Refinement Reflection

(or, how to convert a legacy programming language into a theorem prover)

ANONYMOUS AUTHOR(S)

We introduce *refinement reflection*, a method to extend *legacy* languages—with highly tuned libraries, compilers, and run-times—into theorem provers. The key idea is to reflect the code implementing a user-defined function into the function's (output) refinement type. As a consequence, at *uses* of the function, the function definition is unfolded into the refinement logic in a precise and predictable manner. We have implemented our approach in Liquid Haskell thereby retrofitting theorem proving into Haskell. We show how to use reflection to verify that many widely used instances of the Monoid, Applicative, Functor, and Monad typeclasses actually satisfy key algebraic laws required to make the code using the typeclasses safe. Finally, transforming a mature language—with highly tuned parallel runtime—into a theorem prover enables us to build the first *deterministic parallelism library* that verifies assumptions about associativity and ordering—that are crucial for determinism but simply assumed by existing systems.

ACM Reference format:
Anonymous Author(s). 2017. Refinement Reflection. *Proc. ACM Program. Lang.* 1, 1, Article 1 (June 2017), 29 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 OVERVIEW

We begin with an overview of how refinement reflection allows us to write proofs *of* and *by* Haskell functions.

1.1 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 decorated with logical predicates drawn from an SMT decidable logic [??]. For example, we define the `Nat` type as `Int` refined by the predicate `0 ≤ v` which belongs to the quantifier free logic of linear arithmetic and uninterpreted functions (QF-UFLIA [?]).

```
type Nat = { v:Int | 0 ≤ v }
```

Here, `v` names the value described by the type so the above denotes "set of `Int` values `v` that are not less than 0".

Specification & Verification We can use refinements to define and type the textbook Fibonacci function as:

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

Here, the input type's refinement specifies a *pre-condition* that the parameters must be `Nat`, which is needed to ensure termination, and the output types's refinement specifies a *post-condition* that the result is also a `Nat`. Refinement type checking lets us specify and (automatically) verify the

2017. 2475-1421/2017/6-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

shallow property that if `fib` is invoked with a non-negative `Int`, then it terminates and yields a non-negative `Int`.

Propositions We can use refinements to define a data type representing propositions simply as an alias for unit, a data type that carries no useful runtime information:

```
type Prop = ()
```

which can be *refined* with propositions about the code. For example, the following type states that 2 + 2 equals 4

```
type Plus_2_2_eq_4 = { v: Prop | 2 + 2 = 4 }
```

For clarity, we abbreviate the above type by omitting the irrelevant basic type `Prop` and variable `v`:

```
type Plus_2_2_eq_4 = { 2 + 2 = 4 }
```

Function types encode universally quantified propositions:

```
type Plus_com = x:Int → y:Int → { x + y = y + x }
```

The parameters `x` and `y` refer to input values: any inhabitant of the above is a proof that `Int` addition commutes.

Proofs We *prove* the above theorems by writing Haskell programs. To ease this task, Liquid Haskell provides primitives to construct proof terms by "casting" expressions to `Prop`.

```
data QED = QED

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

To resemble mathematical proofs, we make this casting post-fix. Thus, we write `e ** QED` to cast `e` to a value of `Prop`. For example, we can prove the above propositions by defining `trivial = ()` and then writing:

```
pf_plus_2_2 :: Plus_2_2_eq_4    pf_plus_comm :: Plus_comm
pf_plus_2_2 = trivial           pf_plus_comm = \x y → trivial
```

Via standard refinement type checking, the above code yields the respective verification conditions (VCs),

$$2 + 2 = 4 \qquad\qquad \forall x, y . x + y = y + x$$

which are easily proved 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 ?, Liquid Haskell ensures that all terms with non-trivial refinements provably terminate and evaluate to (non-bottom) values, which makes our proofs sound.

1.2 Refinement Reflection

Suppose we wish to prove properties about the `fib` function, *e.g.* that `fib 2` equals `1`.

```
type fib2_eq_1 = { fib 2 = 1 }
```

Standard refinement type checking runs into two problems. First, for decidability and soundness, arbitrary user-defined functions do not belong to 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 shallow type `Nat → Nat` which is too weak to verify `fib2_eq_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 tells Liquid Haskell to declare 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* $\forall n, m.\ n = m \Rightarrow$ `fib` $n =$ `fib` m. On its own, the uninterpreted function is not terribly useful, as it does not let us prove `fib2_eq_1` which requires reasoning about the *definition* of `fib`.

Step 2: Reflection In the next key step, Liquid Haskell reflects the definition into the refinement type of `fib` by automatically strengthening the user defined type for `fib` to:

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

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

```
fibP v n = v = if n = 0 then 0 else (if n = 1 then 1 else fib (n-1) + fib (n-2))
```

Step 3: Application With the reflected refinement type, each application of `fib` in the code automatically unfolds the `fib` definition *once* in the logic. We prove `fib2_eq_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 proof yields the following VC that is discharged by the SMT solver, even though `fib` is uninterpreted:

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

Note that the verification of `pf_fib2` relies merely on the fact that `fib` was 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 ] ** QED
```

Thus, unlike classical dependent typing, refinement reflection *does not* perform any type-level computation.

Reflection vs. Axiomatization An alternative *axiomatic* approach, used by Dafny [?] and F* [?], is to encode `fib` using a universally quantified SMT formula (or axiom): $\forall n.$ `fibP (fib` n`)` n. Axiomatization offers greater automation than reflection. Unlike Liquid Haskell, F* will verify the following by *automatically instantiating* the above axiom at 2, 1 and `0`:

```
val axPf_fib2: unit → Tot (v:unit {fib 2 = 1})
let axPf_fib2 _ = ()
```

The automation offered by axioms is a bit of a devil's bargain, as axioms render checking of the VCs *undecidable*. 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" and provide a "fuel" parameter [?] that can be used to restrict the numbers of the SMT unfoldings, which ensure termination, but can cause the axiom to not be instantiated at the right places. As an example, F*'s default fuel is not adequate to compute the value of `fib` at 15, rendering the following property unpredictably unsafe.

```
val axPf_fib2: unit → Tot (v:unit {fib 15 = 610})
let axPf_fib2 _ = ()
```

In short, per the authors of Dafny, the undecidability of the VC checking and its attendant heuristics makes verification unpredictable [?].

1.3 Proof By Evaluation

To simplify trivial proof by rewriting (like `fib 2 == 1`) while preserving predictable behavior we introduce PBE (Proof By Evaluation). PBE symbolically evaluates the subexpressions of the proof obligations based on the definitions of the reflected functions. For instance the proof obligation `fib 2 == 1` fires the unfolding of `fib` 2 which in turn fires the unfoldings of both `fib` 1 and `fib` 0. All the three unfoldings are collected in an assumption environment under which the proof obligation is checked strengthening the initial obligation to the following SMT query

$$(\texttt{fib}\ 2 = \texttt{fib}\ 1 + \texttt{fib}\ 0 \land \texttt{fib}\ 1 = 1 \land \texttt{fib}\ 0 = 0) \Rightarrow \texttt{fib}\ 2 = 1$$

Thus the SMT can prove `fib 2 == 1` while treating `fib` as an uninterpreted function.

