Refinement Reflection
(or, how to turn your favorite language into a proof assistant using SMT)

Niki Vazou Ranjit Jhala
University of California, San Diego

Abstract

Refinement Reflection turns your favorite programming language into a proof assistant by reflecting 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, predictable and most importantly, programmer controllable way. In the logic, we encode functions and lambdas using uninterpreted symbols preserving SMT-based decidable verification. In the language, we provide a library of combinators that lets programmers compose proofs from basic refinements and function definitions. We have implemented our approach in the Liquid Haskell system, thereby converting Haskell into an interactive proof assistant, that we used to verify a variety of properties ranging from arithmetic properties of higher order, recursive functions to the Monoid, Applicative, Functor and Monad type class laws for a variety of instances.

1. Introduction

Wouldn’t it be great to write proofs of programs in your favorite language, by writing programs in your favorite language, allowing you to avail of verification, while reusing the libraries, compilers and run-times for your favorite language?

Refinement types [9, 26] offer a form of programming with proofs that can be retrofitted into several languages like ML [5, 24, 37], C [8, 25], Haskell [32], TypeScript [34] and Racket [13]. The retrofitting relies upon restricting refinements to so-called “shallow” specifications that correspond to abstract interpretations of the behavior of functions. For example, refinements make it easy to specify that the list returned by the append function has size equal to the sum of those of its inputs. These shallow specifications fall within decidable logical fragments, and hence, can be automatically verified using SMT based refinement typing.

Refinements are a pale shadow of what is possible with dependently typed languages like Coq, Agda and Idris which permit “deep” specification and verification. These languages come equipped with mechanisms that represent and manipulate the exact descriptions of user-defined functions. For example, we can represent the specification that the append function is associative, and we can manipulate (unfold) its definition to write a small program that constructs a proof of the specification. Dafny [15], F* [30] and Halo [35] take a step towards SMT-based deep verification, by encoding user-defined functions as universally quantified logical formulas or “axioms”. This axiomatic approach offers significant automation but is a devil’s bargain as by relying heavily upon brittle heuristics for “triggering” axiom instantiation, it gives away decidable, and hence, predictable verification [16].

Refinement Reflection In this paper, we present a new approach to retrofitting deep verification into existing languages. Our approach reconciles the automation of SMT-based refinement typing with decidable and predictable verification, and enables users to reify pencil-and-paper proofs simply as programs in the source language. Our key insight is dead simple: the code implementing a user-defined function can be reflected into the function’s (output) refinement type, thus converting the function’s (refinement) type signature into a deep specification of the functions behavior. This simple idea has a profound consequence: at uses of the function, the standard rule for (dependent) function application yields a precise, predictable and most importantly, programmer controllable means of instantiating the deep specification that is not tethered to brittle SMT heuristics. Reflection captures deep specifications as refinements, but poses challenges for the logic and language.

Logic: Algorithmic Verification Our first challenge: how can we encode terms from an expressive higher order language in a decidable refinement logic in order to retain decidable, and hence, predictable, verification? We address this problem by using ideas for defunctionalization from the theorem proving literature which encode functions and lambdas using uninterpreted symbols. This encoding lets us use (SMT-based) congruence closure to reason about equality (§ 4). Of course, congruence is not enough; in general, e.g. to prove two functions extensionally equal, we require facilities for manipulating function definitions.

Language: Proof Composition Thus, as we wish to retrofit proofs into existing languages, our second challenge: how can we design a library of combinators that lets programmers compose proofs from basic refinements and function definitions? We develop such a library, wherein proofs are represented simply as unit-values refined by the proposition that they are proofs of. Refinement reflection lets us unfold definitions simply by applying the function to the relevant inputs, and finally, we show how to build up sophisticated proofs using a small library of combinators that permit reasoning in an algebraic or equational style.

Implementation & Evaluation We have implemented our approach in the Liquid Haskell system, thereby retrofitting deep verification into Haskell, converting it into an interactive proof assistant. Liquid Haskell’s refinement types crucially allow us to soundly account for the dreaded bottom by checking that (refined) functions produce (non-bottom) values [32]. We evaluate our approach by using Liquid Haskell to verify a variety of properties including arithmetic properties of higher order, recursive functions, textbook theorems about functions on inductively defined datatypes, and the Monoid, Applicative, Functor and Monad type class laws for a variety of instances. We demonstrate that our proofs look very much like transcriptions of their pencil-and-paper analogues. Yet, the proofs are plain Haskell functions, where case-splitting and induction are performed by plain pattern-matching and recursion.

To summarize, this paper describes a means of retrofitting deep specification and verification into your favorite language, by making the following contributions:
• We start with an informal description of refinement reflection, and how it can be used to prove theorems about functions, by writing functions (§ 2).
• We formalize refinement reflection using a core calculus, and prove it sound with respect to a denotational semantics (§ 3).
• We show how to keep type checking decidable (§ 4) while using uninterpreted functions and defunctionalization to reason about extensional equality in higher-order specifications (§ 5).
• Finally, we have implemented refinement reflection in Liquid Haskell, a refinement type system for Haskell. We develop a library of (refined) proof combinators and evaluate our approach by proving various theorems about recursive, higher-order functions operating over integers and algebraic data types (§ 6).

2. Overview
We begin with a fast overview of refinement reflection and how it allows us to write proofs of and by Haskell functions.

2.1 Refinement Types
First, we recall some preliminaries about refinement types and how they enable shallow specification and verification.

Refinement types are the source program’s (here Haskell’s) types decorated with logical predicates drawn from a (n SMT decidable) logic [9, 26]. For example, we can refine Haskell’s Int datatype with a predicate \( 0 \leq v \), to get a Nat type:

\[
\text{type Nat} = \{v: \text{Int} \mid 0 \leq v\}
\]

The variable \( v \) names the value described by the type, hence the above can be read as the “set of Int values \( v \) that are greater than 0”. The refinement is drawn from the logic of quantifier free linear arithmetic and uninterpreted functions (QF-UFLIA [4]).

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

\[
\begin{align*}
\text{fib} &:: \text{Nat} \rightarrow \text{Nat} \\
\text{fib} \ n & = \begin{cases} \\
\text{fib} \ (n-1) \ + \ \text{fib} \ (n-2) & \text{if} \ n \geq 2 \\
0 & \text{if} \ n = 0 \\
1 & \text{if} \ n = 1
\end{cases}
\end{align*}
\]

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 type’s refinement specifies a post-condition that the result is also a Nat. Thus refinement type checking, lets us specify and (automatically) verify the shallow property that if fib is invoked with non-negative Int values, then it terminates and yields a non-negative value.

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

\[
\text{type Prop} = ()
\]

but which can be refined with desired propositions about the code. For example, the following states the proposition 2 + 2 equals 4.

\[
\text{type Plus_2_2_eq_4} = \{v: \text{Prop} \mid 2 + 2 = 4\}
\]

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

\[
\text{type Plus_2_2_eq_4} = \{2 + 2 = 4\}
\]

We represent universally quantified propositions as function types:

\[
\text{type Plus_com} = x: \text{Int} \rightarrow y: \text{Int} \rightarrow (x+y = y+x)
\]

Here, the parameters \( x \) and \( y \) refer to input values; any inhabitant of the above type is a proof that Int addition is commutative.

Proofs We can now prove the above theorems simply by writing Haskell programs. To ease this task Liquid Haskell provides primitives to construct proof terms by “casting” expressions to Prop.

\[
\begin{align*}
\text{data QED} & = \text{QED} \\
(\ast\ast) &:: a \rightarrow \text{QED} \rightarrow \text{Prop} \\
_\ast\ast & = ()
\end{align*}
\]

To resemble mathematical proofs, we make this casting post-fix. Thus, we can write \( e \ast\ast \text{QED} \) to cast \( e \) to a value of Prop. For example, we can prove the above propositions simply by writing

\[
\begin{align*}
\text{pf_plus_2_2} &:: \text{Plus_2_2_eq_4} \\
\text{pf_plus_2_2} & = \text{trivial} \ast\ast \text{QED}
\end{align*}
\]

\[
\begin{align*}
\text{pf_plus_comm} &:: \text{Plus_comm} \\
\text{pf_plus_comm} & = \\lambda x y. \text{trivial} \ast\ast \text{QED}
\end{align*}
\]

\[
\text{trivial} = ()
\]

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

\[
2 + 2 = 4 \\
\forall x y. x + y = y + x
\]

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 feeling a bit anxious about whether the dreaded “bottom”, which inhabits all types, makes our proofs suspect. Fortunately, as described in [32], Liquid Haskell ensures that all terms with non-trivial refinements provably evaluate to (non-bottom) values, thereby making our proofs sound.

