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 ≤ v`, to get a `Nat` type:

```
type Nat = {v:Int | 0 ≤ 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:

```
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`. 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:

```
type Prop = ()
```

but which can be *refined* with desired propositions about the code. For example, the following states the proposition $2 + 2$ 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}
```

We represent universally quantified propositions as function types:

```
type Plus_com = x:Int→y:Int→{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`.

```
data QED = QED

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

To resemble mathematical proofs, we make this casting post-fix. Thus, we can write `e ** QED` to cast `e` to a value of `Prop`. For example, we can prove the above propositions simply by writing

```
pf_plus_2_2 :: Plus_2_2_eq_4
pf_plus_2_2 = trivial ** QED

pf_plus_comm :: Plus_comm
pf_plus_comm = \x y → trivial ** QED

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`.

```
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 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:

```
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*

$$\forall n, m.\, n = m \;\Rightarrow\; \texttt{fib}\, n = \texttt{fib}\, m$$

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:

```
predicate 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* during refinement type checking. We prove `fib2_eq_1` by:

```
pf_fib2 :: { fib 2 = 1 }
pf_fib2 = fib 2 == fib 1 + fib 0 ** QED
```

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

```
let { t0 = fib 0; t1 = fib 1; t2 = fib 2 }
in  ( t2 == t1 + t0 ) ** QED
```

Which via standard refinement typing, yields the following verification condition that is easily 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 can automatically *combine* the facts, once they are in the antecedent. Hence, the following would also be 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, F* and HALO, is to encode the definition of `fib` as a universally quantified SMT formula (or axiom):

$$\forall n.\ \texttt{fibP}\,(\texttt{fib}\ n)\,n$$

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

```
axPf_fib2 :: { fib 2 = 1 }
axPf_fib2 = trivial ** QED
```

However, the presence of such axioms renders checking 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 [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 a family of equation combinators $\odot$`.` for each logical operator $\odot$ in $\{=, \neq, \leq, <, \geq, >\}$, the operators in the theory QF-UFLIA. The refinement type of $\odot$`.` *requires* that $x \odot y$ holds and then *ensures* that the returned value is equal to `x`. For example, we define `=.` as:

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

and use it to write the following "equational" proof:

```
eqPf_fib2 :: { fib 2 = 1 }
eqPf_fib2 =  fib 2
          =. fib 1 + fib 0
          =. 1
          ** QED
```

"Because" Combinators Often, we need to compose "lemmas" into larger theorems. For example, to prove `fib 3 = 2` we may wish to reuse `eqPf_fib2` as a lemma. To this end, Liquid Haskell has a "because" combinator:

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

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

```
eqPf_fib3 :: {fib 3 = 2}
eqPf_fib3 =  fib 3
          =. fib 2 + fib 1
          =. 2                    ∵ eqPf_fib2
          ** QED
```

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 `fib` is (locally) increasing, *i.e.* for all n, `fib` $n \leq$ `fib` $(n+1)$

```
fibUp :: n:Nat → {fib n ≤ fib (n+1)}
fibUp n
  | n == 0
  =  fib 0 <. fib 1
  ** QED

  | n == 1
  =  fib 1 ≤. fib 1 + fib 0 ≤. fib 2
  ** QED

  | otherwise
  =  fib n
  =. fib (n-1) + fib (n-2)
  ≤. fib n     + fib (n-2) ∵ fibUp (n-1)
  ≤. fib n     + fib (n-1) ∵ fibUp (n-2)
  ≤. fib (n+1)
  ** QED
```

Case Splitting and Induction The proof `fibUp` works by induction on `n`. In the *base* cases `0` and `1`, we simply assert the relevant inequalities. These are verified as the reflected refinement unfolds the definition of `fib` at those inputs. The derived VCs are (automatically) proved as the SMT solver concludes $0 < 1$ and $1 + 0 \leq 1$ respectively. In the *inductive* case, `fib n` is unfolded to `fib (n-1) + fib (n-2)`, which, because of the induction hypothesis (applied by invoking `fibUp` at `n-1` and `n-2`), and the SMT 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 ≤ 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
  =  f x ≤. f (x+1) ∵ fUp x
         ≤. f y
         ** QED

  | x + 1 < y
  =  f x ≤. f (y-1) ∵ fMono f fUp x (y-1)
         ≤. f y     ∵ fUp (y-1)
         ** 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 `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
```

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:

```
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 § 3.5 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 [32], 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 =  add Z n
        =. n
        ** QED
```

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     =  add Z Z
            =. Z
            ** QED

zeroR (S n) =  add (S n) Z
            =. S (add n Z)
            =. S n          ∵     zeroR n
            ** QED
```

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 (§ 3).

Commutativity Lets conclude by proving that `add` is commutative:

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

add_com    Z  b =  add Z b
                =. b
                =. add b Z      ∵ zeroR b
                ** QED

add_com (S a) b =  add (S a) b
                =. S (add a b)
                =. S (add b a) ∵ add_com a b
                =. add b (S a) ∵ sucR b a
                ** QED
```

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.

3. Refinement Reflection

Our first step towards formalizing refinement reflection is 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.

3.1 Syntax

Figure 1 summarizes the syntax of λ^R, which is essentially the calculus λ^U [32] with explicit recursion and a special **`reflect`** binding form to denote terms that are reflected into the refinement logic. In λ^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 § 4 we will see how to recover decidability by soundly approximating refinements.

The syntactic elements of λ^R are layered into primitive constants, values, expressions, binders and programs.