Inductive Proofs PBE only unfolds reflected functions when it can be statically decided exactly which branch the function unfolds to. For example PBE cannot automatically prove inductive properties like that `fib` is (locally) increasing: for all n, $\texttt{fib}\ n \leq \texttt{fib}\ (n+1)$. PBE will not automatically unfold $\texttt{fib}\ n$ since it is not determined which branch `fib n` will take, for all `n`. The programmer can use standard analysis on the value of `n` to appropriately fire PBE unfoldings and thus construct inductive proofs.

```
fibUp :: n:Nat → { fib n ≤ fib (n+1) }
fibUp n | n == 0
        -- UNFOLDINGS
        -- fib 0 == 1, fib 1 == 1
        =  trivial
        | n == 1
        -- UNFOLDINGS
        -- fib 2 == fib 1 + fib 0, fib 1 == 1, fib 0 == 0
        = trivial
        | otherwise
        -- UNFOLDINGS
        -- fib (n+1) == fib n + fib (n-1)
        -- fib n == fib (n-1) + fib (n-2)
        =  fibUp (n-1) &&& fibUp (n-2)
```

Case Splitting and Induction The proof `fibUp` works by induction on `n`. In the *base* cases `0` and `1`, we simply assert the relevant unfoldings. These are verified as PBE automatically unfolds the reflected `fib` definition 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 automatically 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.

The connective operator &&& is operationally the constant function that moreover combines in the logic the information form both its input arguments.

```
x &&& _ = x
```

Properties of PBE In § 4 we formalize the automatically generated unfoldings via PBE and discuss three main properties of our formalization

- **Completeness** PBE will actually unfold all the reflected functions that can be unfolded to exactly one branch.

- **Termination** Assuming that all the unfolded functions terminated, PBE will generate finitely many unfoldings.

As a result of the above two properties PBE is more predictable than fuel-based axiom instantiation approach that is traditionally used by SMT-verifiers, since PBE is automatically adjusting the number of instantiations (*i.e.* fuel) based on the context. Automatic fuel adjustment though may introduce a different source of unpredictability: verification slow-downs. For instance verification of `fib 16 == 987` is unpredictably unsafe in the fuel-based F* and unpredictably slow in our PBE-based approach. Yet, compared with axiomatization our notion of unpredictability is **local**: PBE is separately strengthening proof obligations thus avoiding the "butterfly effect" [?] where axioms introduced an one point of the code may affect verification in different places.

Reflection and Refinement Types NV:♣ This seems out of place now. ♣ The above examples illustrate how refinement types critically allow us to restrict the domain of both reflected functions and proof terms in order to enable sound theorem proving.

- ***Totality*** First, by refining `Int` to `Nat` we restrict the domain of Haskell's partial function `fib` which lets us use `fib` as a total function in the logic. Specifically, refinement type checking ensures soundness by verifying that `fib` is never called on negative numbers.
- ***Induction*** Second, the refinement type `Nat` in the argument of `fibUp` constrains `fibUp`'s domain, turning `fibUp` into a total proof that encodes induction on Natural numbers, while reasoning about Haskell's runtime integers.

Thus, refinement types let us encode proofs that only hold on refined types, like `Nat` while using Haskell's Integers, instead of forcing to switch to a user or library defined `Data.Natural` Haskell type. As a less elegant alternative, usually followed in dependently typed languages, `fib` could be defined as a function returning a `Maybe` – *i.e.* returning `Nothing` on negative inputs. Then the code and proofs would be littered with cases analyses required to extract `fib`'s result. Another alternative is Dafny's approach that uses code level assumptions and assertions to restrict function's domain and range. While these assumptions and assertions can be semi-automatically discharged using the SMT solver, they impose programmer overheads as they cannot be inferred unlike refinement types that are inferred using the abstract interpretation framework of liquid typing [?].

1.4 Higher Order Reasoning

Refinement reflection allows terminating functions to appear in the refinements but to preserve decidability refinements cannot contain quantifiers. Next we present how our refinement types can be used to encode quantified properties using lambda abstraction and dependent pairs for forall and exists respectively, thus encoding arbitrary higher order properties.

Encoding foralls Refinements smoothly accomodate universal quantification as lambda abstraction. For example, lets prove that every locally increasing function is monotonic, *i.e.* if `f z ≤ f (z+1)` for all `z`, then `f x ≤ f y` for all `x < y`.

```
fMono :: f:(Nat → Int)
      → fUp:(z:Nat → {f z ≤ f (z+1)})
      → x:Nat
      → y:{x < y}
      → {f x ≤ f y} / [y]
fMono f inc x y
  | x + 1 == y = fUp x
  | x + 1 < y  = fMono f fUp x (y-1) &&& fUp (y-1)
```

We prove the theorem by induction on y, which is specified by the annotation / [y] which states that y is a well-founded termination metric that decreases at each recursive call [?]. If x+1 == y, then we use fUp x. 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 use the general fMono theorem to prove that fib increases monotonically:

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

Encoding existentials Refinements encode existential quantification as dependent pairs. For example, we can prove that is there exists an m greater than n then there exists an upper bound to fib n.

```
fibBound :: n:Nat → (m::Int, {n < m}) → (m'::Int, {fib n ≤ m' }) @-}
fibBound n (m, pf) = (fib m, fibMono n m)
```

We use the notation (m::t1, t2) to encode dependent pairs: we bind the first element of the pair to the binder m, then m can appear in the refinements of the type t2 of the second element. To prove fibBound we open the existential assumption to get a value m that is greater then n. Then, fib m bounds fib n as proved by fibMono n m.

Even though quantifiers cannot appear inside the refinements lambdas and dependent pairs allows us to quantify values over the refinement properties. One limitation of this encoding is that quantifiers cannot exist inside logical connectives (like ∧ and ∨). In § 5 we encode logical connectives using data types. For instance conjunction is encoded as a product and disjunction as a union data type, thus naturally getting the expressiveness power of arbitrary higher order logics like Isabelle/HOL.

1.5 Case Study: Peano Numerals

Refinement reflection is not limited to programs operating on integers. We conclude the overview with a small library for Peano numerals, defined via the following algebraic data type:

```
data Peano = Z | S Peano
```

We can add two Peano numbers via:

```
reflect add :: Peano → Peano → Peano
add Z     m = m
add (S n) m = S (add n m)
```

In § **??** we will describe exactly how the reflection mechanism (illustrated via fibP) is extended to account for ADTs like Peano. Note that Liquid Haskell automatically checks that add is total [?], which lets us safely **reflect** it into the refinement logic.