2.2 Refinement Reflection

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

\[
\text{type fib2_eq_1} = \{ \text{fib} 2 = 1 \}
\]

Standard refinement type checking runs into two problems. First, for decidability and soundness, arbitrary user-defined functions do not belong the refinement logic, i.e. we cannot even refer to fib in a refinement. Second, the only information that a refinement type checker has about the behavior of fib is its shallow type specification Nat → Nat which is far too weak to verify fib2_eq_1. To address both problems, we use the following annotation, which sets in motion the three steps of refinement reflection:

\[
\text{reflect fib}
\]

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; as far as the logic is concerned, fib only satisfies the congruence axiom

\[
\forall n, m. n = m \Rightarrow \text{fib} n = \text{fib} m
\]

On its own, the uninterpreted function is not terribly useful, 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:
where \( \text{fibP} \) is an alias for a refinement 
automatically derived from the function's definition:

\[
\text{predicate fibP } v \, n = \\
v = \text{if } n = 0 \text{ then } 0 \text{ else } \\
\text{ if } n = 1 \text{ then } 1 \text{ else } \\
\text{fib}(n-1) + \text{fib}(n-2)
\]

Step 3: Application With the reflected refinement type, each application of \( \text{fib} \) in the code automatically unfolds the \( \text{fib} \) definition once during refinement type checking. We prove \( \text{fib2_eq}_1 \) by:

\[
\begin{align*}
\text{pf_fib2} &:: \{ \text{fib } 2 = 1 \} \\
\text{pf_fib2} &:: \text{fib } 2 \Rightarrow \text{fib } 1 + \text{fib } 0 \quad \text{** QED}
\end{align*}
\]

We write \( f \) to denote places where the unfolding of \( f \)'s definition is important. The proof is verified as the above is A-normalized to:

\[
\begin{align*}
\text{let } \{ t_0 = \text{fib } 0; t_1 = \text{fib } 1; t_2 = \text{fib } 2 \} \\
in \{ t_2 = t_1 + t_0 \} \quad \text{** QED}
\end{align*}
\]

Which via standard refinement typing, yields the following verification condition that is easily discharged by the SMT solver, even though \( \text{fib} \) is uninterpreted:

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

Note that the verification of \( \text{pf_fib2} \) relies merely on the fact that \( \text{fib} \) was applied to \( i.e. \) unfolded at \( 0, 1 \) and \( 2 \). The SMT solver can automatically combine the facts, once they are in the antecedent. Hence, the following would also be verified:

\[
\begin{align*}
(\text{fib}(\text{fib } 0) 0) \land (\text{fib}(\text{fib } 1) 1) \land (\text{fibP}(\text{fib } 2) 2) \\
\Rightarrow (\text{fib } 2 = 1)
\end{align*}
\]

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

Reflection vs. Axiomatization An alternative axiomatic approach, used by Dafny, F* and HALO, is to encode the definition of \( \text{fib} \) as a universally quantified SMT formula (or axiom):

\[
\forall n. \text{fibP}(\text{fib } n) n
\]

Axiomatization offers greater automation than reflection. Unlike Liquid Haskell, Dafny will verify the equivalence of the following by automatically instantiating the above axiom at \( 2, 1 \) and \( 0 \):

\[
\begin{align*}
\text{axPf_fib2} &:: \{ \text{fib } 2 = 1 \} \\
\text{axPf_fib2} &:: \text{trivial} \quad \text{** QED}
\end{align*}
\]

However, the presence of such axioms renders checking the VCs undecidable. In practice, automatic axiom instantiation can easily lead to infinite “matching loops”. For example, the existence of a term \( \text{fib} n \) in a VC can trigger the above axiom, which may then produce the terms \( \text{fib}(n-1) \) and \( \text{fib}(n-2) \), 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 [1] 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. In short, the undecidability of the VC checking and its attendant heuristics makes verification unpredictable [16].

2.3 Structuring Proofs

In contrast to the axiomatic approach, with refinement reflection, the VCs are deliberately designed to always fall in an SMT-decidable logic, as function symbols are uninterpreted. It is upto the programmer to unfold the definitions at the appropriate places, which we have found, with careful design of proof combinators, to be quite a natural and pleasant experience. To this end, we have developed a library of proof combinators that permits reasoning about equalities and linear arithmetic, inspired by Agda [18].

“Equation” Combinators We equip Liquid Haskell with a family of equation combinators \( \odot \) for each logical operator \( \odot \) in \( \{=, \neq, \leq, <, \leq, >, \geq, \geq \} \), the operators in the theory QF-UFIA. 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 \( \odot . as:

\[
(=.) :: x:a \rightarrow y: (a | x=y) \rightarrow (v:a | v=x) \\
x =. _ = x
\]

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

\[
\begin{align*}
eqPf_fib2 &:: \{ \text{fib } 2 = 1 \} \\
eqPf_fib2 &:: \text{fib } 2 \\
= &. \text{fib } 1 + \text{fib } 0 \\
= &. 1 \\
\quad \quad \quad \text{** QED}
\end{align*}
\]

“Because” Combinators Often, we need to compose “lemmas” into larger theorems. For example, to prove \( \text{fib } 3 = 2 \) we may wish to reuse \( \text{eqPf_fib2} \) as a lemma. To this end, Liquid Haskell has a “because” combinator:

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

The operator is simply an alias for function application that lets us write \( x \odot. y : p \) (instead of \( \odot (x \cdot y) \)) where \( \odot \) is extended to accept an optional third proof argument via Haskell's type class mechanisms. We can use the because combinator to prove that \( \text{fib } 3 = 2 \) just by writing plain Haskell code:

\[
\begin{align*}
eqPf_fib3 &:: \{ \text{fib } 3 = 2 \} \\
eqPf_fib3 &:: \text{fib } 3 \\
= &. \text{fib } 2 + \text{fib } 1 \\
= &. 2 \\
\quad \quad \quad \text{eqPf_fib2} \\
\quad \quad \quad \text{** QED}
\end{align*}
\]

Arithmetic and Ordering SMT based refinements let us go well beyond just equational reasoning. Next, lets see how we can use arithmetic and ordering to prove that \( \text{fib} \) is (locally) increasing, \( i.e. \) for all \( n, \text{fib } n \leq \text{fib } (n+1) \):

\[
\begin{align*}
\text{fibUp} &:: n:Nat \rightarrow \{ \text{fib } n \leq \text{fib } (n+1) \} \\
\text{fibUp} &n \\
| n == 0 = \text{fib } 0 <. \text{fib } 1 \\
\quad \quad \quad \text{** QED} \\
| n == 1 = \text{fib } 1 \leq. \text{fib } 1 + \text{fib } 0 \leq. \text{fib } 2 \\
\quad \quad \quad \text{** QED} \\
| \text{otherwise} = \text{fib } n \\
\quad =. \text{fib } (n-1) + \text{fib } (n-2) \\
\quad \leq. \text{fib } n + \text{fib } (n-1) :: \text{fibUp } (n-1) \\
\quad \leq. \text{fib } n + \text{fib } (n-1) :: \text{fibUp } (n-2) \\
\quad \leq. \text{fib } (n+1) \\
\quad \quad \quad \text{** QED}
\end{align*}
\]

Case Splitting and Induction The proof \( \text{fibUp} \) 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 \( \text{fib} \) at those inputs. The derived VCs are
(automatically) proved as the SMT solver concludes 0 ≤ 1 and 1 + 0 ≤ 1 respectively. In the inductive case, \( \text{fib} \, n \) is unfolded to \( \text{fib} \, (n-1) + \text{fib} \, (n-2) \), which, because of the induction hypothesis (applied by invoking \( \text{fibUp} \) at \( n-1 \) and \( n-2 \)), and the SMT solvers arithmetic reasoning, completes the proof.

