^*(\Psi, \Phi \cup \{v = t\}, n)$. *If* $\Phi_n \vdash_S v = t'$ *then for every* $f(\overline{t'}) \prec t'$ *there exists* $f(\overline{t}) \prec \Phi_n$ *such that* $\Phi_n \vdash_S t_i = t'_i$.

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 6.7 (**Congruence**). *If* $\Phi, v = t \vdash_S v = t'$ *and* $v \notin \Phi, t, t'$ *then for every* $f(\overline{t'}) \prec t'$ *there exists* $f(\overline{t}) \prec \Phi, t$ *such that* $\Phi \vdash_S t_i = t'_i$.

Unfolding Preserves Equational Links Next, we use the uncovering Lemma 6.6 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 6.8 (**Link**). *If* $\mathsf{Unfold}^*(\Psi, \Phi \cup \{v = t\}, n) \vdash_S v = t'$ *then* $\mathsf{Unfold}^*(\Psi, \Phi \cup \{v = t\}, n+1) \vdash_S \mathsf{Unfold}(\Psi,\ \Phi \cup \{v = t'\})$.

Equational Proof There is an *equational proof* that t equals t' under Ψ, Φ ($\Psi, \Phi \vdash t \cong t'$) if

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

$$\frac{\Psi, \Phi \vdash t \twoheadrightarrow t'' \quad \Psi, \Phi \vdash t' \twoheadrightarrow t''}{\Psi, \Phi \vdash t \cong t'}\ \textsc{Eq-Sym}$$

RJ ♣ TODO:explain:mimics normalization etc. ♣ There is an *directional proof* that t equals t' under Ψ, Φ ($\Psi, \Phi \vdash t \twoheadrightarrow t'$) if

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 6.9. *If* $\Psi, \Phi \vdash t \twoheadrightarrow t'$ *then* $\exists 0 \leq n.\ \mathsf{Unfold}^*(\Psi, \Phi \cup \{v = t\}, n) \vdash_S v = t'$.

PBE ***is a Fixpoint of Bounded Unfolding*** Next, we show that the proof search procedure PBE 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 6.10 (**Fixpoint**). $\mathsf{PBE}(\Psi, \Phi, t = t')$ *iff* $\exists n.\ \mathsf{Unfold}^*(\Psi, \Phi \cup \{v = t\}, n) \vdash_S v = t'$

The proof follows by observing that $\mathsf{PBE}(\Psi, \Phi, t = t')$ 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 8, Φ_i is equal to $\mathsf{Unfold}^*(\Psi, \Phi \cup \{v = t\}, i)$, which then yields the result.

Completeness of PBE By combining Lemma 6.10 and Lemma 6.8 we can show that PBE is complete, *i.e.* if there is an equational proof that $t = t'$ under Ψ, Φ, then $\mathsf{PBE}(\Psi, \Phi, t = t')$ returns *true*.

Theorem 6.11 (**Completeness**). *If* $\Psi, \Phi \vdash t \cong t'$ *then* $\mathsf{PBE}(\Psi, \Phi, t = t') = \mathit{true}$.

6.4 PBE Terminates

So far, we have shown that our proof search procedure PBE 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, our proof search procedure always terminates.

Intuition RJ ♣ FIXME ♣ Put another way your "clean intuition" applies to single concrete inputs: f n will terminate on any INDIVIDUAL n. However in symbolic land we are working with potentially INFINITE sets of n – each of which would terminate individually – but we need to show that our COLLECTIVE procedure will also terminate. Note that the naive approach of enumerating each n and running it obviously does not terminate. That's why we have to carefully formalize the notion of "execution" at a logical level and use transparency to connect the two worlds. The key insight is that if PBE *does not terminate* then it is because (1) the set of depth n unfoldings form a strictly increasing infinite chain, $\mathsf{Unfold}^*(\Psi, \Phi, 0) \subset \mathsf{Unfold}^*(\Psi, \Phi, 1) \ldots$, which means that (2) there is an infinite sequence of application terms $f_0(\overline{t_0}) \hookrightarrow^* f_1(\overline{t_1}) \hookrightarrow^* \ldots$ which means that (3) the reflected function f_0 is *not terminating*. Thus, when all reflected functions terminate, PBE must terminate! Next, we formalize the above intuition and prove the termination Theorem 6.18.