Add Zero to Left As an easy warm up, lets show that adding zero to the left leaves the number unchanged:

```
zeroL :: n:Peano → { add Z n == n }
zeroL n = trivial
```

Add Zero to Right It is slightly more work to prove that adding zero to the right also leaves the number unchanged.

```
zeroR :: n:Peano → { add n Z == n }
zeroR Z     =  trivial
```

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

Fig. 1. **Syntax of λ^S**

```
  zeroR (S n) =  zeroR n
```

The proof goes by induction, splitting cases on whether the number is zero or non-zero. Consequently, we pattern match on the parameter n, and furnish separate proofs for each case. In the "zero" case, we simply unfold the definition of add. In the "successor" case, after unfolding we (literally) apply the induction hypothesis by using the because operator. Liquid Haskell's termination and totality checker verifies that we are in fact doing induction properly, *i.e.* the recursion in `zeroR` is well-founded (§ 2).

Commutativity Lets conclude by proving that add is commutative:

```
add_com :: a:_ → b:_ → {add a b = add b a}

add_com    Z  b =  zeroR b
add_com (S a) b =  add_com a b &&& sucR b a
```

using a lemma sucR

```
  sucR :: n:Peano → m:Peano →
                  {add n (S m) = S (add n m)}
  sucR = exercise_for_reader
```

Thus, refinement reflection lets us prove properties of Haskell programs just by writing Haskell programs: lemmas are just functions, case-splitting is just branching and pattern matching, and induction is just recursion. Next, we formalize refinement reflection and describe how to keep type checking decidable and predictable.

2 REFINEMENT REFLECTION: λ^R

NV:♣ Add exact reflected definition of fib ♣ We formalize refinement reflection in two 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 § 3 we define a language λ^S that soundly approximates λ^R while enabling decidable SMT-based type checking.

2.1 Syntax

Figure 3 summarizes the syntax of λ^R, which is essentially the calculus λ^U [?] 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.)

2.2 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*.

2.3 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 ? 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* $x \to \tau_x \tau$ where the input x has the type τ_x and the output τ may refer to the input binder x. We write B to abbreviate $\{v : B \mid \texttt{True}\}$, and $\tau_x \to \tau$ to abbreviate $x \to \tau_x \tau$ if x does not appear in τ.

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

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

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

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

2.4 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 ($\mathtt{reflect}\ f : \tau = e\ \mathtt{in}\ p$) as a `let rec`-binder ($\mathtt{let\ rec}\ f : \mathsf{Reflect}(\tau, e) = e\ \mathtt{in}\ p$) during type checking (rule T-Refl in Figure 2).

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

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

As an example, recall from § 1 that the **`reflect`** `fib` strengthens the type of `fib` with the refinement `fibP`.

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 $\mathtt{reflect}\ f$. Then, the rule T-Refl in Figure 2 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.

Consequences for Verification Reflection has a major consequence for verification: instead of being tethered to quantifier instantiation heuristics or having to program "triggers" as in Dafny [?] or F* [?], the programmer can predictably "unfold" the definition of the function during a proof simply by "calling" the function, which, as discussed in § 7, we have found to be a very natural way of structuring proofs.

2.5 Typing Rules

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

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

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{T-Var} \qquad \frac{}{\Gamma \vdash c : \mathsf{prim}(c)}\ \textsc{T-Con}$$

$$\frac{\Gamma \vdash p : \tau' \qquad \Gamma \vdash \tau' \preceq \tau}{\Gamma \vdash p : \tau}\ \textsc{T-Sub}$$

$$\frac{\Gamma \vdash e : \{v : B \mid \{_{|} r\}\}}{\Gamma \vdash e : \{v : B \mid \{_{|} r\} \wedge v = e\}}\ \textsc{T-Exact}$$

$$\frac{\Gamma, x : \tau_x \vdash e : \tau}{\Gamma \vdash \lambda x.e : x \to \tau_x \tau}\ \textsc{T-Fun}$$

$$\frac{\Gamma \vdash e_1 : (x \to \tau_x \tau) \qquad \Gamma \vdash e_2 : \tau_x}{\Gamma \vdash e_1\ e_2 : \tau}\ \textsc{T-App}$$

$$\frac{\begin{array}{cc}\Gamma, x : \tau_x \vdash b_x : \tau_x & \Gamma, x : \tau_x \vdash \tau_x \\ \Gamma, x : \tau_x \vdash b : \tau & \Gamma \vdash \tau\end{array}}{\Gamma \vdash \mathtt{let\ rec}\ x : \tau_x = b_x\ \mathtt{in}\ b : \tau}\ \textsc{T-Let}$$

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

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

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

$$\frac{\Gamma, v : B \vdash e : \mathtt{Bool}^{\Downarrow}}{\Gamma \vdash \{v : B \mid e\}}\ \textsc{WF-Base}$$

$$\frac{\Gamma \vdash \tau_x \qquad \Gamma, x : \tau_x \vdash \tau}{\Gamma \vdash x \to \tau_x \tau}\ \textsc{WF-Fun}$$

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

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

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

Algorithmic-Subtyping $\boxed{\Gamma \vdash_S \tau_1 \preceq \tau_2}$

$$\frac{\begin{array}{c}\Gamma' \doteq \Gamma, v : \{v : B^{\Downarrow} \mid e\} \\ \Gamma' \vdash e' \rightsquigarrow p' \qquad \mathsf{SmtValid}((\!|\Gamma'|\!) \Rightarrow p')\end{array}}{\Gamma \vdash_S \{v : B \mid e\} \preceq \{v : B \mid e'\}}\ \preceq\textsc{-Base-}\lambda^S$$

Fig. 2. Typing of λ^R and algorithmic subtyping of λ^S

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:

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

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

Rules All but two of the rules are standard [**??**]. First, rule T-Refl is used to strengthen the type of each reflected binder with its definition, as described previously in § 2.4. 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 [**??**] and is required to equate the reflected functions with their definitions. For example, the application `fib 1` is typed as $\{v : \mathtt{Int}^{\Downarrow} \mid \mathtt{fibP}\ v\ 1 \wedge v = \mathtt{fib}\ \ 1\}$ where the first conjunct comes from the (reflection-strengthened) output refinement of `fib` § 1 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 **?**, τ is well-formed if all the refinements in τ are `Bool`-typed, provably terminating expressions in Γ.

Subtyping A judgment $\Gamma \vdash \tau_1 \preceq \tau_2$ states that the type τ_1 is a subtype of τ_2 in the environment Γ. 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 [**?**], in **?** we prove that evaluation preserves typing and typing implies denotational inclusion.

THEOREM 2.1. *[Soundness of λ^R]*

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

Theorem 2.1 lets us interpret well typed programs as proofs of propositions. For example, in § 1 we verified that the term `fibUp` proves

$$n \to \texttt{Nat}\{\texttt{fib}\ n \leq \texttt{fib}\ (n+1)\}$$

Via soundness of λ^R, we get that for each valid input `n`, the result refinement is valid.

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

3 ALGORITHMIC CHECKING: λ^S

Next, we describe λ^S, a conservative, first order approximation of λ^R where higher order features are approximated with uninterpreted functions and the undecidable type subsumption rule $\preceq$-BASE-λ^R is replaced with a decidable one $\preceq$-BASE-λ^S, yielding an SMT-based algorithmic type system that is both sound and decidable.