**Higher Order Theorems** Refinements smoothly accomodate higher-order reasoning. For example, lets prove that every locally increasing function is monotonic, i.e. if \( f \, z \leq f \, (z+1) \) for all \( z \), then \( f \, x \leq f \, y \) for all \( x < y \).

\[
\begin{align*}
\text{fMono} :: f : (\text{Nat} \rightarrow \text{Int}) & \rightarrow \text{fUp} : (z : \text{Nat} \rightarrow \{ f \, z \leq f \, (z+1) \}) \\
& \rightarrow x : \text{Nat} \\
& \rightarrow y : \{ x < y \} \\
& \rightarrow \{ f \, x \leq f \, y \} / \{ y \}
\end{align*}
\]

\[
fMono \, f \, \text{inc} \, x \, y \\
\mid x \, + \, 1 \, \Rightarrow \, y \, = \, f \, x \, \leq \, f \, (x+1) \, \Rightarrow \, \text{fUp} \, x \, \leq \, f \, y \\
\Rightarrow \, \text{QED}
\]

\[
\mid x \, + \, 1 \, < \, y \\
\Rightarrow \, f \, x \, \leq \, f \, (y-1) \, \Rightarrow \, \text{fMono} \, f \, \text{fUp} \, x \, (y-1) \, \leq \, f \, y \\
\Rightarrow \, \text{QED}
\]

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 [32]. If \( x+1 = y \), then we use \( \text{fUp} \, x \). Otherwise, \( x+1 < y \), and we use the induction hypothesis i.e. apply \( \text{fMono} \) at \( y-1 \), after which transitivity of the less-than ordering finishes the proof. We can use the general \( \text{fMono} \) theorem to prove that \( \text{fib} \) increases monotonically:

\[
\begin{align*}
\text{fibMono} :: n : \text{Nat} & \rightarrow m : \{ n = \text{m} \} \\
& \rightarrow (\text{fib} \, n \leq \text{fib} \, \text{m}) \\
\text{fibMono} \, f \, \text{fib} \, \text{fibUp}
\end{align*}
\]

2.4 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:

\[
\text{data} \, \text{Peano} = \text{Z} \mid \text{S} \, \text{Peano}
\]

We can add two Peano numbers via:

\[
\begin{align*}
\text{reflect} \, \text{add} & : \text{Peano} \rightarrow \text{Peano} \rightarrow \text{Peano} \\
\text{add} \, \text{Z} \, \text{m} & = \text{m} \\
\text{add} \, (\text{S} \, n) \, \text{m} & = \text{S} \, (\text{add} \, n \, \text{m})
\end{align*}
\]

In \( \S \, 3.5 \) we will describe exactly how the reflection mechanism (illustrated via \( \text{fibP} \)) is extended to account for ADTs like \( \text{Peano} \). Note that Liquid Haskell automatically checks that \( \text{add} \) is total [32], which lets us safely \text{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:

\[
\begin{align*}
\text{zeroL} :: n : \text{Peano} & \rightarrow \{ \text{add} \, \text{z} \, \text{n} \, = \, \text{n} \} \\
\text{zeroL} \, n & = \text{add} \, \text{z} \, \text{n} \\
\Rightarrow \, \text{QED}
\end{align*}
\]

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

\[
\begin{align*}
\text{zeroR} :: n : \text{Peano} & \rightarrow \{ \text{add} \, \text{z} \, \text{n} \, = \, \text{n} \} \\
\text{zeroR} \, \text{Z} & = \text{add} \, \text{Z} \, \text{Z}
\end{align*}
\]

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 \( \text{add} \). 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 \( \text{zeroR} \) is well-founded (\( \S \, 3 \)).

**Commutativity** Lets conclude by proving that \( \text{add} \) is commutative:

\[
\begin{align*}
\text{add_com} :: a : \_ & \rightarrow b : \_ \rightarrow \{ \text{add} \, a \, b \, = \, \text{add} \, b \, a \}
\end{align*}
\]

\[
\begin{align*}
\text{add_com} \, \text{Z} \, \text{b} & = \text{add} \, \text{Z} \, \text{b} \\
& = \, \text{b} \\
\Rightarrow \, \text{add_com} \, \text{b} \, \text{Z} \, \Rightarrow \, \text{zeroR} \, \text{b} \\
& = \, \text{sucR} \, \text{b} \, \text{a} \\
\Rightarrow \, \text{QED}
\end{align*}
\]

using a lemma \( \text{sucR} \):

\[
\begin{align*}
\text{sucR} :: n : \text{Peano} & \rightarrow m : \text{Peano} \rightarrow \\
& \{ \text{add} \, n \, (\text{S} \, \text{m}) \, = \, \text{S} \, (\text{add} \, n \, \text{m}) \}
\end{align*}
\]

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.

3. Refinement Reflection

Our first step towards formalizing refinement reflection is a core calculus \( \lambda^R \) with an undecidable type system based on denotational semantics. We show how the soundness of the type system allows us to prove theorems using \( \lambda^R \).

3.1 Syntax

Figure 1 summarizes the syntax of \( \lambda^R \), which is essentially the calculus \( \lambda^E \) [32] with explicit recursion and a special \text{reflect} binding form to denote terms that are reflected into the refinement logic. In \( \lambda^R \) refinements \( r \) are arbitrary expressions \( e \) (hence \( r ::= e \) in Figure 1). This choice allows us to prove preservation and progress, but renders typechecking undecidable. In \( \S \, 4 \) we will see how to recover decidability by soundly approximating refinements.

The syntactic elements of \( \lambda^R \) are layered into primitive constants, values, expressions, binders and programs.

**Constants** The primitive constants of \( \lambda^R \) include all the primitive logical operators \( \circ \), here, the set \( \{ =, \leq \} \). Moreover, they include the primitive booleans \( \text{True}, \text{False} \), integers \(-1, 0, 1, \text{etc.} \), and logical operators \( \land, \lor, \lnot, \text{etc.} \).

**Data Constructors** We encode data constructors as special constants. For example the data type \([\text{Int}]\), which represents finite lists of integers, has two data constructors: \([\text{nil}]\) and \([\text{cons}]\).
the output of the primitive constant operation. For example, con-
verted to a value; in a single step the expression is reduced to
Application of a constant requires the argument be re-
Constants

Values

Expressions

Binders

Program

Basic Types

Refined Types

Values & Expressions The values of $\lambda^R$ include constants, $\lambda$-abstractions $\lambda x.e$, and fully applied data constructors $D$ that wrap values. The expressions of $\lambda^R$ include values and 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 can be reflected into the reflection logic, followed by a binder. The stratification of programs via binders is required so that arbitrary recursive definitions are allowed but cannot be inserted into the logic via refinements or reflection. (We can allow non-recursive let binders in $e$, but omit them for simplicity.)

3.2 Operational Semantics

Figure 1 summarizes the small step contextual $\beta$-reduction semantics for $\lambda^R$. We write $e \rightarrow e'$ if there exist $e_1, \ldots, e_j$ such that $e$ is $e_1$, $e'$ is $e_j$ and $\forall i, 1 \leq i < j$, we have $e_i \rightarrow e_{i+1}$. We write $e \leftrightarrow e'$ if there exists some finite $s$ such that $e \rightarrow^* e'$. We define $\approx_{\beta}$ to be the reflexive, symmetric, transitive closure of $\rightarrow_{\beta}$.