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

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

$$
\begin{array}{rrcl}
\textbf{\textit{Operators}} & \odot & ::= & = \mid < \\
\textbf{\textit{Constants}} & c & ::= & \wedge \mid \neg \mid \odot \mid +, -, \dots \\
& & \mid & \texttt{True} \mid \texttt{False} \mid 0, 1, -1, \dots \\
\textbf{\textit{Values}} & w & ::= & c \mid \lambda x.e \mid D\,\overline{w} \\
\textbf{\textit{Expressions}} & e & ::= & w \mid x \mid e\;e \\
& & \mid & \texttt{case}\; x = e\; \texttt{of}\; \{D\,\overline{x} \to e\} \\
\textbf{\textit{Binders}} & b & ::= & e \mid \texttt{let rec}\; x : \tau = b\; \texttt{in}\; b \\
\textbf{\textit{Program}} & p & ::= & b \mid \texttt{reflect}\; x : \tau = e\; \texttt{in}\; p \\
\textbf{\textit{Basic Types}} & B & ::= & \texttt{Int} \mid \texttt{Bool} \mid T \\
\textbf{\textit{Refined Types}} & \tau & ::= & \{v : B \mid e\} \mid x : \tau \to \tau
\end{array}
$$

Figure 1. Syntax of λ^R

Contexts $\boxed{C}$

$$
\begin{array}{rcl}
C & ::= & \bullet \\
& \mid & C\;e \mid c\;C \mid D\;\overline{e}\;C\;\overline{e} \\
& \mid & \texttt{case}\; y = C\; \texttt{of}\; \{D_i\;\overline{x} \to e_i\}
\end{array}
$$

Reductions $\boxed{p \hookrightarrow p'}$

$$
\begin{array}{rcl}
C[p] & \hookrightarrow & C[p'], \quad \text{if } p \hookrightarrow p' \\
c\;v & \hookrightarrow & \delta(c, v) \\
(\lambda x.e)\;e' & \hookrightarrow & e[x \mapsto e'] \\
\texttt{case}\; y = D_j\;\overline{e}\;\texttt{of}\;\{D_i\;\overline{x_i} \to e_i\} & \hookrightarrow & e_j[y \mapsto D_j\;\overline{e}][\overline{x_i} \mapsto \overline{e}] \\
\texttt{reflect}\; x : \tau = e\; \texttt{in}\; p & \hookrightarrow & p[x \mapsto \texttt{fix}\;(\lambda x.e)] \\
\texttt{let rec}\; x : \tau = b_x\; \texttt{in}\; b & \hookrightarrow & b[x \mapsto \texttt{fix}\;(\lambda x.b_x)] \\
\texttt{fix}\;p & \hookrightarrow & p\;(\texttt{fix}\;p)
\end{array}
$$

Figure 2. Operational Semantics of λ^R

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 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 refinement 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 β-reduction semantics for λ^R. We write $e \hookrightarrow^j e'$ if there exist $e_1, \dots, e_j$ such that e is e_1, e' is e_j and $\forall i, j, 1 \leq i < j$, we have $e_i \hookrightarrow e_{i+1}$. We write $e \hookrightarrow^\star e'$ if there exists some finite j such that $e \hookrightarrow^j e'$. We define $\approx_\beta$ to be the reflexive, symmetric, 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. 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. 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 $(f = f') \hookrightarrow \texttt{True}$ iff $\forall v.\; f\;v \approx_\beta f'\;v$. 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

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* $\{v : B \mid e\}$. Finally, we have dependent *function types* $x : \tau_x \to \tau$ where the input x has the type τ_x and the output τ may refer to the input binder x. We write B to abbreviate $\{v : B \mid \texttt{True}\}$, and $\tau_x \to \tau$ to abbreviate $x : \tau_x \to \tau$ if x does not appear in τ. We use r to refer to refinements.

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

$$
\begin{array}{rcl}
[\![\{x : B \mid r\}]\!] & \doteq & \{e \mid e : B,\; \text{if } e \hookrightarrow^\star w \text{ then } r[x \mapsto w] \hookrightarrow^\star \texttt{True}\} \\
[\![x : \tau_x \to \tau]\!] & \doteq & \{e \mid e : \lfloor\tau_x \to \tau\rfloor, \forall e_x \in [\![\tau_x]\!].\; e\;e_x \in [\![\tau[x \mapsto e_x]]\!]\}
\end{array}
$$

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

$$
\begin{array}{rcl}
\mathsf{Ty}(3) & \doteq & \{v : \texttt{Int} \mid v = 3\} \\
\mathsf{Ty}(+) & \doteq & \texttt{x} : \texttt{Int} \to \texttt{y} : \texttt{Int} \to \{v : \texttt{Int} \mid v = x + y\} \\
\mathsf{Ty}(\leq) & \doteq & \texttt{x} : \texttt{Int} \to \texttt{y} : \texttt{Int} \to \{v : \texttt{Bool} \mid v \Leftrightarrow x \leq y\}
\end{array}
$$

So, by definition we get the constant typing lemma

Lemma 1. *[Constant Typing] Every constant $c \in [\![\mathsf{Ty}(c)]\!]$.*

Thus, if $\mathsf{Ty}(c) \doteq x : \tau_x \to \tau$, then for every value $w \in [\![\tau_x]\!]$, we require $\delta(c, w) \in [\![\tau[x \mapsto w]]\!]$.

3.4 Refinement Reflection

The simple, but 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: $\texttt{reflect}\; f : \tau = e\; \texttt{in}\; p$ as a `let rec`-binder: $\texttt{let rec}\; f : \mathsf{Reflect}(\tau, e) = e\; \texttt{in}\; p$ during type checking (rule T-REFLECT in Figure 3).

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

$$
\begin{array}{rcl}
\mathsf{Reflect}(\{v : B \mid r\}, e) & \doteq & \{v : B \mid r \wedge v = e\} \\
\mathsf{Reflect}(x : \tau_x \to \tau, \lambda y.e) & \doteq & x : \tau_x \to \mathsf{Reflect}(\tau, e[y \mapsto x])
\end{array}
$$

As an example, recall from § 2 that the **`reflect`** `fib` strengthens the type of `fib` with the reflected refinement `fibP`.