Transparency An environment Γ is *inconsistent* if $\lfloor\Gamma\rfloor \vdash_S \mathit{false}$. 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`.

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 6.12 (**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 `DeIf(·)` hoists `if`-expressions out of the body and into the top-level guards, the below fact follows from the properties of CBV:

Lemma 6.13. *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* $\theta \cdot R[e_i] \hookrightarrow^\star C[f(\theta \cdot R[\overline{e}])]$.

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

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

Steps and Reductions Next, using Lemmas 6.13, 6.2, and the definition of symbolic steps, we show that every symbolic step corresponds to a *sequence* of steps in the concrete semantics:

Lemma 6.14 (**Step-Reductions**). *If* $f(\overline{\lfloor e\rfloor}) \rightsquigarrow f'(\overline{\lfloor e'\rfloor})$ *is a symbolic 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.

Steps and Values Next, we show that if $f(\overline{y}) \rightsquigarrow t'$ is a symbolic step under an Γ that is inhabited by θ then $f\ (\overline{y})$ reduces to a value under θ. The proof follows by observing that if Γ is inhabited by θ, and a particular step is possible, then the guard corresponding to that step must also be true under θ and hence, by totality, the function must reduce to a value under the given store.

Lemma 6.15 (**Step-Value**). *If* $\theta \in (\!|\Gamma|\!)$ *and* $f(\overline{y}) \rightsquigarrow t'$ *is a* $\lfloor R\rfloor, \lfloor\Gamma\rfloor$ *step then* $R[\theta \cdot f(\overline{y})] \hookrightarrow^\star v$.

Symbolic Trace A sequence $f_0(\overline{t_0}), f_1(\overline{t_1}), f_2(\overline{t_2}), \ldots$ is a Ψ, Φ-*symbolic 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* symbolic traces.

Theorem 6.16 (**Finite-Trace**). *If* Γ *is inhabited then every* $(\lfloor R\rfloor, \lfloor\Gamma\rfloor)$*-trace is finite.*

We prove the above lemma by contradiction. Specifically, we show by Lemma 6.14 that an infinite $(\lfloor R \rfloor, \lfloor \Gamma \rfloor)$-trace combined with the assumption that $(\!|\Gamma|\!)$ is not empty yields *at least one* infinite *concrete trace*, which contradicts totality, and hence, all the $(\lfloor R \rfloor, \lfloor \Gamma \rfloor)$ symbolic traces must be finite.

Unfolding Chains and Traces If the unfolding Ψ, Φ yields an infinite chain, then Ψ, Φ has an infinite symbolic trace:

Lemma 6.17 (**Ascending Chains**). *Let* $\Phi_i \doteq \mathsf{Unfold}^*(\Psi, \Phi, i)$. *If there exists an (infinite) ascending chain* $\Phi_0 \subset \ldots \subset \Phi_n \ldots$ *then there exists an (infinite) symbolic trace* $f_0(\overline{t_0}), \ldots, f_n(\overline{t_n}), \ldots$.

To prove the above, we construct a 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}).

PBE ***Terminates*** Finally, we prove that the proof search procedure PBE terminates.

Theorem 6.18 (**Termination**). *If* R *is total under* Γ *and* Γ *is inhabited, then* $\mathsf{PBE}(\lfloor R \rfloor, \lfloor \Gamma \rfloor, p)$ *terminates.*

If PBE loops forever, there must be an infinite strictly ascending chain of unfoldings Φ_i, and hence, by lemma 6.17 and infinite symbolic trace which contradicts lemma 6.16. Thus, PBE must terminate. In practice, to check if Γ is inhabited we need only find suitable concrete values via random [Claessen and Hughes 2000] or SMT-guided generation [Seidel et al. 2015].

