Technical Report: Refinement Reflection: Parallel Legacy Languages as Theorem Provers

Abstract

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

1. Introduction

We introduce *refinement reflection*, a method to extend *legacy* languages—with highly tuned libraries, compilers, and run-times—into theorem provers, by letting programmers to specify and verify arbitrary properties of their code simply by writing programs in the legacy language.

Previously, SMT-based refinement types [18, 45] have been used to retrofit so-called "shallow" verification *e.g.* array bounds checking into ML [8, 43, 61], C [17, 44], Haskell [54], TypeScript [57], and Racket [25]. To keep checking decidable, the specifications are restricted to quantifier free formulas over decidable theories. However, to verify "deep" specifications as in theorem proving languages like Agda, Coq, Dafny, F* and Idris we require mechanisms that *represent* and *manipulate* the exact descriptions of user-defined functions. So far, this has entailed either building new languages on specialized type theories, or encoding functions with SMT axioms which makes verification undecidable and hence, unpredictable.

1. Refinement Reflection Our first contribution is the notion of refinement reflection: 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. Specifically, we show how to use ideas for *defunctionalization* from the theorem proving literature which encode functions and lambdas using uninterpreted symbols, to encode terms from an expressive higher order language as decidable refinements, letting us use SMT-based congruence closure for decidable and predictable verification (§ 3).

2. A Library of Proof Combinators Our second contribution is a *library of combinators* that lets programmers *compose proofs* from basic refinements and function definitions. We show how to represent proofs simply as unit-values refined by the proposition that they prove. We show how to build up sophisticated proofs using a small library of combinators that permit reasoning in an algebraic or equational style. Furthermore, since proofs are literally just programs, our proof combinators let us use standard language mechanisms like branches (to encode case splits), recursion (to encode induction), and functions (to encode auxiliary lemmas) to write proofs that look very much like transcriptions of their pencil-and-paper analogues (§ 2).

3. Verified Typeclass Laws Our third contribution is an implementation of refinement reflection in Liquid Haskell, thereby converting the legacy language Haskell into a theorem prover. We demonstrate the benefits of this conversion in two ways. First, Haskell's typeclass machinery has led to a suite of expressive abstractions and optimizations which, for correctness, crucially require that typeclass *instances* obey crucial algebraic laws. We show how reflection can be used to formally verify that many widely used instances of the Monoid, Applicative, Functor and Monad typeclasses actually satisfy the respective laws, making the use of these typeclasses safe (§ 4).

4. Verified Deterministic Parallelism Finally, to showcase the benefits of retrofitting theorem proving onto legacy languages, we perform a case study in *deterministic parallelism*. Existing deterministic languages place unchecked obligations on the user to guarantee, *e.g.* the associativity of a fold. Violations can compromise type soundness and correctness. Closing this gap requires only modest proof effort—touching only a small subset of the application. But for this solution to be possible requires a *practical*, *parallel* programming language that supports deep verification. Before Liquid Haskell there was no such parallel language. We show how Liquid Haskell lets us verify the unchecked obligations from benchmarks taken from three existing parallel programming systems, and thus, paves the way towards high-performance with correctness guarantees (§ 5).

2. Overview

We begin with an 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-fication and verification.

Refinement types are the source program's (here Haskell's) types decorated with logical predicates drawn from a(n SMT decidable) logic [18, 45]. For example, we can define the `Nat` type by refining Haskell's `Int` type with a predicate `0 ≤ v`:

```
type Nat = { v:Int | 0 ≤ v }
```

Here, `v` names the value described by the type: 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 [6]).

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

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

Here, the input type's refinement specifies a *pre-condition* that the parameters must be `Nat`, which is needed to ensure termination, and the output types's refinement specifies a *post-condition* that the result is also a `Nat`. Refinement type checking lets us specify and (automatically) verify the shallow property that if `fib` is invoked with a non-negative `Int`, then it terminates and yields a non-negative `Int`.

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

```
type Prop = ()
```

which can be *refined* with propositions about the code. For example, the following 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 }
```

Function types encode universally quantified propositions:

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

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

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

```
data QED = QED

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

To resemble mathematical proofs, we make this casting postfix. Thus, we write `e ** QED` to cast `e` to a value of `Prop`. For example, we can prove the above propositions 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 anxious about whether the dreaded "bottom", which inhabits all types, makes our proofs suspect. Fortunately, as described in [54], 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 *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`; in the logic, `fib` only satisfies the *congruence axiom*

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

On its own, the uninterpreted function is not terribly useful, as it does not let us prove `fib2_eq_1` which requires reasoning about the *definition* of `fib`.

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

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

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

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

Step 3: Application With the reflected refinement type, each application of `fib` in the code automatically unfolds the `fib` definition *once* in the logic. We prove `fib2_eq_1` by:

```
pf_fib2 :: { fib 2 = 1 }
pf_fib2 = let { t0 = fib 0; t1 = fib 1;
                t2 = fib 2 } in  ()
```

We write `f` to denote places where the unfolding of `f`'s definition is important. Via refinement typing, the above proof yields the following verification condition that is discharged by the SMT solver, even though `fib` is uninterpreted:

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

Note that the verification of `pf_fib2` relies merely on the fact that `fib` was applied to (*i.e.* unfolded at) `0`, `1` and `2`. The SMT solver automatically *combines* the facts, once they are in the antecedent. The following 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 [29] and F* [50], is to encode `fib` using 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 following by *automatically instantiating* the above axiom at `2`, `1` and `0`:

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

The automation offered by axioms is a bit of a devil's bargain, as they render checking of the VCs *undecidable*. In practice, automatic axiom instantation can easily lead to infinite "matching loops". For example, the existence of a term `fib` n in a VC can trigger the above axiom, which may then produce the terms `fib` $(n-1)$ and `fib` $(n-2)$, which may then recursively give rise to further instantiations *ad infinitum*. To prevent matching loops an expert must carefully craft "triggers" and provide a "fuel" parameter [2] 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, per the authors of Dafny, the undecidability of the VC checking and its attendant heuristics makes verification unpredictable [30].

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 up to 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 [36].

"Equation" Combinators We equip Liquid Haskell with 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 "lemmata" 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 use the because combinator to prove that `fib 3 = 2` with a Haskell function:

```
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 solver's arithmetic reasoning, completes the proof.

2.4 Case Study: Deterministic Parallelism

One benefit of an in-language prover is that it lowers the barrier to *small* verification efforts that touch only a fraction of the program, and yet ensure critical invariants that Haskell's type system cannot. Here we consider parallel programming, which is commonly considered error prone, and entails proof obligations on the user that typically go unchecked.

The situation is especially precarious with parallel programming frameworks that claim to be *deterministic* and thus usable within purely functional programs. These include Deterministic Parallel Java (DPJ [12]), Concurrent Revisions for .NET [14], and Haskell's LVish [27], Accelerate [35], and REPA [24]. Accelerate's parallel fold function, for instance, claims to be deterministic—and its purely functional type means the Haskell optimizer will *assume* its referential transparency—but its determinism depends on an associativity guarantee which must be assured *by the programmer* rather than the type system. Thus simply folding the minus function, `fold (-) 0 arr`, is sufficient to violate determinism and Haskell's pure semantics.

Likewise, DPJ goes to pains to develop a new type system for parallel programming, but then provides a "commutes" annotation for methods updating shared state, compromising the *guarantee* and going back to trusting the user. LVish has the same Achilles heel. Consider set insertion:

```
insert :: Ord a ⇒ a → Set s a → Par s ()
```

Here insert returns an (effectful) `Par` computation, which can be run within a pure function to produce a pure result. At first glance it would seem that trusting the implementation of the concurrent set is sufficient to assure a deterministic outcome. Yet the interface has an `Ord` constraint. This polymorphic function works with user-defined data types, and thus user-defined orderings. What if the user fails to implement a total order? Then, even a correct implementation of, e.g. a concurrent skiplist [22], can reveal different insertion orders due to concurrency.

In summary, parallel programs naturally need to communicate, but the mechanisms of that communication—such as folds or inserts into a shared structure—typically carry additional proof obligations. This in turn makes parallelism a liability. But we can remove the risk with verification.

Verified typeclasses Our solution involves simply changing the `Ord` constraint above to `VerifiedOrd`.

```
insert :: VerifiedOrd a ⇒ a → Set s a → Par s ()
```

This constraint changes the interface but not the implementation of `insert`. The additional methods of the verified type class don't add operational capabilities, but rather impose additional proof obligations:

```
class Ord a ⇒ VerifiedOrd a where
 antisym :: x:a → y:a
         → { x ≤ y && y ≤ x ⇒ x = y }
 trans   :: x:a → y:a → z:a
         → { x ≤ y && y ≤ z ⇒ x ≤ z }
 total   :: x:a → y:a → { x ≤ y || y ≤ x }
```

Similarly, we can extend the `Monoid` typeclass to a `VerifiedMonoid`, with refinements expressing `Monoid` laws.

```
class Monoid a ⇒ VerifiedMonoid a where
 lident :: x:a → { mempty <> x = x }
 rident :: x:a → { x <> mempty = x }
 assoc  :: x:a → y:a → z:a
        → { x <> (y <> z) = (x <> y) <> z }
```

The `VerifiedMonoid` typeclass constraint requires the binary operation to be associative, thus can be safely used to fold on an unknown number of processors.

Verified instances for primitive types `VerifiedOrd` instances for primitive types like `Int`, `Double` are trivial to write; they just appeal to the SMT solver's built-in theories. For example, the following is a valid totality proof on `Int`.

```
totInt :: x:Int → y:Int → {x ≤ y || y ≤ x}
totInt _ _ = trivial ** QED
```

Verified instances for algebraic datatypes To prove the class laws for user defined algebraic datatypes refinement reflection allows for structurally inductive proof terms. For example, we can inductively define Peano numerals

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

We can compare two `Peano` numbers via

```
reflect leq :: Peano → Peano → Bool
leq Z _          = True
leq (S n) Z      = False
leq (S n) (S m) = leq n m
```

In § 3 we will describe exactly how the reflection mechanism (illustrated via `fibP`) is extended to account for ADTs

like Peano. Liquid Haskell automatically checks that leq is total [54], which lets us safely **reflect** it into the logic.

Next, we prove that leq is total on Peano numbers

```
totalPeano
 :: n:Peano → m:Peano → {leq n m || leq m n}
 /  [toInt n + toInt m]
totalPeano Z m = leq Z m ** QED
totalPeano n Z = leq Z n ** QED
totalPeano (S n) (S m)
 =  leq (S n) (S m) || leq (S m) (S n)
 =. leq n m || leq m n
 =. True ∵ totalPeano m n
 ** QED
```

The proof goes by induction, splitting cases on whether the number is zero or non-zero. Consequently, we pattern match on the parameters n and m, and furnish separate proofs for each case. In the "zero" cases, we simply unfold the definition of leq. In the "successor" case, after unfolding we (literally) apply the induction hypothesis by using the because operator. The termination hint [toInt n + toInt m], where toInt maps Peano numbers to integers, is used to verify well-formedness of the totalPeano proof term. Liquid Haskell's termination and totality checker use the hint to verify that we are in fact doing induction properly (§ B).

Similarly to totalPeano, we can define the rest of the VerifiedOrd proof methods and use them to create the verified instance.

```
instance Ord Peano where
  (≤) = leq

instance VerifiedOrd Peano where
  total = totalPeano
```

Proving all the four VerifiedOrd laws is a burden on the programmer. Since Peano is isomorphic to Nats, next we present how to reduce the Peano proofs into the SMT automated integer proofs.

Isomorphisms In order to reuse proofs for a custom datatype, we provide a way to translate verified instances between isomorphic types [7]. We design a typeclass Iso which witnesses the fact that two types are isomorphic.

```
class Iso a b where
  to      :: a → b
  from    :: b → a
  to∘from :: x:a → {to (from x) = x}
  from∘to :: x:a → {from (to x) = x}
```

For two isomorphic types a and b we compare instances of b using a's comparison method.

```
instance (Ord a, Iso a b) ⇒ Ord b where
  x ≤ y = from x ≤ from y
```

Then, we prove that VerifiedOrd laws are closed under isomorphisms. For example, we prove totality of comparison on bs using the VerifiedOrd totality on as

```
iso≤Total :: (VerifiedOrd a, Iso a b)
          ⇒ x:b → y:b → {x ≤ y || y ≤ x}
iso≤Total x y
 =  x ≤ y || y ≤ x
 =. (from x) ≤ (from y) || (from y) ≤ (from x)
    ∵ total (from x) (from y)
 ** QED
```

We use iso≤Total to create a verified instance on bs.

```
instance (VerifiedOrd a, Iso a b)
      ⇒ VerifiedOrd b where
  total   = iso≤Total
```

With the above technique, and using Haskell's instances, getting a VerifiedOrd instance for Peano reduces to definition of an Iso Nat Peano.

Proof Composition via Products Finally, we present a mechanism to automatically reduce proofs on product types to proofs of the product components. For example, lexicographic ordering preserves the ordering laws. First, we use class instances to define lexicographic ordering.

```
instance (VerifiedOrd a, VerifiedOrd b)
      ⇒ Ord (a, b) where
  (x1, y1) ≤ (x2, y2) =
    if x1 == x2 then y1 ≤ y2 else x1 ≤ x2
```

Then, we prove that lexicographic ordering preserves the ordering laws. For example, it preserves totality.

```
prod≤Total :: (VerifiedOrd a, VerifiedOrd b)
  ⇒ p:(a, b) → q:(a, b) → {p ≤ q || q ≤ p}
prod≤Total p@(x1, y1) q@(x2, y2)
 =  p ≤ q || q ≤ p
 =. if x1 == x2 then (y1 ≤ y2 || y2 ≤ y1)
                else True ∵ total x1 x2
 =. if x1 == x2 then True
                else True ∵ total y1 y2
 ** QED
```

Finally, using the prod≤Total proof method, we conclude that each instance defined via the lexicographic ordering is indeed a verified instance.