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. For example, consider $\approx_{\beta}$, the primitive equality operator on integers. We have $\delta(=, n) \approx_{\beta} n$ where $\delta(=, n, m)$ equals $\text{True}$ iff $m$ is the same as $n$. 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 $f'$ we have $(f = f') \leftrightarrow \text{True}$ iff $\forall x. f x \approx_{\beta} f' x$. We assume source terms only contain implementable equalities over non-function types; the above only appears in refinements and allows us to state and prove facts about extensional equality §5.2.

3.3 Types

$\lambda^R$ types include basic types, which are refined with predicates, and dependent function types. Basic types $B$ comprise integers, booleans, and a family of data-types $T$ (representing lists, trees, etc.) For example the data type $[\text{Int}]$ represents lists of integers.

We refine basic types with predicates (boolean valued expressions $e$) to obtain basic refinement types $[v : B | e]$. Finally, we have dependent function types $x : \tau \rightarrow \tau$ where the input $x$ has the type $\tau_x$ and the output $\tau$ may refer to the input binder $x$. We write $B$ to abbreviate $[v : B | \text{True}]$, and $\tau_x \rightarrow \tau$ to abbreviate $x : \tau_x \rightarrow \tau$ if $x$ does not appear in $\tau$. We use $\tau$ to refer to refinements.

Denotations Each type $\tau$ denotes a set of expressions $\{\tau\}$, that are defined via the dynamic semantics [14]. Let $[\tau]$ be the type we get if we erase all refinements from $\tau$ and $e : [\tau]$ be the standard typing relation for the typed lambda calculus. Then, we define the denotation of types as:

$$\{x : B | \tau\} \equiv \{e : e : B, if e \rightarrow^* w \ then \tau \rightarrow w \implies \tau\}$$

Thus, if $\tau \equiv x : \tau_x \rightarrow \tau$, then for every value $w \in [\tau_x]$, we require $\delta(e, w) \in [\tau \rightarrow w]$.

3.4 Reflection Reflection

The simple, but key idea in our work is to strengthen the output type of functions with a refinement that reflects the definition in the logic. We do this by treating each reflect-binder: reflect $f : \tau = e \in p$ as a let-rec-binder: let $f : \text{Reflect}(\tau, e) = e \in p$ during type checking (rule T-REFLECT in Figure 3). Reflect

We write Reflect$(\tau, e)$ for the reflection of term $e$ into the type $\tau$, defined by strengthening $\tau$ as:

$$\text{reflect} \rightarrow \tau \rightarrow \lambda y. e \equiv x : \tau_x \rightarrow \tau \rightarrow \text{Reflect}(\tau, e[y \mapsto x])$$

As an example, recall from §2 that the reflect $\text{fib}$ strengthens the type of $\text{fib}$ with the reflected refinement $\text{fib}$. Consequences for Verification Reflect has two consequences for verification. First, the reflected refinement is not trusted; it is itself verified (as a valid output type) during type checking. Second, instead of being tethered to quantifier instantiation heuristics or having to program “effectors” as in Dafny [15] or F* [30] the programmer can predictably “unfold” the definition of the function during a proof simply by “calling” the function, which we have found to be a very natural way of structuring proofs §6. 3.5 Refining & Reflecting Data Constructors with Measures

We assume that each data type is equipped with a set of measures which are unary functions whose (1) domain is the data type, and (2) body is a single case-expression over the datatype [32]:

$$\text{measure } f : \tau = \lambda x.c e \ y = x \circ \{D, e \rightarrow e_i\}$$

For example, $\text{len}$ measures the size of an $[\text{Int}]$:
measure \( \text{len} :: [\text{Int}] \rightarrow \text{Nat} \)
\[
\text{len} = \lambda x \rightarrow \text{case } x \text{ of } \\
[\varepsilon] \rightarrow 0 \\
(x:xs) \rightarrow 1 + \text{len} \hspace{0.1cm} xs
\]

Checking and Projection We assume the existence of measures that check the top-level constructor, and project their individual fields. In §4.2 we show how to use these measures to reflect functions over datatypes. For example, for lists, we assume the existence of measures:
\[
\begin{align*}
is\text{Nil} & \quad \text{[]} = \text{True} \\
is\text{Nil} & \quad (x:xs) = \text{False} \\
is\text{Cons} & \quad (x:xs) = \text{True} \\
is\text{Cons} & \quad [] = \text{False} \\
sell1 & \quad (x:xs) = x \\
sell2 & \quad (x:xs) = xs
\end{align*}
\]

Refining Data Constructors with Measures We use measures to strengthen the types of data constructors, and use these strengthened types during construction and destruction (pattern-matching). Let: (1) \( D \) be a data constructor, with \( 	ext{unrefined type } \tau : \exists \rightarrow T \) (2) the \( i \)-th measure definition with domain \( T \) is:
\[
\text{measure } f_i : \tau = \lambda x.\text{case } y = x \text{ of } \{ D \tau \rightarrow e_i \}
\]

Then, the refined type of \( D \) is defined:
\[
\text{Ty}(D) \doteq \tau : \exists \rightarrow \{ v : T \mid \land_i f_i v = e_i[\tau \rightarrow \exists] \}
\]

Thus, each data constructor’s output type is refined to reflect the definition of each of its measures. For example, we use the measures \( \text{len} \), \( \text{is\text{Nil}} \), \( \text{is\text{Cons}} \), \( \text{sell1} \), and \( \text{sel2} \) to strengthen the types of [] and : to:
\[
\begin{align*}
\text{Ty}([]) & \doteq \{ v : [\text{Int}] \mid r_1 \} \\
\text{Ty}(\varepsilon) & \doteq x : \text{Int} \rightarrow xs : [\text{Int}] \rightarrow \{ v : [\text{Int}] \mid r \}
\end{align*}
\]

where the output refinements are:
\[
\begin{align*}
r_1 & \doteq \text{len } v = 0 \land \text{is\text{Nil} } v \land \neg \text{is\text{Cons} } v \\
r & \doteq \text{len } v = 1 + \text{len } xs \land \text{is\text{Nil} } v \land \text{is\text{Cons} } v \\
& \quad \land \text{sell1 } v = x \land \text{sel2 } v = xs
\end{align*}
\]

It is easy to prove that Lemma 1 holds for data constructors, by construction. For example, \( \text{len } [] = 0 \) evaluates to \text{true}.

3.6 Typing Rules

Next, we present the type-checking judgments and rules of \( \lambda^R \).

Environments and Closing Substitutions A type environment \( \Gamma \) is a sequence of type bindings \( x_1 : \tau_1, \ldots, x_n : \tau_n \). An environment denotes a set of closing substitutions \( \theta \) which are sequences of expression bindings: \( x_1 \mapsto e_1, \ldots, x_n \mapsto e_n \) such that:
\[
[\Gamma] \doteq [\theta \mid \forall x : \tau \in \Gamma. \theta(x) \in [\theta \cdot \tau] ]
\]

Judgments We use environments to define three kinds of rules:
Well-formedness, Subtyping, and Typing [14, 32]. A judgment \( \Gamma \vdash \tau \) states that the refinement type \( \tau \) is well-formed in the environment \( \Gamma \). Intuitively, the type \( \tau \) is well-formed if all the refinements in \( \tau \) are \text{Bool}-typed in \( \Gamma \). A judgment \( \Gamma \vdash \tau_1 \leq \tau_2 \) states that the type \( \tau_1 \) is a subtype of \( \tau_2 \) in the environment \( \Gamma \). Informally, \( \tau_1 \) is a subtype of \( \tau_2 \) if, when the free variables of \( \tau_1 \) and \( \tau_2 \) are bound to expressions described by \( \Gamma \), the denotation of \( \tau_1 \) is contained in the denotation of \( \tau_2 \). Subtyping of basic types reduces to denotational containment checking. That is, for any closing substitution \( \theta \) in the denotation of \( \Gamma \), for every expression \( e, \) if \( e \in [\theta \cdot \tau_1] \) then \( e \in [\theta \cdot \tau_2] \). A judgment \( \Gamma \vdash \tau \) states that the program \( \tau \) has the type \( \tau \) in the environment \( \Gamma \). That is, when the free variables in \( \tau \) are bound to expressions described by \( \Gamma \), the program \( \tau \) will evaluate to a value described by \( \tau \).