7 EVALUATION

We have implemented refinement reflection in Liquid Haskell. In this section, we evaluate our approach by using Liquid Haskell to verify a variety of deep specifications of Haskell functions drawn from the literature and categorized in Figure 9, totalling about 4,000 lines of specifications and proofs. Verification is fast: from 1 to 7s per file (module), which is about twice as much time as GHC takes to compile the corresponding module. Overall there is about one line of type specification (theorems) for three lines of code (implementations and proof). Next, we detail each of the five classes of specifications, illustrate how they were verified using refinement reflection, and discuss the strengths and weaknesses of our approach. *All* of these proofs require refinement reflection, *i.e.* are beyond the scope of shallow refinement typing.

Proof Strategies. Our proofs use three building blocks, that are seamlessly connected via refinement typing:

- ***Un/folding*** definitions of a function `f` at arguments `e1...en`, which due to refinement reflection, happens whenever the term `f e1 ... en` appears in a proof. For exposition, we render the function whose un/folding is relevant as **f**;
- ***Lemma Application*** which is carried out by using the "because" combinator (`∵`) to instantiate some fact at some inputs;
- ***SMT Reasoning*** in particular, *arithmetic*, *ordering* and *congruence closure* which kicks in automatically (and predictably!), allowing us to simplify proofs by not having to specify, *e.g.* which subterms to rewrite.

7.1 Arithmetic Properties

The first category of theorems pertains to the textbook Fibonacci and Ackermann functions. In § 2 we proved that `fib` is increasing. We prove a higher order theorem that lifts increasing functions to monotonic ones:

```
fMono :: f:(Nat → Int) → fUp:(z:Nat → {f z ≤ f (z+1)}) → x:Nat → y:{x < y} → {f
  x ≤ f y}
```

CATEGORY			LOC
I.	**Arithmetic**		
	Fibonacci	§ 2	48
	Ackermann	[Tourlakis 2008]	280
II.	**Algebraic Data Types**		
	Fold Universal	[Mu et al. 2009]	105
III.	**Typeclasses**	Fig 10	
	Monoid	Peano, Maybe, List	189
	Functor	Maybe, List, Id, Reader	296
	Applicative	Maybe, List, Id, Reader	578
	Monad	Maybe, List, Id, Reader	435
IV.	**Functional Correctness**		
	SAT Solver	[Casinghino et al. 2014]	133
	Unification	[Sjöberg and Weirich 2015]	200
V.	**Deterministic Parallelism**		
	Conc. Sets	§ ??	906
	n-body	§ ??	930
	Par. Reducers	§ ??	55
TOTAL			4155

Fig. 9. **Summary of Case Studies**

Monoid	
Left Id.	`mempty` $x \diamond \equiv x$
Right Id.	$x \diamond$ `mempty` $\equiv x$
Assoc	$(x \diamond y) \diamond z \equiv x \diamond (y \diamond z)$
Functor	
Id.	`fmap id` $xs \equiv$ `id` xs
Distr.	`fmap` $(g \circ h)\ xs \equiv$ (`fmap` $g \circ$ `fmap` h) xs
Applicative	
Id.	`pure id` $\circledast\ v \equiv v$
Comp.	`pure` $(\circ) \circledast u \circledast v \circledast w \equiv u \circledast (v \circledast w)$
Hom.	`pure` $f \circledast$ `pure` $x \equiv$ `pure` $(f\ x)$
Inter.	$u \circledast$ `pure` $y \equiv$ `pure` $(\$\ y) \circledast u$
Monad	
Left Id.	`return` $a \gg\!= f \equiv f\ a$
Right Id.	$m \gg\!=$ `return` $\equiv m$
Assoc	$(m \gg\!= f) \gg\!= g \equiv m \gg\!= (\lambda x \to f\ x \gg\!= g)$

Fig. 10. **Summary of Verified Typeclass Laws**

By instantiating the function argument `f` of `fMono` with `fib`, we proved monotonicity of `fib`.