```
instance (VerifiedOrd a, VerifiedOrd b)
      ⇒ VerifiedOrd (a, b) where
  total   = prod≤Total
```

For example the type (Peano, Peano) is derived to be a VerifiedOrd instance.

In short, we can decompose an algebraic datatype into an isomorphic type using sums and products to generate verified instances for arbitrary Haskell datatypes. This could be combined with the Glasgow Haskell Compiler's (GHC) support for generics [32] to automate the derivation of verified instances for user datatypes. In §5, we use these ideas to develop fully safe interfaces to LVish modules, as well as verifying programming patterns from DPJ.

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

Figure 1. Syntax of λ^R

3. Refinement Reflection

Next, we formalize refinement reflection via a core calculus λ^R. We define a decidable SMT language λ^S to approximate the higher order, potentially diverging target language λ^R and present a decidable and sound type system for λ^R.

3.1 Syntax

Figure 8 summarizes the syntax of λ^R, which is essentially the calculus λ^U [54] with explicit recursion and a special `reflect` binding to denote terms that are reflected into the refinement logic. The elements of λ^R are layered into primitive constants, values, expressions, binders and programs.

Constants The primitive constants of λ^R include 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 Data constructors are special constants. For example, the data type $[Int]$, which represents finite lists of integers, has two data constructors: [] (nil) and : (cons).

Values & Expressions The values of λ^R include constants, λ-abstractions $\lambda x.e$, and fully applied data constructors D that wrap values. The expressions of λ^R include values 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 in the program but cannot be inserted into the logic via refinements or reflection. (We *can* allow non-recursive `let` binders in expressions e, but omit them for simplicity.)

3.2 Operational Semantics

We define $\hookrightarrow$ to be the small step, call-by-name β-reduction semantics for λ^R. We evaluate reflected terms $\texttt{reflect}\ x : \tau = e\ \texttt{in}\ p$ as recursive `let` bindings, with extra termination-check constraints imposed by the type system.

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

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

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

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

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* $\{v : B \mid e\}$. We use $\Downarrow$ to mark provably terminating computations, and use refinements to ensure that if $e{:}\{v : B^{\Downarrow} \mid e'\}$, then e terminates [54]. Finally, we have dependent *function types* $x : \tau_x \rightarrow \tau$ where the input x has the type τ_x and the output τ may refer to the input binder x. We write B to abbreviate $\{v : B \mid \texttt{True}\}$, and $\tau_x \rightarrow \tau$ to abbreviate $x : \tau_x \rightarrow \tau$ if x does not appear in τ.

Constants For each constant c we define its type $\mathsf{Ty}(c)$, *e.g.*

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

3.4 Refinement Reflection

The key idea in our work is to *strengthen* the output type of functions with a refinement that *reflects* the definition of the function in the logic. We do this by treating each `reflect`-binder ($\texttt{reflect}\ f : \tau = e\ \texttt{in}\ p$) 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 10).

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

$$\begin{array}{lcl} \mathsf{Reflect}(\{v : B \mid r\}, e) & \doteq & \{v : B \mid r \wedge v = e\} \\ \mathsf{Reflect}(x : \tau_x \rightarrow \tau, \lambda x.e) & \doteq & x : \tau_x \rightarrow \mathsf{Reflect}(\tau, e) \end{array}$$

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

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

Figure 2. Syntax of λ^S.

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 [29] or F* [50], the programmer can predictably "unfold" the definition of the function during a proof simply by "calling" the function, which, as discussed in § 4, we have found to be a very natural way of structuring proofs.

3.5 The SMT logic λ^S

λ^R is a higher order, potentially diverging language that cannot be used for decidable verification. Next, we describe λ^S, a conservative, first order approximation of λ^R where higher order features are approximated with uninterpreted functions, yielding an SMT-based algorithmic logic that enjoys soundness and decidability.

Syntax Figure 11 summarizes the syntax of λ^S, the *sorted* (SMT-) decidable logic of quantifier-free equality, uninterpreted functions and linear arithmetic (QF-EUFLIA) [6, 38]. 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`. The interpreted functions of λ^S, *e.g.* the logical constants $=$ and $<$, have the function sort $s \rightarrow s$. Other functional values in λ^R, *e.g.* reflected λ^R functions and λ-expressions, have the first-order uninterpreted sort `Fun` $s\ s$. The universal sort `U` represents all other values.

3.6 Transforming λ^R into λ^S

A *type environment* Γ is a sequence of type bindings $x_1 : \tau_1, \ldots, x_n : \tau_n$. We use the type environment to define the judgment $\Gamma \vdash e \rightsquigarrow r$ that transforms a λ^R term e into a λ^S term r. Most of the transformation rules are identity and can be found in [1]. Here we discuss the non-identity ones.

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

$$
\begin{array}{rcl@{\qquad}rcl}
(\!|\texttt{Int}|\!) & \doteq & \texttt{Int} & (\!|T|\!) & \doteq & \texttt{U} \\
(\!|\texttt{Bool}|\!) & \doteq & \texttt{Bool} & (\!|x : \tau_x \rightarrow \tau|\!) & \doteq & \texttt{Fun}\ (\!|\tau_x|\!)\ (\!|\tau|\!)
\end{array}
$$

Embedding Constants Elements shared on both λ^R and λ^S translate to themselves. These elements include booleans, integers, variables, binary and unary operators. SMT solvers do not support currying, and so in λ^S, all function symbols must be fully applied. Thus, we assume that all applications to primitive constants and data constructors are fully applied, *e.g.* by converting source terms like `(+ 1)` to `(\z → z+1)`.

Embedding Functions As λ^S is a first-order logic, we embed λ-abstraction using the uninterpreted function `lam`.

$$
\frac{\Gamma, x : \tau_x \vdash e \rightsquigarrow r \qquad \Gamma \vdash (\lambda x.e) : (x : \tau_x \rightarrow \tau)}{\Gamma \vdash \lambda x.e \rightsquigarrow \texttt{lam}^{(\!|\tau_x|\!)}_{(\!|\tau|\!)}\ x\ r}\ \textsc{T-Fun}
$$

The term $\lambda x.e$ of type $\tau_x \rightarrow \tau$ is transformed to $\texttt{lam}^{s_x}_{s}\ x\ r$ of sort `Fun` $s_x\ s$, where s_x and s are respectively $(\!|\tau_x|\!)$ and $(\!|\tau|\!)$, $\texttt{lam}^{s_x}_{s}$ is a special uninterpreted function of sort $s_x \rightarrow s \rightarrow \texttt{Fun}\ s_x\ s$, and x of sort s_x and r of sort s are the embedding of the binder and body, respectively. As `lam` is an SMT-function, it *does not* create a binding for x. Instead, x is renamed to a *fresh* name pre-declared in the SMT logic.

Embedding Applications Dually, we embed applications via defunctionalization [41] with the uninterpreted app function.

$$
\frac{\Gamma \vdash e' \rightsquigarrow r' \qquad \Gamma \vdash e \rightsquigarrow r \qquad \Gamma \vdash e : \tau_x \rightarrow \tau}{\Gamma \vdash e\ e' \rightsquigarrow \mathsf{app}^{(\!|\tau_x|\!)}_{(\!|\tau|\!)}\ r\ r'}\ \textsc{T-App}
$$

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

Embedding Data Types We translate each data constructor to a predefined λ^S constant s_D of sort $(\!|\mathsf{Ty}(D)|\!)$.

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

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

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

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

Due to the simplicity of their syntax the above checkers and selectors can be automatically instantiated in the logic (*i.e.* without actual calls to the reflected functions at source level) using the measure mechanism [56].

To generalize, let D_i be a data constructor such that

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

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

We translate case-expressions of λ^R into nested `if` terms in λ^S, by using the check functions in the guards, and the select functions for the binders of each case.

$$\frac{\Gamma \vdash e \rightsquigarrow r \qquad \Gamma \vdash e_i[\overline{y_i} \mapsto \overline{\texttt{sel}_{D_i}\ x}][x \mapsto e] \rightsquigarrow r_i}{\Gamma \vdash \texttt{case}\ x = e\ \texttt{of}\ \{D_i\ \overline{y_i} \rightarrow e_i\} \rightsquigarrow \texttt{if app is}_{D_1}\ r\ \texttt{then}\ r_1\ \texttt{else}\ \ldots\ \texttt{else}\ r_n}$$

For example, the body of the list append function

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

is reflected into the λ^S refinement:

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

We favor selectors to the axiomatic translation of HALO [58] to avoid universally quantified formulas and the resulting instantiation unpredictability.

3.7 Typing Rules

Next, we present the typing, well-formedness, and subtyping [26, 54] rules of λ^R.

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

Rules All but two of the rules are standard [26, 54]. First, rule T-REFLECT is used to strengthen the type of each reflected binder with its definition, as described previously in § B.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 [26, 40], and is required to equate the reflected functions with their definitions. For example, the application `(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 comes from rule T-EXACT.

Well-formedness A judgment $\Gamma \vdash \tau$ states that the refinement type τ is well-formed in the environment Γ. Following [54], the type τ is well-formed if all the refinements in τ are `Bool`-typed, provably terminating expressions in Γ.

Subtyping A judgment $\Gamma \vdash \tau_1 \preceq \tau_2$ states that the type τ_1 is a subtype of τ_2 in the environment Γ. Informally, τ_1 is a subtype of τ_2 if, when the refinement of τ_1 *implies* the refinement of τ_2 under the assumptions described by Γ. Subtyping of basic types reduces to implication checking.

Verification Conditions The implication or *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 terminating binders [54]:

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

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

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{T-VAR} \qquad \frac{}{\Gamma \vdash c : \mathsf{Ty}(c)}\ \text{T-CON}$$

$$\frac{\Gamma \vdash p : \tau' \qquad \Gamma \vdash \tau' \preceq \tau}{\Gamma \vdash p : \tau}\ \text{T-SUB}$$

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

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

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

$$\frac{\Gamma, x : \tau_x \vdash b_x : \tau_x \qquad \Gamma, x : \tau_x \vdash \tau_x \qquad \Gamma, x : \tau_x \vdash b : \tau \qquad \Gamma \vdash \tau}{\Gamma \vdash \texttt{let rec}\ x : \tau_x = b_x\ \texttt{in}\ b : \tau}\ \text{T-LET}$$

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

$$\frac{\Gamma \vdash e : \{v : T \mid e_r\} \qquad \Gamma \vdash \tau \qquad \forall i. \mathsf{Ty}(D_i) = \overline{y_j : \tau_j} \rightarrow \{v : T \mid e_{r_i}\} \qquad \Gamma, \overline{y_j : \tau_j}, x : \{v : T \mid e_r \wedge e_{r_i}\} \vdash e_i : \tau}{\Gamma \vdash \texttt{case}\ x = e\ \texttt{of}\ \{D_i\ \overline{y_i} \rightarrow e_i\} : \tau}\ \text{T-CASE}$$

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

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

$$\frac{\Gamma \vdash \tau_x \qquad \Gamma, x : \tau_x \vdash \tau}{\Gamma \vdash x : \tau_x \rightarrow \tau}\ \text{WF-FUN}$$

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

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

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

Figure 3. Typing of λ^R

where we embed each binder as

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

CATEGORY			LOC
I.	**Arithmetic**		
	Fibonacci	§ 2	48
	Ackermann	[52]	280
II.	**Algebraic Data Types**		
	Fold Universal	[36]	105
III.	**Typeclasses**	Fig 5	
	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	[15]	133
	Unification	[46]	200
V.	**Deterministic Parallelism**		
	Concurrent Sets	§ 5.1	906
	n-body simulation	§ 5.2	930
	Parallel Reducers	§ 5.3	55
TOTAL			4155

Figure 4. Summary of Case Studies LOC is the total lines of code and specifications.

It is important to note that since λ^S is carefully restricted to SMT-decidable theories, VC checking and thus type checking of λ^R is decidable.

3.8 Soundness

Following λ^U [54], in [1], we show that if validity checking respects the axioms of β-equivalence, then λ^R is sound.

Theorem 1. *[Soundness of λ^R] Assuming the β-equivalence axioms, if $\emptyset \vdash p : \tau$ and $p \hookrightarrow^\star w$ then $\emptyset \vdash w : \tau$.*

Theorem 2 lets us interpret well typed terminating programs as proofs of propositions. For example, in § 2 we verified that `fibUp` $:: n :$ `Nat` $\rightarrow \{$`fib` $n \leq$ `fib` $(n+1)\}$. Via soundness of λ^R, we get runtime monotonicity of `fib`.

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

Approximation of β-equivalence Though sound and precise, directly extending the logic with β-equivalence axioms would render SMT validity checking undecidable. Instead, in [1] we discuss an incomplete, yet decidable, technique that allows the user to manually instantiate the β-equivalence axioms when required for precise typing.

4. 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 4, totalling about 4,000 lines of specifications and proofs. Next, we detail each of the five classes of specifications, illustrate how they were verified using refinement reflection, and discuss the strengths and weaknesses of our approach. *All* of these proofs require refinement reflection, *i.e.* are beyond the scope of shallow refinement typing.

4.1 Arithmetic Properties

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

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

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

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

Using higher order functions like `fMono`, we mechanized the proofs of the Ackermann function properties from [52]. We proved 12 equational and arithmetic properties using various proof techniques, including instantiation of higher order theorems (like `fMono`), proof by construction, and natural number and generalized induction.

4.2 Algebraic Data Properties

The second category of properties pertains to data types.

Fold Universality Next, we proved properties of list folding, such as the *universal* property of right-folds [36]:

```
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}
```

Our proof is very similar to the Agda `foldr_univ` proof [36]. But, 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`. For example, 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)})
```

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

Figure 5. Typeclass Laws verified using Liquid Haskell

```
  → {(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` explicitly prove the base and inductive cases.

4.3 Typeclass Laws

We used Liquid Haskell to prove the Monoid, Functor, Applicative and Monad Laws, summarized in Figure 5, for various user-defined instances summarized in Figure 4.

Monoid Laws A Monoid is a datatype equipped with an associative binary operator $\diamondsuit$ (`mappend`) 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.

Functor Laws A type is a functor if it has a function `fmap` that satisfies the *identity* and *distribution* (or fusion) laws in Figure 5. 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 [60]. We reflect the definition of `map` for lists and use it to specify fusion and verify it by an inductive proof:

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

Monad Laws The monad laws, which relate the properties of the two operators $\gg\!=$ and `return` (Figure 5), refer to λ-functions which are encoded in our logic as uninterpreted functions. For example, we can encode the associativity property as a refinement type alias:

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

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

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

4.4 Functional Correctness

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

Unification We verified the unification of first order terms, as presented in [46]. 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.

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}
```

This example highlights the benefits of partial verification on a legacy programming language: potential diverging code coexists and invokes proof terms.