Syntax: Terms & Sorts Figure 1 summarizes the syntax of λ^S, the *sorted* (SMT-) decidable logic of quantifier-free equality, uninterpreted functions and linear arithmetic (QF-EUFLIA) [**??**]. 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 $s \to s$. 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.

Semantics: Satisfaction & Validity An assignment σ maps variables to terms $\sigma \doteq \{x_1 \mapsto p_1, \ldots, x_n \mapsto p_n\}$. We write $\sigma \models p$ if the assignment σ is a *model of* p, intuitively if $\sigma \cdot p$ "is true" [**?**]. A predicate p *is satisfiable* if there exists $\sigma \models p$. A predicate p *is valid* if for all assignments $\sigma \models p$.

3.1 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 [**?**]. Here we discuss the non-identity ones.

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

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

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`.

$$\frac{\Gamma, x : \tau_x \vdash e \leadsto p \qquad \Gamma \vdash (\lambda x.e) : (x \rightarrow \tau_x \tau)}{\Gamma \vdash \lambda x.e \leadsto \texttt{lam}_{(\!|\tau|\!)}^{(\!|\tau_x|\!)}\ x\ p} - \models F \hookrightarrow^* u\textsc{n}$$

The term $\lambda x.e$ of type $\tau_x \rightarrow \tau$ is transformed to $\texttt{lam}_s^{s_x}\ x\ p$ of sort $\mathsf{Fun}\ s_x\ s$, where s_x and s are respectively $(\!|\tau_x|\!)$ and $(\!|\tau|\!)$, $\texttt{lam}_s^{s_x}$ is a special uninterpreted function of sort $s_x \rightarrow s \rightarrow \mathsf{Fun}\ s_x\ s$, 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 [?] using the uninterpreted app:

$$\frac{\Gamma \vdash e' \leadsto p' \qquad \Gamma \vdash e \leadsto p \qquad \Gamma \vdash e : \tau_x \rightarrow \tau}{\Gamma \vdash e\ e' \leadsto \mathsf{app}_{(\!|\tau|\!)}^{(\!|\tau_x|\!)}\ p\ p'} - \models A \hookrightarrow^* p\textsc{p}$$

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

$$\Gamma \vdash D \leadsto \mathsf{s}_D$$

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 ?.

To generalize, let D_i be a data constructor such that

$$\mathsf{prim}(D_i) \ \doteq\ \tau_{i,1} \rightarrow \cdots \rightarrow \tau_{i,n} \rightarrow \tau$$

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

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.

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

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 [?] to avoid universally quantified formulas and the resulting unpredictability.

3.2 Correctness of Translation from λ^R to λ^S

Informally, the translation relation $\Gamma \vdash e \leadsto p$ is correct in the sense that if e is a terminating boolean expression then e reduces to `True` *iff* p is SMT-satisfiable by a model that respects β-equivalence.

Definition 3.1 (β-Model). A β–model σ^β is an extension of a model σ where `lam` and `app` satisfy the axioms of β-equivalence:

$$\forall\, x\ y\ e.\ \mathtt{lam}\ x\ e\ =\ \mathtt{lam}\ y\ (e\theta xy) \qquad\qquad \forall\, x\ e_x\ e.\ \mathtt{app}\ (\mathtt{lam}\ x\ e)\ e_x\ =\ e\theta x e_x$$

Semantics Preservation We define the translation of a λ^R term into λ^S under the empty environment as $(\!| e |\!) \doteq p$ *iff* $\emptyset \vdash e \leadsto p$. A *lifted substitution* $\theta^\perp$ is a set of models σ where each "bottom" in the substitution θ is mapped to an arbitrary logical value of the respective sort [?]. We connect the semantics of λ^R and λ^S via the following:

Theorem 3.2. *If* $\Gamma \vdash e \leadsto p$, *then for every* $\mathtt{sub}(\in, [\![)]\!]\Gamma$ *and every* $\sigma \in \mathtt{sub}(\cdot\ \perp)$, *if* $\mathtt{sub}(\cdot\ \perp) \cdot e \hookrightarrow^\star v$ *then* $\sigma^\beta \models p = (\!| v |\!)$.

Corollary 3.3. *If* $\Gamma \vdash e : \mathtt{Bool}$, *$e$ reduces to a value and* $\Gamma \vdash e \leadsto p$, *then for every* $\mathtt{sub}(\in, [\![)]\!]\Gamma$ *and every* $\sigma \in \mathtt{sub}(\cdot\ \perp)$, $\mathtt{sub}(\cdot\ \perp) \cdot e \hookrightarrow^\star \mathtt{True}$ *iff* $\sigma^\beta \models p$.

3.3 Decidable Type Checking

We make the type checking from Figure 2 decidable by checking subtyping via an SMT solver.

Verification Conditions The implication or *verification condition* (VC) $(\!|\Gamma|\!) \Rightarrow p$ 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 [?]:

$$(\!|\Gamma|\!) \doteq \bigwedge_{x\in\Gamma} (\!|\Gamma, x|\!) \quad \text{where we embed each binder as} \quad (\!|\Gamma, x|\!) \doteq \begin{cases} p & \text{if } \Gamma(x) = \{x : B^{\Downarrow} \mid e\},\ \Gamma \vdash e \leadsto p \\ \mathtt{True} & \text{otherwise.} \end{cases}$$

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 version $\preceq$-Base-λ^S of Figure 2 that uses an SMT solver to check the validity of the subtyping VC. We use Corollary 3.3 to prove soundness of subtyping.

Lemma 3.4. *If* $\Gamma \vdash_S \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}$ *then* $\Gamma \vdash \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}$.

Soundness of λ^S We write $\Gamma \vdash_S e : \tau$ 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 3.4 implies the soundness of λ^S.

THEOREM 3.5 (SOUNDNESS OF λ^S). *If* $\Gamma \vdash_S e : \tau$ *then* $\Gamma \vdash e : \tau$.

Note that we *do not* require the β-equivalence axioms for soundness: if a VC is valid *without* the β-equivalence axioms, *i.e.* when `lam` and `app` are uninterpreted, then the VC is trivially valid *with* the axioms, and hence, by Theorem 3.5 the program is safe.

4 PROOF BY EVALUATION

Our terms are measure application, constructor application, and variables.

$$e ::= x \mid C\ \bar{e} \mid m\ \bar{e}$$

Measures are represented as a list of guarded lambdas $[\langle p, \lambda x.e \rangle, \ldots]$, where p is predicate that guards evaluation and the type checker can use to prove termination

We have the following contexts:

$$\Gamma ::= \bullet \mid \Gamma; i : T \mid \Gamma; m : \langle p, \lambda x.e \rangle$$
$$\Delta ::= \bullet \mid \Delta; \phi$$
$$\Sigma ::= \bullet \mid \Sigma; \star \mid \Sigma; t$$

Γ, has our measures, and carries around refinement type information for free variables in scope. Δ just exists to dump all the predicates into, and Σ hold terms awaiting PBE. $\star$ in Σ runs PBE on the term being scrutinized.