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

Using higher order functions like `fMono`, we mechanized the proofs of the Ackermann function properties from [Tourlakis 2008]. We proved 12 equational and arithmetic properties using various proof techniques, including instantiation of higher order theorems (like `fMono`), proof by construction, and natural number and generalized induction.

7.2 Algebraic Data Properties

Fold Universality as encoded in Adga [Mu et al. 2009]. The following encodes the universal property of `foldr`

```
foldr_univ :: f:(a → b → b)
           → h:(L a → b)
           → e:b
           → ys:L a
           → base:{h [] == e }
           → step:(x:a → xs:[a] → {h (x:xs) == f x (h xs)})
           → {h ys == foldr f e ys}
```

The Liquid Haskell proof term of `foldr_univ` is similar to the one in Agda. But, there are two major differences. First, specifications do not support quantifiers: universal quantification of `x` and `xs` in the step assumption is encoded as functional arguments. Second, unlike Agda, Liquid Haskell does not support optional arguments, thus to use `foldr_univ` the proof arguments `base` and `step` that were implicit arguments in Agda need to be explicitly provided. For example, the following code calls `foldr_universal` to prove `foldr_fusion` by explicitly applying proofs for the base and the step arguments

```
foldr_fusion :: h:(b → c)
             → f:(a → b → b)
             → g:(a → c → c)
             → e:b
             → z:[a]
             → x:a → y:b
             → fuse: {h (f x y) == g x (h y)})
             → {(h.foldr f e) z == foldr g (h e) z}

foldr_fusion h f g e ys fuse = foldr_univ g (h . foldr f e) (h e) ys
                                 (fuse_base h f e)
                                 (fuse_step h f e g fuse)
```

Where, for example, `fuse_base` is a function with type

```
fuse_base :: h:(b → c) → f:(a → b → b) → e:b → {(h . foldr f e)
    [] == h e}
```

7.3 Typeclass Laws

We used Liquid Haskell to prove the Monoid, Functor, Applicative, and Monad Laws, summarized in Figure 10, for various user-defined instances summarized in Figure 9.

Monoid Laws A Monoid is a datatype equipped with an associative binary operator ◊ (`mappend`) and an *identity* element `mempty`. We prove that `Peano` (with a suitable `add` and `Z`), `Maybe` (with a suitable `mappend` and `Nothing`), and `List` (with append `++` and `[]`) satisfy the monoid laws.

Functor Laws A type is a functor if it has a function `fmap` that satisfies the *identity* and *distribution* (or fusion) laws in Figure 10. For example, consider the proof of the `map` distribution law for the lists, also known as "map-fusion", which is the basis for important optimizations in GHC [Wiki 2008]. We reflect the definition of `map` for lists and use it to specify fusion and verify it by an inductive proof:

```
map_fusion :: f:(b → c) → g:(a → b) → xs:[a] → {map (f . g) xs = (map f . map g)
  xs}
```

Monad Laws The monad laws, which relate the properties of the two operators ≫= and `return` (Figure 10), refer to λ-functions which are encoded in our logic as uninterpreted functions. For example, we can encode the associativity property as a refinement type alias:

```
type AssocLaw m f g = { m >>= f >>= g = m >>= (\x → f x >>= g) }
```

and use it to prove that the list-bind is associative:

```
assoc :: m:[a] → f:(a →[b]) → g:(b →[c]) → AssocLaw m f g
assoc [] f g     =  [] >>= f >>= g
                 =. [] >>= g
```

```
                =. []
                =. [] >>= (\x → f x >>= g) ** QED
assoc (x:xs) f g =  (x:xs) >>= f  >>= g
                =. (f x ++ xs >>= f) >>= g
                =. (f x >>= g) ++ (xs >>= f >>= g) ∵ bind_append (f x) (xs >>= f) g
                =. (f x >>= g) ++ (xs >>= \y → f y >>= g) ∵ assoc xs f g
                =. (\y → f y >>= g) x ++ (xs >>= \y → f y >>= g) ∵ βeq f g x
                =. (x:xs) >>= (\y → f y >>= g) ** QED
```