Rules All but three of the rules are standard [14, 32]. First, rule \( \text{T-REFLECT} \) is used to strengthen the type of each reflected binder with its definition, as described previously in §3.4. Second, rule \( \text{T-EXACT} \) strengthens the expression with a singleton type equating the value and the expression (i.e., reflecting the expression in the type). This is a generalization of the “selfification” rules from [14, 21], and is required to eutate the reflected functions with their definitions. For example, the application \( \text{fib}(1) \) is typed as \( \{ v : \text{Int} \mid \text{fib} p \vdash 1 \land v = \text{fib} 1 \} \) where the first conjunction comes from the (reflection-strengthened) output refinement

\[
\begin{align*}
\text{Well Formedness} & \\
\Gamma, v : B \vdash e : \text{Bool} & \quad \text{WF-BASE} \\
\Gamma \vdash \{ v : B \mid e \} & \quad \text{WF-FUN} \\
\end{align*}
\]

Subtyping
\[
\begin{align*}
\forall \theta \in [\Gamma]. [\theta \cdot \{ v : B \mid e \}] & \subseteq [\theta \cdot \{ v : B \mid e \}] & \text{\leq-BASE} \\
\Gamma \vdash \{ v : B \mid e \} & \leq \{ v : B \mid e \} & \text{\leq-FUN} \\
\end{align*}
\]

Typing
\[
\begin{align*}
\Gamma \vdash x : \tau & \quad \text{T-VAR} \\
\Gamma \vdash c : \text{Ty}(e) & \quad \text{T-CON} \\
\Gamma \vdash p : \tau & \quad \text{T-SUB} \\
\Gamma \vdash e : \{ v : B \mid e \} & \quad \text{T-EXACT} \\
\Gamma, x : \tau x \vdash e : \tau & \quad \text{T-FUN} \\
\Gamma \vdash e_1 : (x : \tau x \rightarrow \tau) & \quad \text{T-APP} \\
\Gamma \vdash e_2 : \tau x & \quad \text{T-LET} \\
\Gamma \vdash \text{let rec } x : \tau x = b : \tau & \quad \text{T-LET} \\
\Gamma \vdash \text{reflect } f : \text{Reflected}(\tau f, e) = e \in p : \tau & \quad \text{T-REFLECT} \\
\Gamma \vdash \text{fix } x : \tau \rightarrow y : \tau \rightarrow \tau & \quad \text{T-FIX}
\end{align*}
\]

Figure 3. Typing of \( \lambda^R \)
of \( \text{fib} \) § 2, and the second conjunct comes from rule T-EXACT. Finally, rule T-FIX is used to type the intermediate \text{fix} expressions that appear, not in the surface language but as intermediate terms in the operational semantics.

**Soundness** Following \( \lambda^R \) [32], we can show that evaluation preserves typing and that typing implies denotational inclusion.

**Theorem 1.** [Soundness of \( \lambda^R \)]

- **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 \( p \leadsto^* w \) then \( \emptyset \vdash w : \tau. \)

### 3.7 From Programs & Types to Propositions & Proofs

The denotational soundness Theorem 1 lets us interpret well typed programs as proofs of propositions.

**Definitions** A definition \( d \) is a sequence of reflected binders:

\[
d ::= \bullet \mid \text{reflect} \ x : \tau = e \text{ in } d\]

A definition’s environment \( \Gamma(d) \) comprises its binders and their reflected types:

\[
\Gamma(\bullet) \equiv \emptyset
\]

\[
\Gamma(\text{reflect } f : \tau = e \text{ in } d) \equiv (f, \text{Reflect}(\tau, e)), \Gamma(d)
\]

A definition’s substitution \( \theta(d) \) maps each binder to its definition:

\[
\theta(\bullet) \equiv \emptyset
\]

\[
\theta(\text{reflect } f : \tau = e \text{ in } d) \equiv ([f \mapsto \text{fix } e], \theta(d))
\]

**Propositions** A proposition is a type

\[
x_1 : \tau_1 \to \ldots \to x_n : \tau_n \to \{v : \text{Unit} \mid \text{prop} \}
\]

For brevity, we abbreviate propositions like the above to \( \overline{\tau} \rightarrow \{ \text{prop} \} \) and we call \text{prop} the proposition’s refinement. For simplicity we assume that \( \text{fix}(\tau) \equiv \emptyset. \)

**Validity** A proposition \( \overline{\tau} \rightarrow \{ \text{prop} \} \) is valid under \( d \) if

\[
\forall \overline{w} \in [\overline{\tau}], \theta(d) \cdot \text{prop}[\overline{\tau} \rightarrow \overline{w}] \leftrightarrow^* \text{True}
\]

That is, the proposition is valid if its refinement evaluates to True for every (well typed) interpretation for its parameters \( \overline{\tau} \) under \( d. \)

**Proofs** An expression \( e \) proves a proposition \( \tau \) under \( d \) if

\[
\emptyset \vdash d[\text{let rec } x : \tau = e \text{ in unit}] : \text{Unit}
\]

That is, if the expression \( e \) has the proposition’s type \( \tau \) under the definition \( d \)’s environment.

**Theorem 2.** [Proofs] If \( e \) proves \( \tau \) under \( d \) then \( \tau \) is valid under \( d. \)

**Proof.** As \( e \) proves \( \tau \) under \( d, \) we have

\[
\emptyset \vdash d[\text{let rec } x : \tau = e \text{ in unit}] : \text{Unit}
\]

By Theorem 1 on 1 we get

\[
\theta(d) \in [\Gamma(d)]
\]

Furthermore, by the typing rules 1 implies \( \Gamma(d) \vdash e : \tau \) and hence, via Theorem 1

\[
\forall \theta \in [\Gamma(d)]. \theta \cdot e \in [\theta \cdot \tau]
\]

Together, 2 and 3 imply

\[
\theta(d) \cdot e \in [\theta(d) \cdot \tau]
\]

By the definition of type denotations, we have

\[
[\theta(d) \cdot \tau] \equiv \{f \mid \tau \text{ is valid under } d\}
\]

By 4, the above set is not empty, and hence \( \tau \) is valid under \( d. \)

### 4. Algorithmic Verification

Next, we describe \( \lambda^S, \) a conservative approximation of \( \lambda^R \) where the undecidable type subsumption rule is replaced with a decidable one, yielding an SMT-based algorithmic type system that enjoys the same soundness guarantees.

#### 4.1 The SMT logic \( \lambda^S \)

**Syntax: Terms & Sorts** Figure 4 summarizes the syntax of \( \lambda^S, \) the sorted (SMT-) decidable logic of quantifier-free equality, uninterpreted functions and linear arithmetic (QF-EUALG) [4, 19]. The terms of \( \lambda^S \) 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 \( x \overline{\tau} \) of an uninterpreted function \( x. \) The sorts of \( \lambda^S \) include built-in integer \( \text{Int} \) and \( \text{Bool} \) for representing integers and booleans. The interpreted functions of \( \lambda^S, \) e.g. the logical constants \( = \) and \( \land, \) have the function sort \( \text{Bool} \) and \( \text{Int} \) sorts as:

\[
\begin{align*}
\text{Predicates} & : r ::= r \oplus_2 r \mid \oplus_1 r \\
& \mid n \mid b \mid x \mid D \mid x \overline{r} \\
& \mid \text{if } r \text{ then } r \text{ else } r \\
\text{Integers} & : n ::= 0, -1, 1, \ldots \\
\text{Booleans} & : b ::= \text{True} \mid \text{False} \\
\text{Bin Operators} & : \oplus_2 ::= = \mid < \mid \land \mid + \mid - \mid \ldots \\
\text{Un Operators} & : \oplus_1 ::= \land \mid \ldots \\
\text{Model} & : \sigma ::= \sigma'(x : r) \mid \emptyset \\
\text{Sorts} & : s ::= \text{Int} \mid \text{Bool} \mid \text{U} \\
& \mid s \to s \mid \text{Fun } s s
\end{align*}
\]