Consequences for Verification Reflection 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 "triggers" 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]:

$$
\texttt{measure}\;\texttt{f} : \tau = \lambda x.\texttt{case}\; y = x\; \texttt{of}\; \{D_i\;\overline{z} \to e_i\}
$$

For example, `len` measures the size of an $[Int]$:

```
measure len :: [Int] → Nat
len = \x → case x of
               []     → 0
               (x:xs) → 1 + len 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:

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

isCons (x:xs)  = True
isCons []      = False

sel1 (x:xs)    = x
sel2 (x:xs)    = xs
```

Refining Data Constructors with Measures We use measures to strengthen the types of data constructors, and we use these strengthened types during construction and destruction (pattern-matching). Let: (1) D be a data constructor, with *unrefined* type $\overline{x} : \overline{\tau} \rightarrow T$ (2) the i-th measure definition with domain T is:

$$\texttt{measure}\ \mathtt{f_i} : \tau = \lambda x.\mathtt{case}\ y = x\ \mathtt{of}\ \{D\,\overline{z} \rightarrow e_i\}$$

Then, the refined type of D is defined:

$$\mathsf{Ty}(D) \doteq \overline{x}:\overline{\tau} \rightarrow \{v : T \mid \wedge_i f_i\ v = e_i[\overline{z} \mapsto \overline{x}]\}$$

Thus, each data constructor's output type is refined to reflect the definition of each of its measures. For example, we use the measures `len`, `isNil`, `isCons`, `sel1`, and `sel2` to strengthen the types of [] and : to:

$$\mathsf{Ty}([]) \doteq \{v : [\mathit{Int}] \mid r_{[]}\}$$
$$\mathsf{Ty}(:) \doteq x : \mathtt{Int} \rightarrow \mathit{xs} : [\mathit{Int}] \rightarrow \{v : [\mathit{Int}] \mid r_{:}\}$$

where the output refinements are

$$r_{[]} \doteq \mathtt{len}\ v = 0 \wedge \mathtt{isNil}\ v \wedge \neg\mathtt{isCons}\ v$$
$$r_{:} \doteq \mathtt{len}\ v = 1 + \mathtt{len}\ \mathit{xs} \wedge \neg\mathtt{isNil}\ v \wedge \mathtt{isCons}\ v$$
$$\wedge\ \mathtt{sel1}\ v = x \wedge \mathtt{sel2}\ v = \mathit{xs}$$

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

3.6 Typing Rules

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

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

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

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 τ is well-formed in the environment Γ. Intuitively, the type τ is well-formed if all the refinements in τ are `Bool`-typed in Γ. 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. That is, for any closing substitution θ in the denotation of Γ, for every expression

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

$$\frac{\Gamma, v : B \vdash e : \mathtt{Bool}}{\Gamma \vdash \{v : B \mid e\}}\ \textsc{WF-Base} \qquad \frac{\Gamma \vdash \tau_x \qquad \Gamma, x : \tau_x \vdash \tau}{\Gamma \vdash x : \tau_x \rightarrow \tau}\ \textsc{WF-Fun}$$

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \mathtt{fix}\ : x : \tau \rightarrow y : \tau \rightarrow \tau}\ \textsc{T-Fix}$$

Figure 3. Typing of λ^R

e, if $e \in [\![\theta \cdot \tau_1]\!]$ then $e \in [\![\theta \cdot \tau_2]\!]$. 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 three of the rules are standard [14, 32]. First, rule T-REFLECT is used to strengthen the type of each reflected binder with its definition, as described previously in § 3.4. 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 [14, 21], and is required to equate the reflected functions with their definitions. For example, the application $(\texttt{fib}\ 1)$ is typed as $\{v : \texttt{Int} \mid \texttt{fibP}\ v\ 1 \wedge v = \texttt{fib}\ 1\}$ where the first conjunct comes from the (reflection-strengthened) output refinement of `fib` § 2, and the second conjunct comes from rule T-EXACT. Finally, rule T-FIX is used to type the intermediate `fix` expressions that appear, not in the surface language but as intermediate terms in the operational semantics.

Soundness Following λ^U [32], we can show that evaluation preserves typing and that typing implies denotational inclusion.

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

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:

$$d ::= \bullet \mid \texttt{reflect}\ x : \tau = e\ \texttt{in}\ d$$

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

$$\begin{aligned}\Gamma(\bullet) &\doteq \emptyset \\ \Gamma(\texttt{reflect}\ f : \tau = e\ \texttt{in}\ d) &\doteq (f, \mathsf{Reflect}(\tau, e)),\ \Gamma(d)\end{aligned}$$

A *definition's substitution* $\theta(d)$ maps each binder to its definition:

$$\begin{aligned}\theta(\bullet) &\doteq [] \\ \theta(\texttt{reflect}\ f : \tau = e\ \texttt{in}\ d) &\doteq [[f \mapsto \texttt{fix}\ f\ e],\ \theta(d)]\end{aligned}$$

"Propositions" A *proposition* is a type

$$x_1 : \tau_1 \to \ldots \to x_n : \tau_n \to \{v : \texttt{Unit} \mid \mathit{prop}\}$$

For brevity, we abbreviate propositions like the above to $\overline{x} : \overline{\tau} \to \{\mathit{prop}\}$ and we call *prop* the *proposition's refinement*. For simplicity we assume that $\texttt{fv}(\tau_i) = \emptyset$.

"Validity"

A proposition $\overline{x} : \overline{\tau} \to \{\mathit{prop}\}$ is *valid under d* if

$$\forall \overline{w} \in [\![\overline{\tau}]\!].\ \theta(d) \cdot \mathit{prop}[\overline{x} \mapsto \overline{w}] \hookrightarrow^\star \texttt{True}$$

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

"Proofs" A binder b *proves* a proposition τ under d if

$$\emptyset \vdash d[\texttt{let rec}\ x : \tau = b\ \texttt{in unit}] : \texttt{Unit}$$

That is, if the binder b has the proposition's type τ under the definition d's environment.