SAT Solver As another example, we implemented and verified the simple SAT solver used to illustrate and evaluate the features of the dependently typed language Zombie [15]. The solver takes as input a formula `f` and returns an assignment that *satisfies* `f` if one exists, as specified below.

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

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

5. Verified Deterministic Parallelism

Finally, we evaluate our deterministic parallelism prototypes. Aside from the lines of proof code added, we evaluate the impact on runtime performance. Were we using a proof tool external to Haskell, this would not be necessary.

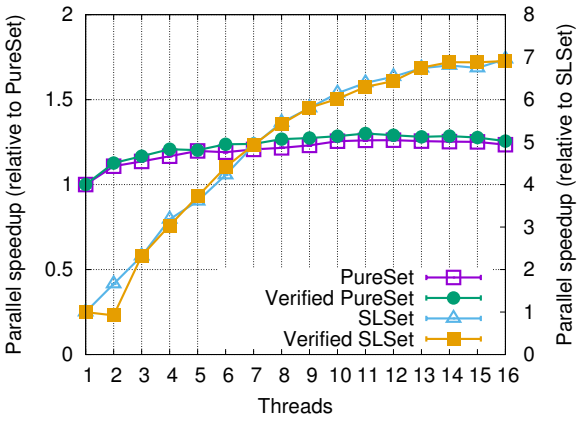

Figure 6. Parallel speedup for doing 1 million parallel inserts over 10 iterations, verified and unverified, relative to the unverified version, for PureSet and SLSet

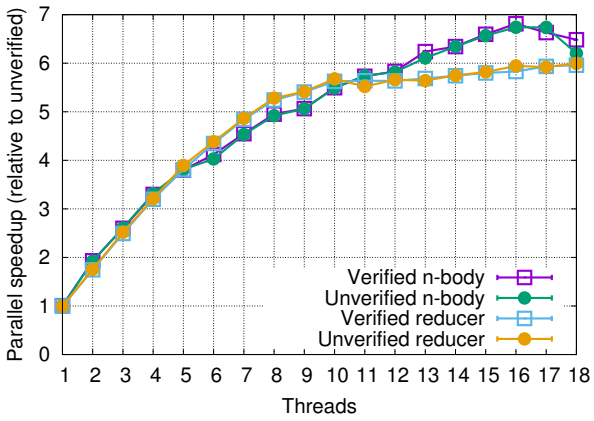

Figure 7. Parallel speedup for doing a parallel n-body simulation and parallel array reduction. The speedup is relative to the unverified version of each respective class of program.

But our proofs are Haskell programs—they are necessarily visible to the compiler. In particular, this means a proliferation of unit values and functions returning unit values. Also, typeclass instances are witnessed at runtime by "dictionary" data structures passed between functions. Layering proof methods on top of existing classes like `Ord` (from § 2.4) could potentially add indirection or change the code generated, depending on the details of the optimizer. In our experiments we find little or no effect on runtime performance. Benchmarks were run on a single-socket Intel® Xeon® CPU E5-2699 v3 with 18 physical cores and 64GiB RAM.

5.1 LVish: Concurrent Sets

First, use the `verifiedInsert` operation (from § 2.4) to observe the runtime slowdown imposed by the extra proof methods of `VerifiedOrd`. We benchmark concurrent sets storing 64-bit integers. Figure 6 compares the parallel speedups for a fixed number of parallel `insert` operations against parallel `verifiedInsert` operations, varying the number of concurrent threads. There is a slight observable difference between the two lines because the extra proof methods do exist at runtime. We repeat the experiment for two set implementations: a concurrent skiplist (SLSet) and a purely functional set inside an atomic reference (PureSet).

5.2 `monad-par`: n-body simulation

Next, we verify deterministic behavior of an n-body simulation program that leverages `monad-par`, a Haskell library which provides deterministic parallelism for pure code [33].

Each simulated particle is represented by a type `Body` that stores its position, velocity and mass. The function `accel` computes the relative acceleration between two bodies:

```
accel :: Body → Body → Accel
```

where `Accel` represents the three-dimensional acceleration

```
data Accel = Accel Real Real Real
```

To compute the total acceleration of a body `b` we (1) compute the relative acceleration between `b` and each body of the system (`Vec Body`), and (2) we add each acceleration component. For efficiency, we use a parallel `mapReduce` for the above computation that first *maps* each vector body to get the acceleration relative to `b` (`accel b`) and then adds each `Accel` value by pointwise addition. `mapReduce` is only deterministic if the element is a `VerifiedMonoid` from § 2.

```
mapReduce :: VerifiedMonoid b ⇒ (a→b) → Vec a → b
```

To enforce the determinism of an n-body simulation, we need to provide a `VerifiedMonoid` instance for `Accel`. We can easily prove that (`Real`, `+`, `0.0`) is a monoid. By product proof composition, we get a verified monoid instance for

```
type Accel' = (Real, (Real, Real))
```

which is isomorphic to `Accel` (*i.e.* `Iso Accel Accel'`).

Figure 7 shows the results of running two versions of the n-body simulation with 2,048 bodies over 5 iterations, with and without verification, using floating point doubles for `Real`[1]. Notably, the two programs have almost identical runtime performance. This demonstrates that even when verifying code that is run in a tight loop (like `accel`), we can expect that our programs won't be slowed down by an unacceptable amount.

5.3 DPJ: Parallel Reducers

The Deterministic Parallel Java (DPJ) project provides a deterministic-by-default semantics for the Java programming language [12]. In DPJ, one can declare a method as `commutative` and thus *assert* that racing instances of that method result in a deterministic outcome. For example:

[1] Floating point numbers notoriously violate associativity, but we use this approximation because Haskell does net yet have an implementation of *superaccumulators* [16].

```
commutative void updateSum(int n) writes R
  { sum += n; }
```

But, DPJ provides no means to formally prove commutativity and thus determinism of parallel reduction. In Liquid Haskell, we specified commutativity as an extra proof method that extends the `VerifiedMonoid` class.

```
class VerifiedMonoid a ⇒ VerifiedComMonoid a where
 commutes :: x:a → y:a → { x <> y = y <> x }
```

Provably commutative appends can be used to deterministically update a reducer variable, since the result is the same regardless of the order of appends. We used LVish [27] to encode a reducer variable with a value `a` and a region `s` as `RVar s a`.

```
newtype RVar s a
```

We specify that safe (*i.e.* deterministic) parallel updates require provably commutative appending.

```
updateRVar :: VerifiedComMonoid a
           ⇒ a → RVar s a → Par s ()
```

Following the DPJ program, we used `updateRVar`'s provably deterministic interface to compute, in parallel, the sum of an array with $3x10^9$ elements by updating a single, global reduction variable using a varying number of threads. Each thread sums segments of an array, sequentially, and updates the variable with these partial sums. In Figure 7, we compare the verified and unverified versions of our implementation to observe no appreciable difference in performance.