We have the following operational semantics:

$$\frac{\Gamma \vdash p(x)}{\langle \Gamma; (m : \langle p, \lambda \bar{x}.e \rangle), m\ \bar{e} \rangle \hookrightarrow \langle \Gamma; (m : \langle p, \lambda \bar{x}.e \rangle), e[\bar{e}/\bar{x}] \rangle}$$
$$\langle \Gamma, x \rangle \hookrightarrow \langle \Gamma, x \rangle$$
$$\frac{\langle \Gamma, e_i \rangle \hookrightarrow \left\langle \Gamma, e_i' \right\rangle}{\langle \Gamma, C\bar{e} \rangle \hookrightarrow \langle \Gamma, C\bar{e}' \rangle}$$

We proceed to define the judgements for $\vdash$, $\vdash_n$, and $\vdash^*$:

$\vdash p$ is just EUF, so we have the following rules:

$$\frac{\vdash a = b}{\vdash b = a}[sym]$$

$$\frac{}{\vdash a = a}[refl]$$

$$\frac{\vdash a = b}{\vdash f(a) = f(b)}[cong]$$

$$\frac{\vdash a = b \qquad \vdash b = c}{\vdash a = c}[trans]$$

$\Gamma, \Delta \vdash_n p$ has semantics:

$$\frac{\vdash p}{\Gamma, \Delta \vdash_0 p}[0]$$

$$\frac{\Gamma, \Delta \vdash_0 \phi \to p}{\Gamma, \Delta; \phi \vdash_0 p}[\Delta]$$

$$\frac{\vdash \Delta \to q(\bar{e}) \qquad \Gamma; m : \langle q, \lambda \bar{x}.e \rangle, \Delta; m\ \bar{e} = e[\bar{e}/\bar{x}] \vdash_n p}{\Gamma; m : \langle q, \lambda \bar{x}.e \rangle, \Delta \vdash_{n+1} p}[n]$$

Furthermore, we write $\Gamma, \Delta \vdash_* \phi$ if $\exists n | \Gamma, \Delta \vdash_n \phi$

$\Gamma, \Delta, \Sigma \vdash^* p$ defines our PBE semantics and is more involved:

$$\frac{\Gamma, \Delta, \Sigma; \phi \vdash^* \phi}{\Gamma, \Delta, \Sigma; \star \vdash^* \phi}$$

$$\frac{\vdash \Delta \rightarrow \phi}{\Gamma, \Delta, \Sigma \vdash^* \phi}$$

$$\frac{\Gamma; x : \{x : T | R(v)\}, \Delta; R(x), \Sigma \vdash^* \phi}{\Gamma; x : \{x : T | R(v)\}, \Delta, \Sigma; x \vdash^* \phi}$$

$$\frac{\Gamma; C : \bar{x} \rightarrow \{v : T | R(\bar{x})\}, \Delta; R(\bar{e}), \Sigma; \bar{e} \vdash^* \phi}{\Gamma; C : \bar{x} \rightarrow \{v : T | R(\bar{x})\}, \Delta, \Sigma; C\ \bar{e} \vdash^* \phi}$$

$$\frac{\Gamma; m : \langle q, \lambda \bar{x}.e \rangle, \Delta, \Sigma; \bar{e} \vdash^* q(\bar{e}) \qquad \Gamma; m : \langle q, \lambda \bar{x}.e \rangle, \Delta; m\ \bar{e} = e[\bar{e}/\bar{x}], \Sigma; e[\bar{e}/\bar{x}] \vdash^* p}{\Gamma; m : \langle q, \lambda \bar{x}.e \rangle, \Delta, \Sigma; m\ \bar{e} \vdash^* p}$$

Theorem 4.1. $\Gamma, \bullet, \star \vdash^* t = t'$ *iff* $\Gamma, \bullet \vdash_* t = t'$

Proof. *Sketch.* $\Gamma, \bullet \vdash_* t = t'$ means there exists an n such that $\Gamma, \bullet \vdash_n t = t'$. We proceed by induction on n:

In the case where $n = 0$, the only proof of $\Gamma, \bullet \vdash_0 t = t'$ is $\frac{\vdash t=t'}{\Gamma, \bullet \vdash_0 t=t'}$, so we have $\frac{\vdash t=t'}{\Gamma, \bullet, \star \vdash t=t'}$

In the inductive case, given

$$\forall t, t', \Gamma, \bullet \vdash_n t = t' \implies \Gamma, \bullet, \star \vdash^* t = t'$$

we want to show

$$\Gamma, \bullet \vdash_{n+1} t = t' \implies \Gamma, \bullet, \star \vdash^* t = t'$$

Destruction on $\vdash_{n+1}$ gives us:

$$\Gamma, \bullet; m\ \bar{e} = e[\bar{e}/\bar{x}] \vdash_n t = t' \implies \Gamma, \bullet, \star \vdash^* t = t'$$

If $\Gamma, \bullet \vdash_n t = t'$, then we're done, by the induction hypothesis. If $\Gamma, \bullet \nvdash_n t = t'$, but $\Gamma, \bullet; m\ \bar{e} = e[\bar{e}/\bar{x}] \vdash_n t = t'$, then there exist $p_1, \ldots, p_n$, such that $\nvdash p_1 \rightarrow \ldots \rightarrow p_n \rightarrow t = t'$ but $\vdash m\ \bar{e} = e[\bar{e}/\bar{x}] \rightarrow p_1 \rightarrow \ldots \rightarrow p_n \rightarrow t = t'$

Now, consider the derivation tree of $t = t'$ under the semantics of the $\vdash$ judgement. Assume for the sake of contradiction that $m\ \bar{e}$ is not a subterm of $p_1 \rightarrow \ldots \rightarrow p_n \rightarrow t = t'$. Then there exists a proof of $t = t'$ from the premises $p_1 \ldots p_n, m\ \bar{e} = e[\bar{e}/\bar{x}]$ that does not mention $m\ \bar{e}$. But then $\vdash p_1 \rightarrow \ldots \rightarrow p_n \rightarrow t = t'$, a contradiction. So $m\ \bar{e}$ is a subterm of $p_1 \rightarrow \ldots \rightarrow p_n \rightarrow t = t'$.

Now we should that all such subterms are instantiated by proof by evaluation. It should be clear from inspecting the semantics that all subterms of $t = t'$ are instantiated, and by the last rule, each subterm of the p_is are instantiated.