Theorem 2. *[Proofs] If b proves τ under d then τ is valid under d.*

Proof. As b proves τ under d, we have

$$\emptyset \vdash d[\texttt{let rec}\ x : \tau = b\ \texttt{in unit}] : \texttt{Unit} \tag{1}$$

By Theorem 1 on 1 we get

$$\theta(d) \in [\![\Gamma(d)]\!] \tag{2}$$

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

$$\forall \theta \in [\![\Gamma(d)]\!].\ \theta \cdot b \in [\![\theta \cdot \tau]\!] \tag{3}$$

Together, 2 and 3 imply

$$\theta(d) \cdot b \in [\![\theta(d) \cdot \tau]\!] \tag{4}$$

By the definition of type denotations, we have

$$[\![\theta(d) \cdot \tau]\!] \ \doteq\ \{f \mid \tau \text{ is valid under } d\} \tag{5}$$

By 4, the above set is not empty, and hence τ is valid under d. □

$$\begin{array}{rrcl}
\textit{\textbf{Predicates}} & r & ::= & r \oplus_2 r \mid \oplus_1 r \\
 & & \mid & n \mid b \mid x \mid D \mid x\ \overline{r} \\
 & & \mid & \texttt{if}\ r\ \texttt{then}\ r\ \texttt{else}\ r \\
\textit{\textbf{Integers}} & n & ::= & 0, -1, 1, \ldots \\
\textit{\textbf{Booleans}} & b & ::= & \texttt{True} \mid \texttt{False} \\
\textit{\textbf{Bin Operators}} & \oplus_2 & ::= & = \mid < \mid \wedge \mid + \mid - \mid \ldots \\
\textit{\textbf{Un Operators}} & \oplus_1 & ::= & \neg \mid \ldots \\
\textit{\textbf{Model}} & \sigma & ::= & \sigma, (x : r) \mid \emptyset \\
\textit{\textbf{Sort Arguments}} & s_a & ::= & \texttt{Int} \mid \texttt{Bool} \mid \texttt{U} \mid \texttt{Fun}\ s_a\ s_a \\
\textit{\textbf{Sorts}} & s & ::= & s_a \to s
\end{array}$$

Figure 4. Syntax of λ^S

Example: Fibonacci is increasing In § 2 we verified that under a definition d that includes `fib`, the term `fibUp` proves

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

Thus, by Theorem 2 we get

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

4. Algorithmic Verification

Next, we describe λ^S, a conservative approximation of λ^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 λ^S

Syntax: Terms & Sorts Figure 4 summarizes the syntax of λ^S, the *sorted* (SMT-) decidable logic of quantifier-free equality, uninterpreted functions and linear arithmetic (QF-EUFLIA) [4, 19]. 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 $x\ \overline{r}$ 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 uninterpreted sort `Fun` s s. The universal sort `U` represents all other values.

Semantics: Satisfaction & Validity An assignment σ is a mapping from variables to terms $\sigma \doteq \{x_1 \mapsto r_1, \ldots, x_n \mapsto r_n\}$. We write $\sigma \models r$ if the assignment σ is a *model of* r, intuitively if $\sigma\ r$ "is true" [19]. A predicate r *is satisfiable* if there exists $\sigma \models r$. A predicate r *is valid* if for all assignments $\sigma \models r$.

Transformation $\boxed{\Gamma \vdash e \rightsquigarrow r}$

$$\frac{}{\Gamma \vdash b \rightsquigarrow b}\ \text{T-BOOL} \qquad \frac{}{\Gamma \vdash n \rightsquigarrow n}\ \text{T-INT}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow r_1 \quad \Gamma \vdash e_2 \rightsquigarrow r_2}{\Gamma \vdash e_1 \oplus_2 e_2 \rightsquigarrow r_1 \oplus_2 r_2}\ \text{T-BIN}$$

$$\frac{\Gamma \vdash e \rightsquigarrow r}{\Gamma \vdash \oplus_1 e \rightsquigarrow \oplus_1 r}\ \text{T-UN} \qquad \frac{}{\Gamma \vdash x \rightsquigarrow x}\ \text{T-VAR}$$

$$\frac{}{\Gamma \vdash c \rightsquigarrow \mathtt{s}_c}\ \text{T-OP} \qquad \frac{}{\Gamma \vdash D \rightsquigarrow \mathtt{s}_D}\ \text{T-DC}$$

$$\frac{\Gamma, x:\tau_x \vdash e \rightsquigarrow r \quad \Gamma \vdash (\lambda x.e) : (x:\tau_x \to \tau)}{\Gamma \vdash \lambda x.e \rightsquigarrow \mathtt{lam}_{(\!|\tau|\!)}^{(\!|\tau_x|\!)}\ x\ r}\ \text{T-FUN}$$

$$\frac{\Gamma \vdash e' \rightsquigarrow r' \quad \Gamma \vdash e \rightsquigarrow r \quad \Gamma \vdash e : \tau_x \to \tau}{\Gamma \vdash e\, e' \rightsquigarrow \mathtt{app}_{(\!|\tau|\!)}^{(\!|\tau_x|\!)}\ r\ r'}\ \text{T-APP}$$

$$\frac{\Gamma \vdash e \rightsquigarrow r \quad \Gamma \vdash e_i[x \mapsto e] \rightsquigarrow r_i}{\begin{array}{lll}\Gamma & \vdash & \mathtt{case}\ x = e\ \mathtt{of}\ \{\mathtt{True} \to e_1; \mathtt{False} \to e_2\} \\ & \rightsquigarrow & \mathtt{if}\ r\ \mathtt{then}\ r_1\ \mathtt{else}\ r_2\end{array}}\ \text{T-IF}$$

$$\frac{\Gamma \vdash e_i[\overline{y_i} \mapsto \overset{\Gamma \vdash e \rightsquigarrow r}{\overline{\mathtt{sel}_{D_i}\ x}}][x \mapsto e] \rightsquigarrow r_i}{\begin{array}{lll}\Gamma & \vdash & \mathtt{case}\ x = e\ \mathtt{of}\ \{D_i\ \overline{y_i} \to e_i\} \\ & \rightsquigarrow & \mathtt{if}\ \mathtt{app}\ \mathtt{is}_{D_1}\ r\ \mathtt{then}\ r_1\ \mathtt{else}\ \ldots\ \mathtt{else}\ r_n\end{array}}\ \text{T-CASE}$$

Figure 5. Transforming λ^R terms into λ^S.