Where the `bind_append` fusion lemma states that:

```
bind_append
  :: xs:[a] → ys:[a] → f:(a → [b]) → {(xs++ys) >>= f = (xs >>= f)++(
     ys >>= f)}
```

Notice that the last step requires β-equivalence on anonymous functions, which we get by explicitly inserting the redex in the logic, via the following lemma with `trivial` proof

```
βeq :: f:_ → g:_ → x:_ → {f x >>= g = (\y → f y >>= g) x}
βeq _ _ _ = trivial
```

7.4 Functional Correctness

Next, we proved correctness of two programs from the literature: a unification algorithm and a SAT solver.

Unification We verified the unification of first order terms, as presented in [Sjöberg and Weirich 2015]. First, we define a predicate alias for when two terms `s` and `t` are equal under a substitution `su`:

```
eq_sub su s t = apply su s == apply su t
```

Now, we defined a Haskell function `unify s t` that can diverge, or return `Nothing`, or return a substitution `su` that makes the terms equal:

```
unify :: s:Term → t:Term → Maybe {su | eq_sub su s t}
```

For specification and verification, we only needed to reflect `apply` and not `unify`; thus, we only had to verify that the former terminates, and not the latter. We proved correctness by invoking separate helper lemmata. For example, to prove the post-condition when unifying a variable `TVar i` with a term `t` in which `i` *does not* appear, we apply a lemma `not_in`:

```
unify (TVar i) t2 | not (i ∈ freeVars t2) = Just (const [(i, t2)] ∵ not_in i t2)
```

i.e. if `i` is not free in `t`, the singleton substitution yields `t`:

```
not_in :: i:Int → t:{Term | not (i ∈ freeVars t)} → {eq_sub [(i,t)] (
    TVar i) t}
```

This example highlights the benefits of partial verification on a legacy programming language: potential diverging code (*e.g.* the function `unify`) coexists and invokes proof terms.

SAT Solver As another example, we implemented and verified the simple SAT solver used to illustrate and evaluate the features of the dependently typed language Zombie [Casinghino et al. 2014]. The solver takes as input a formula `f` and returns an assignment that *satisfies* `f` if one exists, as specified below.

```
solve :: f:Formula → Maybe {a:Assignment | sat a f}
```

The function `sat a f` returns `True` iff the assignment `a` satisfies the formula `f`. Verifying `solve` follows directly by reflecting `sat` into the refinement logic.

8 RELATED WORK

SMT-Based Verification SMT-solvers have been extensively used to automate program verification via Floyd-Hoare logics [Nelson 1981]. Our work is inspired by Dafny's Verified Calculations [Leino and Polikarpova 2016], 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. Dafny, and the related F* [Swamy et al. 2016] which like Liquid Haskell, uses types to compose proofs, offer more automation by translating recursive functions to SMT axioms. However, unlike reflection, this axiomatic approach renders typechecking and verification undecidable (in theory) and leads to unpredictability and divergence (in practice) [Leino and Pit-Claudel 2016].

RJ ♣ RELATED ♣ Of course it may be that sometimes in practice, discretion is the better part of valor – *i.e.* it may be more expedient to quickly time out after a fixed, small number of instantiations rather than to perform an exhaustive search. RJ ♣ PBE brings completeness for the first time yada ♣ RJ ♣ More stuff from "butterfly effect" text? (see scratch.tex) ♣

Dependent types Our work is also inspired by dependently typed systems like Coq [Bertot and Castéran 2004] and Agda [Norell 2007]. Reflection shows how deep specification and verification in the style of Coq and Agda can be *retrofitted* into existing languages via refinement typing. Furthermore, we can use SMT to significantly automate reasoning over important theories like arithmetic, equality and functions. It would be interesting to investigate how the tactics and sophisticated proof search of Coq *etc.* can be adapted to the refinement setting.