6. Related Work

SMT-Based Verification SMT-solvers have been extensively used to automate program verification via Floyd-Hoare logics [38]. Our work is inspired by Dafny's Verified Calculations [31], a framework for proving theorems in Dafny [29], 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* [50] which like Liquid Haskell, uses types to compose proofs, offer more automation by translating recursive functions to SMT axioms. However, unlike reflection, this axiomatic approach renders typechecking and verification undecidable (in theory) and leads to unpredictability and divergence (in practice) [30].

Dependent types Our work is inspired by dependently typed systems like Coq [10] and Agda [39]. Reflection shows how deep specification and verification in the style of Coq and Agda can be *retrofitted* into existing languages via refinement typing. Furthermore, we can use SMT to significantly automate reasoning over important theories like arithmetic, equality and functions. It would be interesting to investigate how the tactics and sophisticated proof search of Coq *etc.* can be adapted to the refinement setting.

Dependent Types in Haskell Integration of dependent types into Haskell has been a long standing goal that dates back to Cayenne [5], a Haskell-like, fully dependent type language with undecidable type checking. In a recent line of work [19] 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 runtime. That is, while both systems automatically lift Haskell code to either uninterpreted logical functions or type families, with refinements, the logical functions are not accessible at run-time, and promotion cannot affect the semantics of the program. As an advantage (resp. disadvantage), refinements cannot degrade (resp. optimize) the performance of programs.

Proving Equational Properties Several authors have proposed tools for proving (equational) properties of (functional) programs. Systems [48] and [4] 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 [47] generates proofs by term rewriting and Halo [58] 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 [20] proves equalities by rewriting the GHC core language, guided by user specified scripts. In contrast, our proofs are simply Haskell programs, we can use SMT solvers to automate reasoning, and, most importantly, we can connect the validity of proofs with the semantics of the programs.

Deterministic Parallelism Deterministic parallelism has plenty of theory but relatively few practical implementations. Early discoveries were based on limited producer-consumer communication, such as single-assignment variables [3, 51], Kahn process networks [23], and synchronous dataflow [28]. Other models use synchronous updates to shared state, as in Esterel [9] or PRAM. Finally, work on type systems for permissions management [37, 59], supports the development of *non-interfering* parallel programs that access disjoint subsets of the heap in parallel. Parallel functional programming is also non-interfering [21, 34]. Irrespective of which theory is used to support deterministic parallel programming, practical implementations such as Cilk [11] or Intel CnC [13] are limited by host languages with type systems insufficient to limit side effects, much less prove associativity. Conversely, dependently typed languages like Agda and Idris do not have parallel programming APIs and runtime systems.

References

[1] Supplementary material.

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

[3] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11:598–632, October 1989.

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

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

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

[7] Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. In *International Conference on Foundations of Software Science and Computation Structures*, pages 57–71. Springer, 2001.

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

[9] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, Jan 2003.

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

[11] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Notices*, 30:207–216, August 1995.

[12] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association.

[13] Zoran Budimlic, Michael Burke, Vincent Cave, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sagnak Tasirlar. The CnC programming model. *Journal of Scientific Programming*, 2010.

[14] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 691–707, New York, NY, USA, 2010. ACM.

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

[16] Sylvain Collange, David Defour, Stef Graillat, and Roman Iakymchuk. Full-Speed Deterministic Bit-Accurate Parallel Floating-Point Summation on Multi- and Many-Core Architectures. working paper or preprint, February 2014.

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

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

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

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

[21] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: a heterogeneous parallel language. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP '07, pages 37–44, New York, NY, USA, 2007. ACM.

[22] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59. ACM, 2004.

[23] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[24] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM.

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

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

[27] Lindsey Kuper, Aaron Turon, Neelakantan R Krishnaswami, and Ryan R Newton. Freeze after writing: quasi-deterministic parallel programming with lvars. In *POPL '14*, 2014.

[28] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, January 1987.

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

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

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

[32] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 37–48, New York, NY, USA, 2010. ACM.

[33] Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 71–82, New York, NY, USA, 2011. ACM.

[34] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 65–78, New York, NY, USA, 2009. ACM.

[35] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional GPU programs. In *ICFP: International Conference on Functional Programming*, pages 49–60. ACM, 2013.

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

[37] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 557–570, New York, NY, USA, 2012. ACM.

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

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

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

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

[42] S. R. Della Rocca and L. Paolini. *The Parametric Lambda Calculus, A Metamodel for Computation*. 2004.

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

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

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

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

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

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

[49] Martin Sulzmann, Gregory J. Duck, Simon P. Jones, and Peter J. Stuckey. Understanding Functional Dependencies via Constraint Handling Rules. *Journal of Functional Programming*, 2006.

[50] 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.

[51] L. G. Tesler and H. J. Enea. A language design for concurrent processes. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 403–408, New York, NY, USA, 1968. ACM.

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

[53] N. Vazou, P. Rondon, and R. Jhala. Abstract refinement types. In *ESOP*, 2013.

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

[55] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Technical Report: Refinement Types for Haskell. http://goto.ucsd.edu/~nvazou/icfp14/haskell-refinements-techrep.pdf, 2014.

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

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

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

[59] Edwin Westbrook, Jisheng Zhao, Zoran Budimlić, and Vivek Sarkar. *Practical Permissions for Race-Free Parallelism*, pages 614–639. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

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

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

A. Implementation: Liquid Haskell

Refinement Reflection is fully implemented in Liquid Haskell and will be included in the next release. The implementation can be found in the Liquid Haskell GitHub repository, all the benchmarks of § 2 and § 4 are included in the tests, the benchmarks of § 5.1 are in the LVars repo, and the rest benchmarks of § 5 are in the verified-instances repo.

Next, we describe the file Proves.hs, the library of proof combinators used by our benchmarks and discuss known limitations of our implementation.

A.1 `Proves`: The Proof Combinators Library

In this section we present `Proves`, a Haskell library used to structure proof terms. `Proves` is inspired by Equational Reasoning Data Types in Adga [36], providing operators to construct proofs for equality and linear arithmetic in Haskell. The constructed proofs are checked by an SMT-solver via Liquid Types.

Proof terms are defined in `Proves` as a type alias for unit, a data type that curries no run-time information

```
type Proof = ()
```

Proof types are refined to express theorems about program functions. For example, the following `Proof` type expresses that `fib 2 == 1`

```
fib2 :: () → {v:Proof | fib 2 == 1}
```

We simplify the above type by omitting the irrelevant basic type `Proof` and variable `v`

```
fib2 :: () → { fib 2 == 1 }
```

`Proves` provides primitives to construct proof terms by casting expressions to proofs. To resemble mathematical proofs, we make this casting post-fix. We write `p *** QED` to cast `p` to a proof term, by defining two operators `QED` and `***` as

```
data QED = QED

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

Proof construction. To construct proof terms, `Proves` provides a proof constructor $\odot$`.` for logical operators of the theory of linear arithmetic and equality: $\{=, \neq, \leq, <, \geq, >\} \in \odot$. $\odot$`. x y` ensures that $x \odot y$ holds, and returns `x`

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

-- for example
==.:: x:a → y:{a| x==y} → {v:a| v==x}
```

For instance, using `==.` we construct a proof, in terms of Haskell code, that `fib 2 == 1`:

```
fib2 _
  =   fib 2
  ==. fib 1 + fib 0
  ==. 1
  *** QED
```

Reusing proofs: Proofs as optional arguments. Often, proofs require reusing existing proof terms. For example, to prove `fib 3 == 2` we can reuse the above `fib2` proof. We extend the proof combinators, to receive an *optional* third argument of `Proof` type.