4.2 Transforming λ^R into λ^S

The judgment $\Gamma \vdash e \rightsquigarrow r$ states that a λ^R term e is transformed, under an environment Γ, into a λ^S term r. The transformation rules are summarized in Figure 5.

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

$$\begin{array}{rclrcl}(\!|\mathtt{Int}|\!) & \doteq & \mathtt{Int} & (\!|T|\!) & \doteq & \mathtt{U} \\ (\!|\mathtt{Bool}|\!) & \doteq & \mathtt{Bool} & (\!|x:\tau_x \to \tau|\!) & \doteq & \mathtt{Fun}\ (\!|\tau_x|\!)\ (\!|\tau|\!)\end{array}$$

Embedding Constants Elements shared on both λ^R and λ^S translate to themselves. These elements include booleans (T-BOOL), integers (T-INT), variables (T-VAR), binary (T-BIN) and unary (T-UN) 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 *saturated*, *i.e.* fully applied, *e.g.* by converting source level terms like `(+ 1)` to `(\z -> z + 1)`.

Embedding Functions As λ^S is a first-order logic, we embed λ-abstraction and application using the uninterpreted functions $\mathtt{lam}$ and $\mathtt{app}$. We embed λ-abstractions using $\mathtt{lam}$ as shown in rule T-FUN. The term $\lambda x.e$ of type $\tau_x \to \tau$ is transformed to $\mathtt{lam}_s^{s_x}\ x\ r$ of sort $\mathtt{Fun}\ s_x\ s$, where s_x and s are respectively $(\!|\tau_x|\!)$ and $(\!|\tau|\!)$, $\mathtt{lam}_s^{s_x}$ is a special uninterpreted function of sort $s_x \to s \to \mathtt{Fun}\ s_x\ s$, and x of sort s_x and r of sort s are the embedding of the binder and body, respectively. As $\mathtt{lam}$ 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 defunctionalization [23] using an uninterpreted *apply* function $\mathtt{app}$ as shown in rule T-APP. The term $e\, e'$, where e and e' have types $\tau_x \to \tau$ and τ_x, is transformed to $\mathtt{app}_s^{s_x}\ r\ r' : s$ where s and s_x are respectively $(\!|\tau|\!)$ and $(\!|\tau_x|\!)$, the $\mathtt{app}_s^{s_x}$ is a special uninterpreted function of sort $\mathtt{Fun}\ s_x\ s \to s_x \to s$, 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^S constant $\mathtt{s}_D$ of sort $(\!|\mathsf{Ty}(D)|\!)$. Let D_i be a non-boolean data constructor such that

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

Then the *check function* $\mathtt{is}_{D_i}$ has the sort $\mathtt{Fun}\ (\!|\tau|\!)\ \mathtt{Bool}$, and the *select function* $\mathtt{sel}_{D_{i,j}}$ has the sort $\mathtt{Fun}\ (\!|\tau|\!)\ (\!|\tau_{i,j}|\!)$. Rule T-CASE translates case-expressions of λ^R into nested $\mathtt{if}$ terms in λ^S, 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

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

is reflected into the λ^S refinement:

$$\mathtt{if}\ \mathtt{isNil}\ xs\ \mathtt{then}\ ys\ \mathtt{else}\ \mathtt{sel1}\ xs\ :\ (\mathtt{sel2}\ xs \mathbin{++} ys)$$

We favor selectors to the axiomatic translation of HALO [35] and F* [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 $\mathtt{True}$ *iff* r is SMT-satisfiable by a model that respects β-equivalence.

Definition 1 (β-Model). *A $\beta-$model σ^β is an extension of a model σ where $\mathtt{lam}$ and $\mathtt{app}$ satisfy the axioms of β-equivalence:*

$$\begin{array}{rcl}\forall x\ y\ e.\mathtt{lam}\ x\ e & = & \mathtt{lam}\ y\ (e[x \mapsto y]) \\ \forall x\ e_x\ e.(\mathtt{app}\ (\mathtt{lam}\ x\ e)\ e_x & = & e[x \mapsto e_x]\end{array}$$

Semantics Preservation We define the translation of a λ^R term into λ^S under the empty environment as $(\!|e|\!)\ \doteq\ r$ if $\emptyset \vdash e \rightsquigarrow r$. 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 [32]. We connect the semantics of λ^R and translated λ^S via the following theorems:

Theorem 3. *If $\Gamma \vdash e \rightsquigarrow r$, then for every $\theta \in [\![\Gamma]\!]$ and every $\sigma \in \theta^\perp$, if $\theta^\perp \cdot e \hookrightarrow^\star v$ then $\sigma^\beta \models r = (\!|v|\!)$.*

Corollary 1. *If $\Gamma \vdash e : \mathtt{Bool}$, e reduces to a value and $\Gamma \vdash e \rightsquigarrow r$, then for every $\theta \in [\![\Gamma]\!]$ and every $\sigma \in \theta^\perp$, $\theta^\perp \cdot e \hookrightarrow^\star \mathtt{True}$ iff $\sigma^\beta \models r$.*

4.4 Decidable Type Checking

Figure 6 summarizes the modifications required to obtain decidable type checking. Namely, basic types are extended with labels that track termination and subtyping is checked via an SMT solver.