Dependent Types in Haskell Integration of dependent types into Haskell has been a long standing goal that dates back to Cayenne [Augustsson 1998], a Haskell-like, fully dependent type language with undecidable type checking. In a recent line of work Eisenberg and Stolarek [2014] aim to allow fully dependent programming within Haskell, by making "type-level programming ... at least as expressive as term-level programming". Our approach differs in two significant ways. First, reflection allows SMT-aided verification, which drastically simplifies proofs over key theories like linear arithmetic and equality. Second, refinements are completely erased at run-time. That is, while both systems automatically lift Haskell code to either uninterpreted logical functions or type families, with refinements, the logical functions are not accessible at run-time and promotion cannot affect the semantics of the program. As an advantage (resp. disadvantage), refinements cannot degrade (resp. optimize) the performance of programs.

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. In contrast, our proofs are simply Haskell programs, we can use SMT solvers to automate reasoning, and, most importantly, we can connect the validity of proofs with the semantics of the programs.

Deterministic Parallelism Deterministic parallelism has plenty of theory but relatively few practical implementations. Early discoveries were based on limited producer-consumer communication, such as single-assignment variables [Arvind et al. 1989; Tesler and Enea 1968], Kahn process networks [Kahn 1974], and synchronous dataflow [Lee and Messerschmitt 1987]. Other models use synchronous updates to shared state, as in Esterel [Benveniste et al. 2003] or PRAM. Finally, work on type systems for permissions management [Naden et al. 2012; Westbrook et al. 2012], supports the development of *non-interfering* parallel programs that access disjoint subsets of the heap in parallel. Parallel functional programming is also non-interfering [Fluet et al. 2007; Marlow et al. 2009]. Irrespective of which theory is used to support deterministic parallel programming, practical implementations such as Cilk [Blumofe et al. 1995] or Intel CnC [Budimlic et al. 2010] are limited by host languages with type systems insufficient to limit side effects, much less prove associativity. Conversely, dependently typed languages like Agda and Idris do not have parallel programming APIs and runtime systems.

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.

Arvind, R. S. Nikhil, and K. K. Pingali. 1989. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.* (1989).

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*.

A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* (2003).

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

R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. 1995. Cilk: an efficient multithreaded runtime system. *SIGPLAN Notices* (1995).

Z. Budimlic, M. Burke, V. Cave, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar. 2010. The CnC Programming Model. *Journal of Scientific Programming* (2010).

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

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)*.

M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. 2007. Manticore: a heterogeneous parallel language. In *DAMP*.

G. Gentzen. 1935. ... (1935).

W. A. Howard. 1980. The formulae-as-types notion of construction. (1980). http://lecomte.al.free.fr/ressources/PARIS8_LSL/Howard80.pdf
G. Kahn. 1974. The semantics of a simple language for parallel programming. In *Information Processing*.
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).
E. A. Lee and D. G. Messerschmitt. 1987. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. Comput.* (1987).
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, S. Peyton-Jones, and S. Singh. 2009. Runtime support for multicore Haskell. In *ICFP*.
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).
K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. 2012. A Type System for Borrowing Permissions. In *POPL*.
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).
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*.
L. G. Tesler and H. J. Enea. 1968. A language design for concurrent processes. In *AFIPS*.
G. Tourlakis. 2008. AckermannâĂŹs Function. (2008). http://www.cs.yorku.ca/~gt/papers/Ackermann-function.pdf.
N. Vazou, A. Bakst, and R. Jhala. 2015. Bounded refinement types. In *ICFP*.
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
Makarius Wenzel. 2016. The Isabelle System Manual. (2016). https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2016-1/doc/system.pdf
E. Westbrook, J. Zhao, Z. Budimlić, and V. Sarkar. 2012. *Practical Permissions for Race-Free Parallelism.*
GHC Wiki. 2008. GHC Optimisations. (2008). https://wiki.haskell.org/GHC_optimisations.
H. Xi and F. Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types.. In *PLDI*.