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

$\odot$`. x y p` returns `x` while the third argument `p` explicitly proves $x \odot y$.

Optional Arguments. The proof term argument is optional. To implement optional arguments in Haskell we use the standard technique where for each operator $\odot$`!` we define a type class `Opt`$\odot$ that takes as input two expressions `a` and returns a result `r`, which will be instantiated with either the result value `r:=a` or a function form a proof to the result `r:=Proof → a`.

```
class Opt⊙ a r where
  (⊙.) :: a → a → r
```

When no explicit proof argument is required, the result type is just an `y:a` that curries the proof `x ⊙ y`

```
instance Opt⊙ a a where
(⊙.) :: x:a→y:{a| x ⊙ y}→{v:a | v==x }
(⊙.) x _ = x
```

Note that Haskell's type inference [49] requires both type class parameters `a` and `r` to be constrainted at class instance matching time. In most our examples, the result type parameter `r` is not constrained at instance matching time, thus due to the Open World Assumption the matching instance could not be determined. To address the above, we used another common Haskell trick, of generalizing the instance to type arguments `a` and `b` and then constraint `a` and `b` to be equal `a~b`. This generalization allows the instance to always match and imposed the equality constraint after matching.

```
instance (a~b)⇒Opt⊙ a b where
(⊙.) :: x:a→y:{x ⊙ y}→{v:b | v==x }
(⊙.) x _ = x
```

To explicitly provide a proof argument, the result type `r` is instantiated to `r:= Proof → a`. For the same instance matching restrictions as above, the type is further generalized to return some `b` that is constraint to be equal to `a`.

```
instance (a~b)⇒Opt⊙ a (Proof→b) where
(⊙.) :: x:a→y:a→{x ⊙ y}→{v:b | v==x }
(⊙.) x _ _ = x
```

As a concrete example, we define the equality operator `==.` via the type class `OptEq` as

```
class OptEq a r where
 (==.):: a → a → r

instance (a~b)⇒OptEq a b where
 (==.)::x:a→y:{a|x==y}→{v:b|v==x}
 (==.) x _ = x

instance (a~b)⇒OptEq a (Proof→b) where
 (==.)::x:a→y:a→{x==y}→{v:b|v==x}
 (==.) x _ _ = x
```

Explanation Operator. The "explanation operator" `(?)`, or `(∵)`, is used to better structure the proofs. `(?)` is an infix operator with same fixity as `(⊙.)` that allows for the equivalence `x ⊙. y ? p == (⊙.) x y p`

```
(?) :: (Proof → a) → Proof → a
f ? y = f y
```

Putting it all together Using the above operators, we prove that `fib 3 == 2`, reusing the previous proof of `fib 2 == 1`, in a Haskell term that resembles mathematical proofs

```
fib3 :: () →  {fib 3 == 2}
fib3 _
  =   fib 3
  ==. fib 2 + fib 1
  ==. 2              ? fib2 ()
  *** QED
```

Unverified Operators All operators in `Proves`, but two are implemented in Haskell with implementations verified by Liquid Haskell. The "unsound" operators are the assume (1). `(==?)` that eases proof construction by assuming equalities, to be proven later and (2). `(=∀)` extentional proof equality.

Assume Operator `(==?)` eases proof construction by assuming equalities while the proof is in process. It is not implemented in that its body is `undefined`. Thus, if we run proof terms including assume operator, the proof will merely crash (instead of returning `()`). Proofs including the assume operator are not considered complete, as via assume operator any statement can be proven,

Function Extensional Equality Unlike the assume operator that is undefined and included in unfinished thus unsound proofs, the functions extensionality is included in valid proofs that assume function extensionality, an axioms that is assumed, as it cannot be proven by our logic.

To allow function equality via extensionality, we provide the user with a function comparison operator that for each function `f` and `g` it transforms a proof that for every argument `x`, `f x = g x` to a proof on function equality `f = g`.

```
(=∀) :: Arg a ⇒ f:(a → b) → g:(a → b)
     → p:(x:a → {f x = g x})
     → {f = g}
```

The function `(=∀)` is not implemented in the library: it returns () and its type is assumed. But soundness of its usage requires the argument type variable `a` to be constrained by a type class constraint `Arg a`, for both operational and type theoretic reasons.

From *operational* point of view, an implementation of `(=∀)` would require checking equality of `f x = g x` *forall* arguments `x` of type `a`. This equality would hold due to the proof argument `p`. The only missing point is a way to enumerate all the argument `a`, but this could be provided by a method of the type clas `Arg a`. Yet, we have not implement `(=∀)` because we do not know how to provide such an implementation that can provably satisfy `(=∀)`'s type.

From *type theoretic* point of view, the type variable argument `a` appears only on negative positions. Liquid type inference is smart enough to infer that since `a` appears only negative `(=∀)` cannot use any `a` and thus will not call any of its argument arguments `f`, `g`, nor the `p`. Thus, at each call site of `(=∀)` the type variable 'a' is instantiated with the refinement type `{v:a | false}` indicating dead-code (since as will not be used by the callee.) Refining the argument `x:a` with false at each call-site though leads to unsoundness, as each proof argument `p` is a valid proof under the false assumption. What Liquid inference cannot predict is our intention to call `f`, `g` and `p` at *every possible argument*. This information is capture by the type class constraint `Arg a` that (as discussed before [53]) states that methods of the type class `Arg a` may create values of type `a`, thus, due to lack of information on the values that are created by the methods of `Arg a`, `a` can only be refined with `True`.

With extensional equality, we can prove that `\x → x` is equal to `\x → id x`, by providing an explicit explanation that if we call both these functions with the same argument `x`, they return the same result, for each `x`.

```
safe :: Arg a ⇒ a
     → {(\x → id x) = (\x → x)}
safe _  = (\x → x)
        =∀(\x → id x) ∵ (exp ())

exp :: Arg a ⇒ a → x:a
    → {(\x → id x) x = (\x → x) x}
exp _ x =  id x
        ==. x
        *** QED
```

Note that the result of `exp` is an equality of the redexes `(\x → id x) x` and `((\x → x) x`. Extentional function equality requires as argument an equality on such redexes. Via β equality instantiations, both such redexes will automatically reduce, requiring `exp` to prove `id x = x`, with is direct.

Admittedly, proving function equality via extensionality is requires a cumbersome indirect proof. For each function equality in the main proof one needs to define an explanation function that proves the equality for every argument.

A.2 Engineering Limitations

The theory of refinement reflection is fully implemented in Liquid Haskell. Yet, to make this extension *usable* in *real world applications* there are four known engineering limitations that need to be addressed. All these limitations seem straightforward to address and we plan to fix them soon.