**Example: Fibonacci is increasing** In § 2 we verified that under a definition \( d \) that includes \( \text{fib} \), the term \( \text{fib}(n+1) \) proves

\[
n : \text{Nat} \to \{ \text{fib } n \leq \text{fib } (n+1) \}
\]

Thus, by Theorem 2 we get

\[
\forall n.0 \leq n \leftrightarrow^* \text{True} \Rightarrow \text{fib } n \leq \text{fib } (n+1) \leftrightarrow^* \text{True}
\]
Then the λ translates case-expressions of non-boolean data constructor such that:

\[
\begin{align*}
\Gamma \vdash e \rightsquigarrow r \\
\Gamma \vdash e_1 \rightsquigarrow r_1 \\
\Gamma \vdash e_2 \rightsquigarrow r_2 \\
\Gamma \vdash e \rightsquigarrow r \\
\Gamma \vdash e \rightsquigarrow r \\
\Gamma \vdash e \rightsquigarrow r \\
\Gamma \vdash e \rightsquigarrow r \\
\Gamma \vdash e \rightsquigarrow r \\
\end{align*}
\]

Embedding Functions As λ\(^2\) is a first-order logic, we embed λ-abstraction and application using the uninterpreted functions \(\lambda\text{am}\) and \(\text{app}\). We embed λ-abstractions using \(\lambda\text{am}\) as shown in rule T-FUN. The term \(\lambda x.e\) of type \(\tau_0 \rightarrow \tau\) is transformed to \(\lambda x.\text{am}_x r\) of sort \(\text{Fun}\ \tau_0 \rightarrow \tau\), where \(\text{am}_x\) and \(\tau_0\) are respectively \(\{\tau_i\}\) and \(\{\tau_i\}\), and \(\lambda x.\text{am}_x r\) is a special uninterpreted function of sort \(\text{Fun} \ \tau_0 \rightarrow \tau\), and \(x\) of sort \(\tau_0\) and \(r\) are the embedding of the binder and body, respectively. As \(\lambda\text{am}\) is just an SMT-function, it does not create a binding for \(x\). Instead, the binder \(x\) is renamed to a fresh name pre-declared in the SMT environment.

Embedding Applications Dually, we embed applications via decomposition [23] using an uninterpreted \(\text{apply}\) function \(\text{app}\) as shown in rule T-APP. The term \(e\ e'\), where \(e\) and \(e'\) have types \(\tau_0 \rightarrow \tau\) and \(\tau_0\), is transformed to \(\text{app}_x r r'\) : \(s\) where \(s\) and \(s\) are respectively \(\{\tau_i\}\) and \(\{\tau_i\}\), and \(\text{app}_x r r'\) is a special uninterpreted function of sort \(\text{Fun}\ \tau_0 \rightarrow \tau\), and \(r\) and \(r'\) are the respective translations of \(e\) and \(e'\).

Embedding Data Types Rule T-DC translates each data constructor to a predefined λ\(^2\) constant \(\text{sd}\) of sort \(\{\text{Ty}(D)\}\). Let \(D\) be a non-boolean data constructor such that:

\[
\text{Ty}(D) \doteq \tau_1, \ldots, \tau_n \rightarrow \tau
\]

Then the check function \(\text{is}_{D}\) has the sort \(\{\tau\}\ \text{Bool}\), and the select function \(\text{sel}_{D, i}\) has the sort \(\text{Fun}\ \{\tau\}\ \{\tau_i\}\). Rule T-CASE translates case-expressions of λ\(^3\) into nested if terms in λ\(^4\), by using the check functions in the guards, and the select functions for the binders of each case. For example, following the above, the body of the list append function

\[
\begin{align*}
\text{Figure 5. Transforming } \lambda^{\beta} \text{ terms into } \lambda^{\delta}. \\
\text{all applications to primitive constants and data constructors are saturated, i.e. fully applied, e.g. by converting source level terms like } (+~1) \text{ to } (\lambda\ z \rightarrow z + 1).
\end{align*}
\]

Well Formedness

\[
\begin{align*}
\Gamma, v : \text{Bool} \vdash e : \text{Bool} \quad \frac{\Gamma \vdash e \rightsquigarrow r}{\Gamma \vdash \text{ref} \vdash \Gamma \vdash e \rightsquigarrow r}. \\
\Gamma \vdash \text{ref} \vdash \Gamma \vdash \text{ref} \vdash \Gamma \vdash \text{ref} \vdash \Gamma \vdash \text{ref}
\end{align*}
\]

Subtyping

\[
\begin{align*}
\Gamma' \vdash \Gamma, v : \{B | e\} \quad \frac{\Gamma' \vdash e' \rightsquigarrow r'}{\text{Valid}(\{\Gamma'\} \Rightarrow r')} \\
\Gamma \vdash \text{ref} \vdash \Gamma \vdash \text{ref} \vdash \Gamma \vdash \text{ref} \vdash \Gamma \vdash \text{ref}
\end{align*}
\]

Figure 6. Algorithmic Typing (other rules in Figs 1 and 3.)

[ ] ++ \(\text{ys} = \text{ys}\) 
\((x : x \text{ss}) \quad ++ \text{ys} = x : (x \text{ss} ++ \text{ys})\) is reflected into the \(\lambda^S\) refinement:

\[
\begin{align*}
\text{if} \quad \text{isNil} \ x s \quad \text{then} \quad \text{ys} \quad \text{else} \quad \text{sel} \ x s \quad : \ \{\text{sel} \ x s \quad ++ \ \text{ys}\}
\end{align*}
\]

We favor selectors to the axiomatic translation of HALO [35] and F\(^\star\) [30] to avoid universally quantified formulas and the resulting instantiation unpredictability.

4.3 Correctness of Translation

Informally, the translation relation \(\Gamma \vdash e \rightsquigarrow r\) is correct in the sense that if \(e\) is a terminating boolean expression then \(e\) reduces to True \(r\) \(r\) is SMT-satisfiable by a model that respects β-equivalence.

Definition 1 (β-Model). A \(\beta\)-model \(\sigma\) is an extension of a model \(\sigma\) where \(\lambda\text{am}\) and \(\text{app}\) satisfy the axioms of \(\beta\)-equivalence:

\[
\begin{align*}
\forall x y. \lambda x. e \vdash r & \iff \lambda y. (e[x \rightarrow y]) \\
\forall x e. e. (\text{app} (\lambda x. e) e) & \equiv e[x \rightarrow e]
\end{align*}
\]

Semantics Preservation We define the translation of a \(\lambda^\beta\) term into \(\lambda^S\) under the empty environment as \(\{\![e]\!\} \vdash \Gamma\) \(\Gamma\) if \(\Gamma \vdash e \rightsquigarrow r\). A lifted substitution \(\theta^\beta\) is a set of models \(\theta\) where each “bottom” in the substitution \(\theta\) is mapped to an arbitrary logical value of the respective sort [32]. We connect the semantics of \(\lambda^\beta\) and translated \(\lambda^S\) via the following theorems:

Theorem 3. If \(\Gamma \vdash e \rightsquigarrow r\), then for every \(\theta \in \Gamma\) and every \(\sigma \in \theta^\beta\), if \(\theta^\beta \cdot e \rightsquigarrow r^\beta\) then \(\theta^\beta \cdot e \rightsquigarrow r\).