Termination Under arbitrary beta-reduction semantics (which includes lazy evaluation), soundness of refinement type checking requires checking termination, for two reasons: (1) to ensure that refinements cannot diverge, and (2) to account for the environment during subtyping [32]. We use $\Downarrow$ to mark provably terminating computations, and extend the rules to use refinements to ensure that if $\Gamma \vdash_S e : \{v : B^\Downarrow \mid r\}$, then e terminates [32].

Verification Conditions The *verification condition* (VC) $(\!|\Gamma|\!) \Rightarrow r$ is *valid* only if the set of values described by Γ, is subsumed by the set of values described by r. Γ is embedded into logic by conjoining (the embeddings of) the refinements of provably

$$\textbf{\textit{Refined Types}} \qquad \tau \quad ::= \quad \{v : B^{[\Downarrow]} \mid e\} \ \mid \ x : \tau \rightarrow \tau$$

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

$$\frac{\Gamma, v : B \vdash_S e : \mathtt{Bool}^{\Downarrow}}{\Gamma \vdash_S \{v : B \mid e\}} \ \text{WF-BASE}$$

Subtyping $\boxed{\Gamma \vdash_S \tau \preceq \tau'}$

$$\frac{\Gamma' \ \doteq \ \Gamma, v : \{B^{\Downarrow} | e\} \qquad \Gamma' \vdash e' \rightsquigarrow r' \qquad \mathsf{Valid}((\!|\Gamma'|\!) \Rightarrow r')}{\Gamma \vdash_S \{v : B \mid e\} \preceq \{v : B \mid e'\}} \ \preceq\text{-BASE}$$

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

terminating binders [32]:

$$(\!|\Gamma|\!) \ \doteq \ \bigwedge_{x \in \Gamma} (\!|\Gamma, x|\!)$$

where we embed each binder as

$$(\!|\Gamma, x|\!) \ \doteq \ \begin{cases} r & \text{if } \Gamma(x) = \{v : B^{\Downarrow} \mid e\},\ \Gamma \vdash e[v \mapsto x] \rightsquigarrow r \\ \mathtt{True} & \text{otherwise.} \end{cases}$$

Subtyping via SMT Validity We make subtyping, and hence, typing decidable, by replacing the denotational base subtyping rule $\preceq$ -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* $\Gamma \vdash_S \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}$ *then* $\Gamma \vdash \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}$.

Soundness of λ^S Lemma 2 directly implies the soundness of λ^S.

Theorem 4 (Soundness of λ^S). *If* $\Gamma \vdash_S e : \tau$ *then* $\Gamma \vdash e : \tau$.

5. Reasoning About Lambdas

Though λ^S, as presented so far, is sound and decidable, it is *imprecise*: our encoding of λ-abstractions and applications via uninterpreted functions makes it impossible to prove theorems that require α- and β-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 combinator for safely asserting extensional equality § 5.2. In the rest of this section, for clarity we omit `app` when it is clear from the context.

5.1 Equivalence

As soundness relies on satisfiability under a σ^β (see Definition 1), we can safely *instantiate* the axioms of α- and β-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 λ terms we enforce a *normalization invariant* that for each lambda term $\mathtt{lam}\ x_i\ e$, the index i is greater than any lambda argument appearing in e.

α-instances For each syntactic term $\mathtt{lam}\ x_i\ e$, and λ-binder x_j such that $i < j$ appearing in the VC, we generate an *α-equivalence instance predicate* (or *α-instance*):

$$\mathtt{lam}\ x_i\ e = \mathtt{lam}\ x_j\ e[x_i \mapsto x_j]$$

The conjunction of α-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 `Reader`:

$$\begin{aligned} &d = \mathtt{lam}\ x_1\ (x\ x_1) \\ &\quad \Rightarrow \mathtt{lam}\ x_2\ ((\mathtt{lam}\ x_1\ (x\ x_1))\ x_2) = \mathtt{lam}\ x_1\ (d\ x_1) \end{aligned}$$

The α instance $\mathtt{lam}\ x_1\ (d\ x_1) = \mathtt{lam}\ x_2\ (d\ x_2)$ derived from the VC's hypothesis, combined with congruence immediately yields the VC's consequence.

β-instances For each syntactic term $\mathtt{app}\ (\mathtt{lam}\ x\ e)\ e_x$, with e_x not containing any λ-abstractions, appearing in the VC, we generate an *β-equivalence instance predicate* (or *β-instance*):

$$\mathtt{app}\ (\mathtt{lam}\ x_i\ e)\ e_x = e[x_i \mapsto e_x], \text{ s.t. } e_x \text{ is } \lambda\text{-free}$$

We require the λ-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.

$$(f\ x \gg\!= g) = \mathtt{app}\ (\mathtt{lam}\ y\ (f\ y \gg\!= g))\ x$$

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

Normalization The combination of α- and β-instances is often required to discharge proof obligations. For example, when proving that the bind operator for the `Reader` monad is associative, we need to prove the VC:

$$\mathtt{lam}\ x_2\ (\mathtt{lam}\ x_1\ w) = \mathtt{lam}\ x_3\ (\mathtt{app}\ (\mathtt{lam}\ x_2\ (\mathtt{lam}\ x_1\ w))\ w)$$

The SMT solver proves the VC via the equalities corresponding to an α and then β-instance:

$$\begin{aligned} \mathtt{lam}\ x_2\ (\mathtt{lam}\ x_1\ w) &=_\alpha \mathtt{lam}\ x_3\ (\mathtt{lam}\ x_1\ w) \\ &=_\beta \mathtt{lam}\ x_3\ (\mathtt{app}\ (\mathtt{lam}\ x_2\ (\mathtt{lam}\ x_1\ w))\ w) \end{aligned}$$