Which gives us

$$\Gamma, \bullet, \bullet; t = t' \vdash^* m\ \bar{e} = e[\bar{e}/\bar{x}]$$

and by applying the inductive hypothesis to our premise, we have $\Gamma, \bullet, \star \vdash^* m\ \bar{e} = e[\bar{e}/\bar{x}] \rightarrow t = t'$ so we have $\Gamma, \bullet, \star \vdash^* t = t'$

□

$$\begin{array}{rrcll}
\textbf{\textit{Values}} & v & ::= & \mathrm{c} & \text{constants} \\
 & & \mid & \mathtt{C}(\overline{v}) & \text{ctor-application} \\
\textbf{\textit{Operators}} & \oplus & ::= & \{=,\ \wedge,\ \ldots\} & \\
\textbf{\textit{Terms}} & e, p, b & ::= & v & \text{values} \\
 & & \mid & x & \text{variables} \\
 & & \mid & e \oplus e & \text{prim-op} \\
 & & \mid & f(\overline{e}) & \text{func-application} \\
 & & \mid & \mathtt{C}(\overline{e}) & \text{ctor-application} \\
\textbf{\textit{Functions}} & F & ::= & \lambda\overline{x}.\langle\overline{p \Rightarrow b}\rangle & \\
\textbf{\textit{Definitions}} & \Psi & ::= & \emptyset \mid f \mapsto F, \Psi & \\
\textbf{\textit{Environments}} & G & ::= & \mathit{true} \mid p, G &
\end{array}$$

Fig. 3. Syntax of Predicates, Terms and Reflected Functions

```
def Apps($e$):
  return [$(f, \overline{e}) \mid f(\overline{e}) \prec e$]

def Init($G$, $q$):
  $tree$ = $\emptyset$
  Link($tree$, Root, Apps($q$))
  for $p$ in $G$:
    Link($tree$, Root, Apps($p$))
  return $tree$

def Link($t$, $n$, $ns$):
  $ns'$ = [$n' \mid n'$ in $ns$ if $n' \notin t$]
  if $ns' = \emptyset$:
    $t$ += ($n \rightarrow$ Done)
    return ($t$, False)
  else:
    for $n'$ in $ns'$:
      $t$ += ($n \rightarrow n'$)
      return ($t$, True)
```

Fig. 4. This is some code

4.1 Completeness of PBE

Semantics A *store* σ is a mapping from (free) variables to values. We write $\sigma(e)$ for the term obtained by substituting free variables x in e with the corresponding values in the store σ. We write $\Psi, \sigma \models e$ if $\Psi \models \sigma(e) \hookrightarrow^* \mathit{true}$.

```
def PBE($\Psi, G, p$):
  $tree$ = Init($G$, $p$)
  $cont$ = True
  while cont && $\neg$SmtValid($G \vdash p$):
    $cont$ = False
    for ($f, \overline{e}$) in Leaves($tree$):
      $q$ = Inst($\Psi$, $G$, $f$, $\overline{e}$)
      if $q \neq true$:
        $G$ = $G \wedge q$
        ($tree, cont$) = Link($tree$, ($f, \overline{e}$), Apps($q$))
  return SmtValid($G \vdash p$)

def Inst($\Psi$, $G$, $f$, $\overline{e}$):
  let $\lambda\overline{x}.\langle\overline{d}\rangle$ = $\Psi(f)$
  for $p \Rightarrow b$ in $\overline{d}$:
    if SmtValid($G \vdash p\,[\overline{e}/\overline{x}]$):
      return $(f(\overline{x}) = b)\,[\overline{e}/\overline{x}]$
  return $true$
```

Fig. 5. Proof by Evaluation

Subterm Evaluation The following lemma states that if a store satisfies a reflected function's guard, then the evaluation of the corresponding function body requires evaluating *every* sub-term inside the body.

LEMMA 4.2. *Let* $\Psi(f) \equiv \lambda\overline{x}.\langle\overline{p \Rightarrow b}\rangle$. *If* $\Psi, \sigma \models p_i$ *and* $g(\overline{e}) \prec b_i$ *then* $\Psi \models \sigma(b_i) \hookrightarrow^* C[g(\sigma(\overline{e}))]$.

Totality Assumption The following assumption states that if a store satisfies a reflected function's guard, then the corresponding body reduces to exactly one value. Both of these are checked via refinement typing [?].

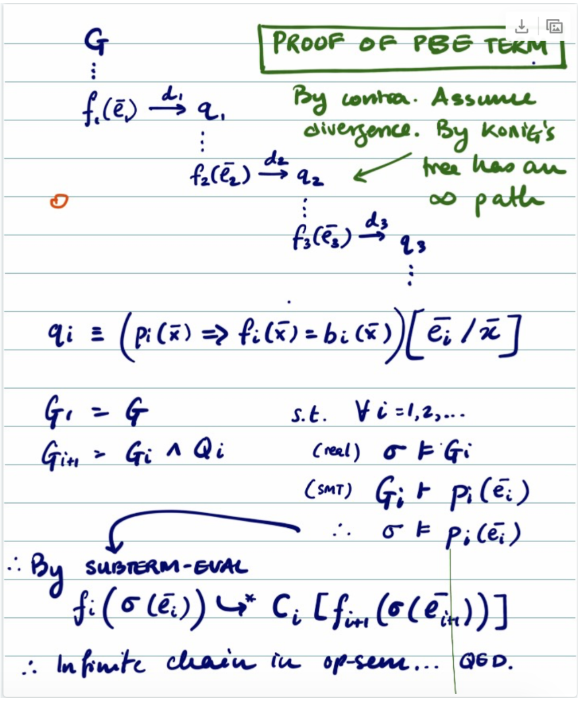
LEMMA 4.3 (**TOTALITY**). *Let* $\Psi(f) \equiv \lambda\overline{x}.\langle\overline{p \Rightarrow b}\rangle$.
(1) If $\Psi, \sigma \models p_i$ *then* $\Psi \models \sigma(b_i) \hookrightarrow^* v$.
(2) If $\Psi, \sigma \models p_i$, $\Psi, \sigma \models p_j$ *then* $i = j$.

Instantiation A term q is a Ψ*-instance* if $q \equiv (p_i \Rightarrow f(\overline{x}) = b_i)\,[\overline{e}/\overline{x}]$ for some f such that $\Psi(f) \equiv \lambda\overline{x}.\langle\overline{p \Rightarrow b}\rangle$. A set of terms Q is a Ψ*-instance* if every $q \in Q$ is an Ψ-instance.

LEMMA 4.4 (**SAT-INST**). *If* $\Psi, \sigma \models G$ *and* Q *is a* Ψ*-instance, then* $\Psi, \sigma \models G \wedge Q$.

Transparency We say G is *inhabited* if for some σ we have $\Psi, \sigma \models p$ for each $p \in G$. We say G is *inconsistent* if SmtValid($G \vdash$ *false*). We say G is *transparent* if it is either inhabited or inconsistent.

As an example of a *non-transparent* G_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 σ 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`.


PROOF OF PBE TERM

G

$f_1(\bar{e}_1) \xrightarrow{d_1} q_1$

$f_2(\bar{e}_2) \xrightarrow{d_2} q_2$

$f_3(\bar{e}_3) \xrightarrow{d_3} q_3$

By contra. Assume divergence. By König's tree has an ∞ path

$$q_i \equiv \left(p_i(\bar{x}) \Rightarrow f_i(\bar{x}) = b_i(\bar{x})\right)\left[\bar{e}_i / \bar{x}\right]$$

$G_1 = G$ s.t. $\forall i = 1, 2, \ldots$

$G_{i+1} = G_i \wedge Q_i$ (real) $\sigma \models G_i$

(SMT) $G_i \vdash p_i(\bar{e}_i)$

$\therefore \sigma \models p_i(\bar{e}_i)$

$\therefore$ By SUBTERM-EVAL

$$f_i(\sigma(\bar{e}_i)) \hookrightarrow^* C_i[f_{i+1}(\sigma(\bar{e}_{i+1}))]$$

$\therefore$ Infinite chain in op-sem... QED.