Corollary 1. If \(\Gamma \vdash e : \text{Bool}\), \(e\) reduces to a value and \(\Gamma \vdash e \rightsquigarrow r\), then for every \(\theta \in \Gamma\) and every \(\sigma \in \theta^\beta\), \(\theta^\beta \cdot e \rightsquigarrow r\). True \(i\) \(r\) \(r\) is valid only if the set of values described by \(\Gamma\), is subsumed by the set of values described by \(r\). \(\Gamma\) is embedded into logic
by conjoining (the embeddings of) the refinements of provably terminating binders [32]:

\[
\{ \Gamma \} \vdash \bigwedge_{x \in \Gamma} \{ \Gamma, x \} \quad \text{where we embed each binder as}
\]

\[
\{ \Gamma, x \} \equiv \begin{cases} r & \text{if } \Gamma(x) = \{ v : B^\alpha \mid e \}, \Gamma \vdash e[v \mapsto x] \leadsto r \\ \text{True} & \text{otherwise} \end{cases}
\]

### Subtyping via SMT Validity

We make subtyping, and hence, typing decidable, by replacing the denotational base subtyping rule \( \leq_{\text{BASE}} \) with a conservative, algorithmic version that uses an SMT solver to check the validity of the subtyping VC. We use Corollary 1 to prove soundness of subtyping.

**Lemma 2.** If \( \Gamma \vdash S \{ v : B \mid e_1 \} \leq \{ v : B \mid e_2 \} \) then \( \Gamma \vdash \{ v : B \mid e_1 \} \leq \{ v : B \mid e_2 \} \).

**Soundness of \( \lambda^\beta \).** Lemma 2 directly implies the soundness of \( \lambda^\beta \).

**Theorem 4** (Soundness of \( \lambda^\beta \)). If \( \Gamma \vdash e : \tau \) then \( \Gamma \vdash e : \tau \).

## 5. Reasoning About Lambdas

Though \( \lambda^\beta \), as presented so far, is sound and decidable, it is *imprecise*: our encoding of \( \lambda \)-abstractions and applications via uninterpreted functions makes it impossible to prove theorems that require \( \alpha \)- and \( \beta \)-equivalence, or extensional equality. Next, we show how to address the former by strengthening the VCs with equalities §5.1, and the latter by introducing a binder for safely asserting extensional equality §5.2. In the rest of this section, for clarity we omit \( \text{app} \) when it is clear from the context.

### 5.1 Equivalence

As soundness relies on satisfiability under a \( \sigma^\beta \) (see Definition 1), we can safely instantiate the axioms of \( \alpha \)- and \( \beta \)-equivalence on any set of terms of our choosing and still preserve soundness (Theorem 4). That is, instead of checking the validity of a VC \( p \Rightarrow q \), we check the validity of a strengthened VC, \( a \Rightarrow p \Rightarrow q \), where \( a \) is a (finite) conjunction of equivalence instances derived from \( p \) and \( q \), as discussed below.

**Representation Invariant** The lambda binders, for each SMT sort, are drawn from a pool of names \( x_i \) where the index \( i = 1, 2, \ldots \). When representing \( \lambda \) terms we enforce a normalization invariant that for each lambda term \( \lambda x_i \ e \), the index \( i \) is greater than any lambda argument appearing in \( e \).

**\( \alpha \)-instances** For each syntactic term \( \lambda x_i \ e \), and \( \lambda \)-binders \( x_j \) such that \( i < j \) appearing in the VC, we generate an \( \alpha \)-equivalence instance predicate (or \( \alpha \)-instance):

\[
\lambda x_i \ e = \lambda x_j \ e[x_i \mapsto x_j]
\]

The conjunction of \( \alpha \)-instances can be more precise than De Bruijn representation, as they let the SMT solver deduce more equalities via congruence. For example, consider the VC needed to prove the applicative laws for \( \text{Reader} \):

\[
d \equiv \lambda x_1 \ (x_1)
\]

\[
\Rightarrow \lambda x_2 \ (\lambda x_1 \ (x_1) \ x_2) \equiv \lambda x_1 \ (d \ x_1)
\]

The \( \alpha \) instance \( \lambda x_1 \ (d \ x_1) \equiv \lambda x_2 \ (d \ x_2) \) derived from the VC’s hypothesis, combined with congruence immediately yields the VC’s consequence.

**\( \beta \)-instances** For each syntactic term \( \text{app} (\lambda x \ e) \ e \), with \( e \) not containing any \( \lambda \)-abstractions, appearing in the VC, we generate an \( \beta \)-equivalence instance predicate (or \( \beta \)-instance):

\[
\text{app} (\lambda x \ e) \ e' \equiv e[x_i \mapsto e_x], \text{s.t. } e' \text{ is } \lambda \text{-free}
\]

We require the \( \lambda \)-free restriction as a simple way to enforce that the reduced term \( e[x_i \mapsto e'] \) enjoys the representation invariant.

For example, consider the following VC needed to prove that the bind operator for lists satisfies the monadic associativity law.

\[
(f x \gg= g) = \text{app} (\lambda y \ (f y \gg= g) \ x)
\]

The right-hand side of the above VC generates a \( \beta \)-instance that corresponds directly to the equality, allowing the SMT solver to prove the (strengthened) VC.

**Normalization** The combination of \( \alpha \)- and \( \beta \)-instances is often required to discharge proof obligations. For example, when proving that the bind operator for the \( \text{Reader} \) monad is associative, we need to prove the VC:

\[
\lambda x_2 (\lambda x_1 w) = \lambda x_3 (\text{app} (\lambda x_2 (\lambda x_1 w)) w)
\]

The SMT solver proves the VC via the equalities corresponding to an \( \alpha \) and then \( \beta \)-instance:

\[
\lambda x_2 (\lambda x_1 w) = \lambda x_3 (\text{app} (\lambda x_2 (\lambda x_1 w)) w)
\]

### 5.2 Extensionality

Often, we need to prove that two functions are equal, given the definitions of reflected binders. For example, consider

\[
\text{reflect id}
\]

\[
\text{id} x = x
\]

Liquid Haskell accepts the proof that \( \text{id} x = x \) for all \( x \):

\[
\begin{align*}
\text{id}_x \text{_eq}_x & : x : a \to \{ \text{id} x = x \\
\text{id}_x \text{_eq}_x & = \{ \forall x : \text{id} x = x \}
\end{align*}
\]

\[
\text{fails} :: \text{id}_x \text{_eq}_x
\]

\[
\text{fails} = \{ \forall x : \text{id} x \neq x \}
\]

\[
\{ \text{id}_x \text{_eq}_x \}
\]

as “calling” \( \text{id} \) unfolds its definition, completing the proof. However, consider this \( \eta \)-expanded variant of the above proposition:

\[
\text{type} \ \text{id}_x \text{_eq}_x = \{ \{ \forall x : \text{id} x \} = (\forall y : \text{id} y) \}
\]

Liquid Haskell rejects the proof:

\[
\text{fails} :: \text{id}_x \text{_eq}_x
\]

\[
\text{fails} = \{ \forall x : \text{id} x \neq \text{id} y \}
\]

\[
\{ \text{id}_x \text{_eq}_x \}
\]

The invocation of \( \text{id} \) unfolds the definition, but the resulting equality refinement \( \{ \text{id} x = x \} \) is *trapped* under the \( \lambda \)-abstraction. That is, the equality is absent from the typing environment at the top level, where the left-hand side term is compared to \( \forall y : \text{id} y \). Note that the above equality requires the definition of \( \text{id} \) and hence is outside the scope of purely the \( \alpha \)- and \( \beta \)-instances.

**An Extensionality Operator** To allow function equality via extensionality, we provide the user with a (family of) function comparison operator(s) that transform an explanation \( p \) which is a proof that \( f x = g x \) for every argument \( x \) into a proof that \( f = g \).

\[
=\forall \ : f : (a \to b) \to g : (a \to b)
\]

\[
\to \text{exp} : (x : a \to f x = g x) \to (f = g)
\]

Of course, \( =\forall \) cannot be implemented; its type is *assumed*. We can use \( =\forall \) to prove \( \text{id}_x \text{_eq}_x \) by providing a suitable explanation:

\[
\text{pf}_x : \text{id}_x \text{_eq}_x
\]

\[
\text{pf}_x = \{ \forall y : \text{id} y \} \ =\forall \ (\forall x : \text{id} x) \ \text{'} \ : \text{exp} \quad \text{QED}
\]

where

\[
\text{exp} = (\forall x : \text{id} x \ =. \ =. \ \text{** QED})
\]

The explanation is the second argument to \( =\forall \) which has the following type that syntactically fires \( \beta \)-instances:

\[
x : a \to (\forall x : \text{id} x) x = (\forall x : \text{id} x) x
\]
seamlessly connected via refinement typing:

Our proofs use three building blocks, that are

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

from the literature and categorized in Figure 7, totalling about 2500

verify a variety of deep specifications of Haskell functions drawn

We have implemented refinement reflection in Liquid Haskell. In

6. 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 2500

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

that are

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,

Proof Strategies.