5.2 Extensionality

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

```
reflect id
id x = x
```

Liquid Haskell accepts the proof that `id x = x` for all `x`:

```
id_x_eq_x :: x:a → {id x = x}
id_x_eq_x = \x → id x =. x ** QED
```

as "calling" `id` unfolds its definition, completing the proof. However, consider this η-expanded variant of the above proposition:

```
type Id_eq_id = {(\x → id x) = (\y → y)}
```

Liquid Haskell *rejects* the proof:

```
fails :: Id_eq_id
fails =  (\x → id x) =. (\y → y) ** QED
```

The invocation of `id` unfolds the definition, but the resulting equality refinement `{id x = x}` is *trapped* under the λ-abstraction. That is, the equality is absent from the typing environment at the *top* level, where the left-hand side term is compared to `\y → y`. Note that the above equality requires the definition of `id` and hence is outside the scope of purely the α- and β-instances.

CATEGORY			LOC
I.	**Arithmetic**		
	Fibonacci	§ 2	48
	Ackermann	[31] , Fig. 8	280
II.	**Algebraic Data Types**		
	Fold Universal	[18]	105
	Fold Fusion	[18]	
III.	**Typeclasses**	Fig 9	
	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	[7]	133
	Unification	[27]	200
TOTAL			2264

Figure 7. Summary of Case Studies

An Exensionality 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`.

```
=∀ :: f:(a → b) → g:(a → b)
   → exp:(x:a → {f x = g x})
   → {f = g}
```

Of course, =∀ cannot be implemented; its type is *assumed*. We can use =∀ to prove `Id_eq_id` by providing a suitable explanation:

```
pf_id_id :: Id_eq_id
pf_id_id = (\y → y) =∀ (\x → id x) ∵ expl
    ** QED
  where
    expl = (\x → id x =. x ** QED)
```

The explanation is the second argument to ∵ which has the following type that syntactically fires β-instances:

```
x:a → {(\x → id x) x = ((\x → x) x}
```

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 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.

Ackermann's Function

$$A_n(x) \doteq \begin{cases} x+2 & \text{, if } n=0 \\ 2 & \text{, if } x=0 \\ A_{n-1}(A_n(x-1)) \end{cases} \qquad A_n^h(x) \doteq \begin{cases} x & \text{, if } h=0 \\ A_n(A_n^{h-1}(x)) \end{cases}$$

Properties

1.			$A_{n+1}(x)$	$=$	$A_n^x(2)$
2.			$x+1$	$<$	$A_n(x)$
3.			$A_n(x)$	$<$	$A_n(x+1)$
4.	$x<y$	$\Rightarrow$	$A_n(x)$	$<$	$A_n(y)$
5.	$0<x$	$\Rightarrow$	$A_n(x)$	$<$	$A_{n+1}(x)$
6.	$0<x, n<m$	$\Rightarrow$	$A_n(x)$	$<$	$A_m(x)$
7.			$A_n^h(x)$	$<$	$A_n^{h+1}(x)$
8.			$A_n^h(x)$	$<$	$A_n^h(x+1)$
9.	$x<y$	$\Rightarrow$	$A_n^h(x)$	$<$	$A_n^h(y)$
10.		$\Rightarrow$	$A_n^h(x)$	$<$	$A_{n+1}^h(x)$
11.	$0<n, l-2<x$	$\Rightarrow$	$x+l$	$<$	$A_n(x)$
12.	$0<n, l-2<x$	$\Rightarrow$	$A_n^l(x)$	$<$	$A_{n+1}(x)$
13.			$A_n^x(y)$	$<$	$A_{n+1}(x+y)$

Figure 8. Ackermann Properties [31], $\forall n, m, x, y, h, l \geq 0$

6.1 Arithmetic Properties

The first category of theorems pertains to the textbook Fibonacci and Ackermann functions. The former were shown in § 2. The latter are summarized in Figure 8, which shows two alternative definitions for the Ackermann function. We proved equivalence of the definition (Prop 1) and various arithmetic relations between them (Prop 2 — 13), by mechanizing the proofs from [31].

Monotonicity Prop 3. shows that $A_n(x)$ is increasing on x. We derived Prop 4. by applying `fMono` theorem from § 2 with input function the partially applied Ackermann Function $A_n(\star)$. Similarly, we derived the monotonicity Prop 9. by applying `fMono` to the locally increasing Prop. 8 and $A_n^h(\star)$. Prop 5. proves that $A_n(x)$ is increasing on the *first* argument n. As `fMono` applies to the *last* argument of a function, we cannot directly use it to derive Prop 6. Instead, we define a variant `fMono2` that works on the first argument of a binary function, and use it to derive Prop 6.

Constructive Proofs In [31] Prop 12. was proved by constructing an auxiliary *ladder* that counts the number of (recursive) invocations of the Ackermann function, and uses this count to bound $A_n^h(x)$ and $A_n(x)$. It turned out to be straightforward and natural to formalize the proof just by defining the `ladder` function in Haskell, reflecting it, and using it to formalize the algebra from [31].

6.2 Algebraic Data Properties

The second category of properties pertain to algebraic data types.