Normal We say G is *normal* if all applications in G are of the form $f(\overline{x})$. We can easily *normalize* a G by introducing fresh variables for each sub-term.

THEOREM 4.5 (**TERMINATION**). *If G is transparent and G, p is normal then* $\text{PBE}(\Psi, G, p)$ *terminates.*

THEOREM 4.6 (**SOUNDNESS & COMPLETENESS**). $\text{PBE}(\Psi, G, p)$ *iff* $\Psi, G \vdash_* p$.

5 ENCODING OF HIGHER ORDER LOGICS IN LIQUID HASKELL

NV:♣ To add 'Our constructions show how any natural deduction proof (Gentzen 1935) can be embedded as a Liquid Haskell proof.' You probably also want some discussion to contrast this to the related work you mention your reviewers keep comparing your work with.♣

Idea:

- We can express higher order logic given the syntax of Figure 6.
- You can prove properties because natural deduction rules type check (*i.e.* are safe).
- Some examples illustrate the proving method.

LIQUID HASKELL can express arbitrary higher order properties, *i.e.* has the same expressive power as Isabelle/HOL or Agda with a single universe type. For decidable type checking, refinements are first order, non-quantified expressions. We quantify refinements by encoding

- $\forall$ as a lambda abstraction and
- $\exists$ as a dependent pair

getting the HOL of Figure 6.

ϕ ::=		*Formulas:*
	$\{v : \texttt{Proof} \mid e\}$	*first order terms, with* $\texttt{True}, \texttt{False} \in e$
|	$\phi_1 \rightarrow \phi_2$	*implication:* $\phi_1 \Rightarrow \phi_2$
|	$\phi \rightarrow \{v : \texttt{Proof} \mid \texttt{False}\}$	*negation:* $!\phi$
|	$\texttt{PAnd}\ \phi_1\ \phi_2$	*conjunction:* $\phi_1 \land \phi_2$
|	$\texttt{POr}\ \phi_1\ \phi_2$	*disjunction:* $\phi_1 \lor \phi_2$
|	$x : a \rightarrow \phi$	*forall:* $\forall x.\phi$
|	$(x :: a, \phi)$	*exists:* $\exists x.\phi$

Fig. 6. Encoding of Higher Order Logic in LIQUID HASKELL types. Function binders are not represented in negation and implication where they are not relevant.

5.1 First Order Terms

The logical terms in LIQUID HASKELL are non-qualified Haskell expressions e as presented in Figure 1 of [?] (and defunctionalized in Figure 2 to the SMT logic). These expressions include constants, boolean operations, lambda abstractions, applications and in practice are extended to include decidable SMT theories, including non-qualified linear arithmetic and set theory. In the absence of reflected functions, reasoning over first order terms is automatically performed by the SMT-solver on decidable theories including linear arithmetic and congruence. When first order terms include reflected functions reasoning is performed via reflection of type level computations.

5.2 Implication

Implication $\phi_1 \Rightarrow \phi_2$ is encoded as a function from the proof of ϕ_1 to the proof of ϕ_2.

Implication Elimination. This encoding let us eliminate implication proofs by function application, thus safely encoding the natural deduction rule of modus ponens:

```
implElim :: p:Bool → q:Bool → {v:Proof | p} → ({v:Proof | p} → {v:
    Proof | q})
          → {v:Proof | q}
implElim _ _ p f = f p
```

Implication Refinement & Reification. If the formulas ϕ_1 and ϕ_2 are over basic expressions (non-qualified), that is $\phi_i \equiv \{v : \texttt{Proof} | e_i\}$, then implication can be directly encoded in the refinements as $\{v : \texttt{Proof} | e_1 \Rightarrow e_2\}$. We call this process refinement of the implication and the dual reification:

```
implRefine :: b1:Bool → b2:Bool
           → ({v:Proof | b1} → {v:Proof | b2})
           → {v:Proof | b1 ⇒ b2}
implRefine b1 _ fb
  | b1        = fb trivial
  | otherwise = trivial

implReify :: b1:Bool → b2:Bool
          → {v:Proof | b1 ⇒ b2}
          → ({v:Proof | b1} → {v:Proof | b2})
implReify _ _ b1b2 b1 = trivial
```

5.3 Negation

Negation is encoded as an implication to the proof of false.

Negation Refinement & Reification. We reify negation by trivially proving using SMT that for each property b both b and its negation imply false.

```
type False = {v:Proof | false }

notReify :: b:Bool → {v:Proof | not b} → ({v:Proof | b} → False)
notReify _ notb b = trivial
```

To refine the negation of a property b, if b holds, then we apply its negation to get false., otherwise, the negation of b is trivially true.

```
notRefine :: b:Bool → ({v:Proof | b} → False) → {v:Proof | not b}
notRefine b f
  | b         = f trivial
  | otherwise = trivial
```

5.4 Conjunction

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
```

Conjunction Refinement & Reification. We refine the conjunction by opening the PAnd thus assuming both the conjuncts.

```
andRefine :: b1:Bool → b2:Bool → PAnd {v:Proof | b1} {v:Proof | b2}
          → {v:Proof | b1 && b2 }
andRefine _ _ (PAnd b1 b2) = trivial
```

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

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

Conjunction Natural Deduction Rules. To introduce conjunction we wrap the two proofs for the formulas ϕ_1 and ϕ_2.

```
andIntro :: b1:{Bool | b1} → b2:{Bool | b2} → PAnd {v:Proof | b1} {v:
    Proof | b2}
andIntro b1 b2 = PAnd trivial trivial
```

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

```
andElimLeft :: b1:Bool → b2:Bool → PAnd {v:Proof | b1} {v:Proof | b2}
            → {v:Proof | b1 }
andElimLeft _ _ (PAnd b1 b2) = b1

andElimRight :: b1:Bool → b2:Bool → PAnd {v:Proof | b1} {v:Proof | b2}
             → {v:Proof | b2 }
andElimRight _ _ (PAnd b1 b2) = b2
```

5.5 Disjunction

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

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

Disjunction Refinement & Reification. We refine the disjunction by case analyzing o the POr and getting either the left or the right disjunct.

```
orRefine :: b1:Bool → b2:Bool
         → POr {v:Proof | b1} {v:Proof | b2}  ]
         → {v:Proof | b1 || b2 }
orRefine _ _ (POrLeft  p1) = p1
orRefine _ _ (POrRight p2) = p2

orReify :: b1:Bool → b2:Bool
        → {v:Proof | b1 || b2 }
        → POr {v:Proof | b1} {v:Proof | b2}
orReify b1 b2 p
  | b1 = POrLeft  p
  | b2 = POrRight p
```

Disjunction Natural Deduction Rules. To introduce disjunction we wrap the proof for either the formula ϕ_1 or ϕ_2 using either the POrLeft or the POrRight constructors respectively.