All

lines of specifications and proofs. Next, we detail each of the four

categories of specifications and proofs. There are three building blocks,
suitable

Haskell to prove that

Peano

where

assoc [] ys zs = ([]) ++ ys ++ zs

** QED

assoc [] ys zs = ([]) ++ ys ++ zs

** QED

assoc (x:xs) ys zs = (x:(xs ++ ys)) ++ zs

=. x:((xs ++ ys) ++ zs)

=. x:((xs ++ (ys ++ zs))

∴ assoc xs ys zs

=. (xs:ys) ++ (ys ++ zs)

∴ QED

Figure 9. TypeClass Laws verified using Liquid Haskell

→ fuse: (h (f x y) = g x (h y)))
→ ((h . foldr f e) z = foldr g (h e) z)

foldr_fusion h f g =
= foldr_univ g (h . foldr f e) (h e) ys

(fuse_base h f e)

(fuse_step h f e g fuse)

where fuse_base and fuse_step prove the base and inductive
cases, and for example fuse_base is a function with type

fuse_base :: h:(b -> c) -> f:(a -> b) -> e:b -> (h . foldr f e) [] = h e

6.3 TypeClass Laws

We used Liquid Haskell to prove the Monoid, Functor, Applicative and Monad Laws, summarized in Figure 9, for various user-defined instances summarized in Figure 7.

Monoid Laws A Monoid is a datatype equipped with an associative binary operator ◊ and an identity element mempty. We use Liquid Haskell to prove that Peano (with add and 2), Maybe (with a suitable `mappend` and Nothing), and List (with `append` and []) satisfy the monoid laws. For example, we prove that `[]` is associative by reifying the textbook proof [12] into a Haskell function, where the induction corresponds to case-splitting and recurring on the first argument:

```haskell
assoc :: xs:[a] -> ys:[a] -> zs:[a] -> ((xs ++ ys) ++ zs = xs ++ (ys ++ zs))

assoc [] ys zs = ([]) ++ ys ++ zs

** QED

assoc (x:xs) ys zs = (x:(xs ++ ys)) ++ zs

=. x:((xs ++ ys) ++ zs)

=. x:(xs ++ (ys ++ zs))

∴ assoc xs ys zs

=. (xs:ys) ++ (ys ++ zs)

** QED
```

Functor Laws A type is a functor if it has a function `fmap` that satisfies the `identity` and `distribution` (or fusion) laws in Figure 9. For example, consider the proof of the `fmap` distribution law for the lists, also known as "map-fusion", which is the basis for important optimizations in GHC [36]. We reflect the definition of `fmap`:

```haskell
reflect map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : fmap f xs
```

and then specify fusion and verify it by an inductive proof:

```haskell
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 9), refer to λ-functions, thus their proof exercises our support for defunctionalization and η- and β-equivalence. For example, consider the proof of the associativity law for the list monad. First, we reflect the bind operator:

```haskell
reflect (>>=) :: [a] -> (a -> [b]) -> [b]
(x:xs) >>= f = f x ++ (xs >>= f)
([]) >>= f = []
```

Next, we define an abbreviation for the associativity property:

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

Finally, we can prove that the list-bind is associative:

```haskell
assoc :: m:[a] -> f:(a -> [b]) -> g:(b -> [c])
→ AssocLaw m f g
assoc [] f g
= [ [] >>= f >>= g ]
= [ [] >>= g ]
= [ [] >>= (λx → f x >>= g) ]

** QED
assoc (x:xs) f g
= (x:xs) >>= f >>= g
=(f x x ++ xs >>= f) >>= g
= (f x >>= g) ++ (xs >>= (λ y → f y >>= g))

∴ assoc xs f g
= (λy → f y >>= g) x ++
(xs >>= (λ y → f y >>= g))

∴ βeq f g x
= (λy → f y >>= g) x

** QED
```

Where the bind-append fusion lemma states that:

```haskell
bind_append ::
xs:[a] → ys:[a] → zs:[a] → (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:

```haskell
βeq :: f:_ → g:_-> x:_ →

= (bind (f x) g = (λ y → bind (f y) g) x)

βeq _ _ _ = trivial
```

6.4 Functional Correctness

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

SAT Solver We implemented and verified the simple SAT solver used to illustrate and evaluate the features of the dependently typed language Zombie [7]. The solver takes as input a formula \( \varepsilon \) and returns an assignment that satisfies \( \varepsilon \) if one exists.
solve :: f:Formula → Maybe {a:Asgn|sat a f}
solve f = find ('sat' f) (assignments f)

Function assignments f returns all possible assignments of the formula f and sat a f returns True iff the assignment a satisfies the formula f:

reflect sat :: Asgn → Formula → Bool
assignments :: Formula → [Asgn]

Verification of solve follows simply by reflecting sat into the refinement logic, and using (bounded) refinements to show that find only returns values on which its input predicate yields True [33].

find :: p:(a → Bool) → [a]
→ Maybe {via | p v}

Unification As another example, we verified the unification of first order terms, as presented in [27]. 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 can define 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 the 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.

As before, we prove correctness by invoking separate helper lemmas. 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)

7. Related Work

SMT-Based Verification SMT-solvers have been extensively used to automate program verification via Floyd-Hoare logics [19]. Our work is inspired by Dafny’s Verified Calculations [17], a framework for proving theorems in Dafny [15], 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* [30] which like Liquid Haskell, uses types to compose proofs, offer more automation by translating recursive functions to SMT axioms. However, unlike reflectiom this axiomatic approach renders typechecking / verification undecidable (in theory) and leads to unpredictability and divergence (in practice) [16].

Dependent types Our work is inspired by dependent type systems like Coq [6] and Agda [20]. 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 for Non-Terminating Programs Zombie [7, 27] integrates dependent types in non terminating programs and supports automatic reasoning for equality. Vazou et al. have previously [32] shown how Liquid Types can be used to check non-terminating programs. Reflection makes Liquid Haskell at least as expressive as Zombie, without having to axiomatize the theory of equality within the type system. Consequently, in contrast to Zombie, SMT based reflection lets Liquid Haskell verify higher-order specifications like foldr_fusion.

Dependent Types in Haskell Integration of dependent types into Haskell has been a long standing goal that dates back to Cayenne [3], a Haskell-like, fully dependent type language with undecidable type checking. In a recent line of work [10] Eisenberg et al. 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) our proofs cannot degrade (resp. optimize) the performance of programs.

Proving Equational Properties Several authors have proposed tools for proving (equational) properties of (functional) programs. Systems [29] and [2] 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 [28] generates proofs by term rewriting and Halo [35] 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 [11] proves equalities by rewriting GHC core 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.

8. Conclusions & Future Directions

We have shown how refinement reflection – namely reflecting the definitions of functions in their output refinements – can be used to convert a language into a proof assistant, while ensuring (refinement) type checking stays decisive and predictable via careful design of the logic and proof combinators.

Our evaluation shows that refinement reflection lets us prove deep specifications of a variety of implementations, and identifies important avenues for research. First, while proofs are possible, they can sometimes be cumbersome. For example, in the proof of associativity of the monadic bind operator for the Reader monad three of eight (extensional) equalities required explanations, some nested under multiple lambda-abstractions. Thus, it would be valuable to use recent advances in refinement-based synthesis [22] to automate proof construction. Second, while our approach to alpha- and beta-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 SMT axiomatization, thereby automating proofs predictably.
References

[34] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript. In PLDI, 2016.