Fold Univerality Next, we proved properties of list folding, such as the following, describing the *universal* property of right-folds [18]:

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

	Monoid
Left Ident.	$\texttt{mempty}\ x \diamondsuit\ \equiv x$
Right Ident.	$x \diamondsuit \texttt{mempty} \equiv x$
Associativity	$(x \diamondsuit y) \diamondsuit z \equiv x \diamondsuit (y \diamondsuit z)$
	Functor
Ident.	$\texttt{fmap id}\ xs \equiv \texttt{id}\ xs$
Distribution	$\texttt{fmap}\ (g \circ\ h)\ xs \equiv (\texttt{fmap}\ g\ \circ\ \texttt{fmap}\ h)\ xs$
	Applicative
Ident.	$\texttt{pure id} \circledast\ v \equiv v$
Compos.	$\texttt{pure}\ (\circ) \circledast u \circledast v \circledast w \equiv u \circledast (v \circledast w)$
Homomorph.	$\texttt{pure}\ f\ \circledast\ \texttt{pure}\ x \equiv \texttt{pure}\ (f\ x)$
Interchange	$u\ \circledast\ \texttt{pure}\ y \equiv \texttt{pure}\ (\$\ y)\ \circledast\ u$
	Monad
Left Ident.	$\texttt{return}\ a \gg\!= f \equiv f\ a$
Right Ident.	$m \gg\!= \texttt{return}\ \equiv m$
Associativity	$(m \gg\!= f) \gg\!= g \equiv m \gg\!= (\lambda x \to f\ x \gg\!= g)$

Figure 9. Typeclass Laws verified using Liquid Haskell

Our proof `foldr_univ` differs from the one in Agda, in two ways. First, we encode Agda's universal quantification over `x` and `l` in the assumption `stp` using a function type. Second, unlike Agda, Liquid Haskell does not support implicit arguments, so at *uses* of `foldr_univ` the programmer must explicitly provide arguments for `base` and `stp`, as illustrated below.

Fold Fusion Let us define the usual composition operator:

```
reflect . :: (b → c) → (a → b) → a → c
f . g      = \x → f (g x)
```

We can prove the following `foldr_fusion` theorem (that shows operations can be pushed inside a `foldr`), by applying `foldr_univ` to explicit `bas` and `stp` proofs:

```
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 `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→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 $\diamondsuit$ and an *identity* element `mempty`. We use Liquid Haskell to prove that `Peano` (with `add` and `Z`), `Maybe` (with a suitable `mappend` and `Nothing`), and `List` (with append `++` and `[]`) satisfy the monoid laws. For example, we prove that `++` (§ 3.5) 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:

```
assoc :: xs:[a] → ys:[a] → zs:[a] →
       {(xs ++ ys) ++ zs = xs ++ (ys ++ zs)}

assoc [] ys zs     = ([] ++ ys) ++ zs
                   =. [] ++ (ys ++ zs)
                   ** QED
assoc (x:xs) ys zs = ((x:  xs)++ ys) ++ zs
                   =. (x: (xs ++ ys))++ zs
                   =.  x:((xs ++ ys) ++ zs)
                   =.  x: (xs ++ (ys ++ zs))
                     ∵ assoc xs ys zs
                   =. (x:xs)  ++ (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`:

```
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:

```
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 $\gg\!=$ 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:

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

Next, we define an abbreviation for the associativity property:

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

Finally, we can 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:_ →
   {bind (f x) g = (\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 `f` and returns an assignment that *satisfies* `f` 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 {v:a | 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 reflectionm 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 decidable 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 λ-abstractions. Thus, it would be valuable to use recent advances in refinement-based synthesis [22] to automate proof construction. Second, 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 SMT axiomatization, thereby automating proofs predictably.

References

[1] Nada Amin, K. Rustan M. Leino, and Tiark Rompf. Computing with an SMT Solver. In *TAP*, 2014.

[2] Kazuyuki Asada, Ryosuke Sato, and Naoki Kobayashi. Verifying relational properties of functional programs by first-order refinement. In *PEPM*, 2015.

[3] L. Augustsson. Cayenne - a language with dependent types. In *ICFP*, 1998.

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

[5] J. Bengtson, K. Bhargavan, C. Fournet, A.D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*, 2008.

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

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

[8] Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *ESOP*, 2007.

[9] R. L. Constable and S. F. Smith. Partial objects in constructive type theory. In *LICS*, 1987.

[10] Richard A. Eisenberg and Jan Stolarek. Promoting functions to type families in Haskell. In *Haskell*, 2014.

[11] Andrew Farmer, Neil Sculthorpe, and Andy Gill. Reasoning with the HERMIT: Tool support for equational reasoning on GHC Core programs. Haskell, 2015.

[12] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[13] Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. Occurrence typing modulo theories. In *PLDI*, 2016.

[14] K.W. Knowles and C. Flanagan. Hybrid type checking. *ACM TOPLAS*, 2010.

[15] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. LPAR, 2010.

[16] K. Rustan M. Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In *CAV*, 2016.

[17] K. Rustan M. Leino and Nadia Polikarpova. Verified calculations. In *VSTTE*, 2016.

[18] Shin-cheng Mu, Hsiang-shang Ko, and Patrik Jansson. Algebra of Programming in Agda: Dependent Types for Relational Program Derivation. *J. Funct. Program.*, 2009.

[19] G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[20] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers, 2007.

[21] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP TCS*, 2004.

[22] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, 2016.

[23] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *25th ACM National Conference*, 1972.

[24] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[25] P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, 2010.

[26] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in pvs. *IEEE TSE*, 1998.

[27] Vilhelm Sjöberg and Stephanie Weirich. Programming up to congruence. *POPL*, 2015.

[28] W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. In *TACAS*, 2012.

[29] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In *PLDI*, 2016.

[30] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *POPL*, 2016.

[31] George Tourlakis. Ackermanns Function. http://www.cs.yorku.ca/~gt/papers/Ackermann-function.pdf, 2008.

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

[33] Niki Vazou, Alexander Bakst, and Ranjit Jhala. Bounded refinement types. In *ICFP*, 2015.

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

[35] D. Vytiniotis, S.L. Peyton-Jones, K. Claessen, and D. Rosén. Halo: haskell to logic through denotational semantics. In *POPL*, 2013.

[36] GHC Wiki. GHC optimisations. https://wiki.haskell.org/GHC_optimisations.

[37] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, 1998.