```
orIntroLeft :: b1:Bool → b2:Bool → {v:Proof | b1}
            → POr {v:Proof | b1} {v:Proof | b2}
orIntroLeft _ _ p = POrLeft p

orIntroRight :: b1:Bool → b2:Bool → {v:Proof | b2}
             → POr {v:Proof | b1} {v:Proof | b2}
orIntroRight _ _ p = POrRight p
```

To eliminate conjunction we case analyzing the conjunction and use either the left or the right conjunct.

```
orElim :: p:Bool → q:Bool → r:Bool
       → POr {v:Proof | p} {v:Proof | q}
       → ({v:Proof | p} → {v:Proof | r})
       → ({v:Proof | q} → {v:Proof | r})
       →  {v:Proof | r}
orElim _ _ _ (POrLeft  p) fp _ = fp p
orElim _ _ _ (POrLeft q)  _ fq = fq q
```

5.6 Forall

Forall $\forall x.\phi$ is encoded as a lambda abstraction $x : a \rightarrow \phi$.

Forall introduction and elimination. Introductions and eliminations are encoded by lambda abstraction and application.

```
∀Elim :: p:(a → Bool) → (x:a → {v:Proof | p x} ) → y:a → {v:Proof |
    p y}
∀Elim _ f y = f y

∀Intro :: p:(a → Bool) → (t:a → {v:Proof | p t}) → (x:a → {v:Proof |
     p x})
∀Intro _ f = f
```

5.7 Exists

Existentials $\exists x.\phi$ is encoded as a dependent pair: a pair that contains x and a proof of a formula that depends on the first element x. In Liquid Haskell we name the first element of the pair as `(x::a, ` ϕ`)`. Internally dependent pairs are implemented via Abstract Refinement Types, while preserving decidable type checking.

Exists introduction and elimination. To introduce an existential we pack an element x with a proof that x satisfies a property $p\ x$.

```
existsIntro :: p:(a → Bool)
            → x:a → {v:Proof | p x}
            → (y::a,{v:Proof | p y})
existsIntro p x pr⊙ = (x, pr⊙)
```

To eliminate an existential we open the dependent pair.

```
existsElim :: x:Bool → p:(a → Bool) → (t::a,{v:Proof | p t})
           → (s:a → {v:Proof | p s}
           → {v:Proof | x})
           → {v:Proof | x } @-}

existsElim x p (t, pt) f = f t pt
```

6 EXAMPLES

We present some proofs of higher order propertied and present how such properties can extend specific theories (like lists). These and more examples can be found in https://github.com/nikivazou/LiquidHOL.

6.1 Existentials over disjunction

We prove distribution of existentials over disjunction:

$$(\exists x.(f\ x \lor g\ x)) \Rightarrow ((\exists x.f\ x) \lor (\exists x.g\ x))$$

The proof proceeds by existential case splitting and introduction:

```
existsOrDistr :: f:(a → Bool) → g:(a → Bool)
              → (x::a, POr {v:Proof | f x} {v:Proof | g x})
              → POr (x::a, {v:Proof | f x}) (x::a, {v:Proof | g x})
existsOrDistr f g (x,POrLeft fx)  = POrLeft  (x,fx)
existsOrDistr f g (x,POrRight fx) = POrRight (x,fx)
```

6.2 Foralls over conjunction

We prove distribution of foralls over conjunction:

$$(\forall x.(f\ x \wedge g\ x)) \Rightarrow ((\forall x.f\ x) \wedge (\forall x.g\ x)))$$

The proof proceeds by forall introduction and elimination:

```
∀AndDistr :: f:(a → Bool) → g:(a → Bool)
           → (x:a → PAnd {v:Proof | f x} {v:Proof | g x})
           → PAnd (x:a → {v:Proof | f x}) (x:a → {v:Proof | g x})
∀AndDistr f g andx
  = PAnd (\x → case andx x of PAnd fx _ → fx)
         (\x → case andx x of PAnd _ gx → gx)
```

6.3 Forall - exists over implication

We prove distribution of foralls over conjunction:

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

The proof proceeds by forall elimination and existential introduction:

```
∀ExistsImpl :: p:(a → Bool) → q:(a → a → Bool)
  → (x:a → (y::a, {v:Proof | p x} → {v:Proof | q x y} ))
  → (x:a → ({v:Proof | p x} → (y::a, {v:Proof | q x y})))
∀ExistsImpl p q f x px
  = case f x of
      (y, pxToqxy) → (y,pxToqxy px)
```

6.4 Even lists

As a last example we see how quantifiers interact with the reflected functions by proving that forall lists xs if there exists a 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:List a → (ys::List a, {v:Proof | xs == ys ++ ys })
           → (n::Int, {v:Proof | length xs == n + n})
even_lists xs (ys,pf) = (length ys, lenAppend ys ys &&& pf)

lenAppend :: xs:List a → ys:List a → {length (xs ++ ys) == length xs +
    length ys}
lenAppend N _          = trivial
lenAppend (Cons x xs) ys = lenAppend xs ys
```

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 7, 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

CATEGORY		LOC
I. Arithmetic		
Fibonacci	§ 1	48
Ackermann	[?]	280
II. Algebraic Data Types		
Fold Universal	[?]	105
III. Typeclasses	Fig 8	
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	[?]	133
Unification	[?]	200
V. Deterministic Parallelism		
Conc. Sets	§ ??	906
n-body	§ ??	930
Par. Reducers	§ ??	55
TOTAL		4155

Fig. 7. **Summary of Case Studies**

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

Fig. 8. **Summary of Verified Typeclass Laws**

(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 § 1 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}
```

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 [?]. 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 [?]. 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 8, for various user-defined instances summarized in Figure 7.

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 8. 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 [?]. 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 8), 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 [?]. 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 [?]. 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 [?]. Our work is inspired by Dafny's Verified Calculations [?], a framework for proving theorems in Dafny [?], 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* [?] 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) [?].

Dependent types Our work is also inspired by dependently typed systems like Coq [?] and Agda [?]. 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 [?], a Haskell-like, fully dependent type language with undecidable type checking. In a recent line of work **?** 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 **?** and **?** 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 [**?**] generates proofs by term rewriting and Halo [**?**] 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 [**?**] 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 [**??**], Kahn process networks [**?**], and synchronous dataflow [**?**]. Other models use synchronous updates to shared state, as in Esterel [**?**] or PRAM. Finally, work on type systems for permissions management [**??**], supports the development of *non-interfering* parallel programs that access disjoint subsets of the heap in parallel. Parallel functional programming is also non-interfering [**??**]. Irrespective of which theory is used to support deterministic parallel programming, practical implementations such as Cilk [**?**] or Intel CnC [**?**] 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.

9 CONCLUSIONS AND FUTURE WORK

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

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

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