The language of refinements is typed lambda calculus. That is the types of the lambda arguments are explicitly specified instead of being inferred. As another minor limitation, the refinement language parser requires the argument to be enclosed in parenthesis in applications where the function is not a variable. Thus the Haskell expression `(\x → x) e` should be written as `(\x:a → x) (e)` in the refinement logic,

Class instances methods can not be reflected. Instead, the methods we want to use in the theorems/propositions should be defined as Haskell functions. This restriction has two major implications. Firstly, we can not verify correctness of library provided instances but we need to redifine them ourselves. Secondly, we cannot really verify class instances with class preconditions. For example, during verification of monoid associativity of the Maybe instance

```
instance (Monoid a) ⇒ Monoid (Maybe a)
```

there is this `Monoid a` class constraint assumption we needed to raise to proceed verification.

Only user defined data types can currently used in verification. The reason for this limitation is that reflection of case expressions requires checker and projector measures for each data type used in reflected functions. Thus, not only should these data types be defined in the verified module, but also should be be injected in the logic by providing a refined version of the definition that can (or may not) be trivially refined.

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

Figure 8. Syntax of λ^R

For example, to reflect a function that uses Peano numbers, the Haskell *and* the refined Peano definitions should be provided

```
data Peano = Z | S Peano

{-@ data Peano [toInt]
      = Z
      | S {prev :: Peano}
  @-}
```

Note that the termination function `toInt` that maps `Peano` numbers to natural numbers is also crucial for soundness of reflection.

There is no module support. All reflected definitions, including, measures (automatically generated checkers and selector, but also the classic lifted Haskell functions to measures) and the reflected types of the reflected functions, are not exposed outside of the module they are defined. Thus all definitions and propositions should exist in the same module.

B. 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.

B.1 Syntax

Figure 8 summarizes the syntax of λ^R, which is essentially the calculus λ^U [54] 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 8). This choice allows us to prove preservation and progress, but renders typechecking undecidable. In § D 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.

Contexts $\boxed{C}$

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

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

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

Figure 9. Operational Semantics of λ^R

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").

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.)

B.2 Operational Semantics

Figure 8 summarizes the small step contextual β-reduction semantics for λ^R. We write $e \hookrightarrow^j e'$ if there exist $e_1, \ldots, e_j$ 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 $(f = f') \hookrightarrow$ `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 § F.2.

B.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 [26]. 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:

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

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

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

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 $\delta(c, w) \in [\![\tau[x \mapsto w]]\!]$.

B.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: `reflect` $f : \tau = e$ `in` p as a `let rec`-binder: `let rec` $f : \mathsf{Reflect}(\tau, e) = e$ `in` p during type checking (rule T-REFLECT in Figure 10).

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

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

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 [29] or F* [50] 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 § 4.

B.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 [54]:

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

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 § D.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 $\overline{x} : \overline{\tau} \to T$ (2) the i-th measure definition with domain T is:

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

Then, the refined type of D is defined:

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

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:

$$\begin{aligned}\mathsf{Ty}([]) &\doteq \{v : [Int] \mid r_{[]}\}\\ \mathsf{Ty}(:) &\doteq x : \mathtt{Int} \to xs : [Int] \to \{v : [Int] \mid r_{:}\}\end{aligned}$$

where the output refinements are

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

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

B.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 [26, 54]. 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 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 [26, 54]. First, rule T-REFLECT is used to strengthen the type of each reflected binder with its definition, as described previously in § B.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 [26, 40], and is required to equate the reflected functions with their definitions. For example, the application (`fib 1`) is typed as $\{v : \mathtt{Int} \mid \mathtt{fibP}\ v\ 1 \wedge v = \mathtt{fib}\ 1\}$ where the first conjunct comes from the (reflection-strengthened) output refinement

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

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{T-VAR} \qquad \frac{}{\Gamma \vdash c : \mathsf{Ty}(c)}\ \text{T-CON}$$

$$\frac{\Gamma \vdash p : \tau' \quad \Gamma \vdash \tau' \preceq \tau}{\Gamma \vdash p : \tau}\ \text{T-SUB}$$

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

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

$$\frac{\Gamma \vdash e_1 : (x : \tau_x \to \tau) \quad \Gamma \vdash e_2 : \tau_x}{\Gamma \vdash e_1\ e_2 : \tau}\ \text{T-APP}$$

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

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

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

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

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

$$\frac{\Gamma \vdash \tau_x \quad \Gamma, x : \tau_x \vdash \tau}{\Gamma \vdash x : \tau_x \to \tau}\ \text{WF-FUN}$$

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

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

$$\frac{\Gamma \vdash \tau_x' \preceq \tau_x \quad \Gamma, x : \tau_x' \vdash \tau \preceq \tau'}{\Gamma \vdash x : \tau_x \to \tau \preceq x : \tau_x' \to \tau'}\ \preceq\text{-FUN}$$

Figure 10. Typing of λ^R

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 [54], we can show that evaluation preserves typing and that typing implies denotational inclusion.

Theorem 2. *[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* $\emptyset \vdash w : \tau$.

B.7 From Programs & Types to Propositions & Proofs

The denotational soundness Theorem 2 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:

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

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

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

"Propositions" A *proposition* is a type

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

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

"Validity"

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

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

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

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

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

Theorem 3. *[Proofs] If b proves τ under d then τ is valid under d.*

Proof. As b proves τ under d, we have

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

By Theorem 2 on 1 we get

$$\theta(d) \in [\![\Gamma(d)]\!] \qquad (2)$$

Furthermore, by the typing rules 1 implies $\Gamma(d) \vdash b : \tau$ and hence, via Theorem 2

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

Together, 2 and 3 imply

$$\theta(d) \cdot b \in [\![\theta(d) \cdot \tau]\!] \qquad (4)$$

By the definition of type denotations, we have

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

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

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 3 we get

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

C. Proof of Soundness

We prove Theorem 2 of § B by reduction to Soundness of λ^U [54].

Theorem 4. *[Denotations] If* $\Gamma \vdash p : \tau$ *then* $\forall \theta \in [\![\Gamma]\!].\theta \cdot p \in [\![\theta \cdot \tau]\!]$.

Proof. We use the proof from [55] and specifically Lemma 4 that is identical to the statement we need to prove. Since the proof proceeds by induction in the type derivation, we need to ensure that all the modified rules satisfy the statement.

- T-Exact Assume $\Gamma \vdash e : \{v : B \mid e_r \wedge v = e\}$. By inversion $\Gamma \vdash e : \{v : B \mid e_r\}$(1). By (1) and IH we get $\forall \theta \in [\![\Gamma]\!].\theta \cdot e \in [\![\theta \cdot \{v : B \mid e_r\}]\!]$. We fix a $\theta \in [\![\Gamma]\!]$ We get that if $\theta \cdot e \hookrightarrow^\star w$, then $\theta \cdot e_r[v \mapsto w] \hookrightarrow^\star \texttt{True}$. By the Definition of $=$ we get that $w = w \hookrightarrow^\star \texttt{True}$. Since $\theta \cdot (v = e)[v \mapsto w] \hookrightarrow^\star w = w$, then $\theta \cdot (e_r \wedge v = e)[v \mapsto w] \hookrightarrow^\star \texttt{True}$. Thus $\theta \cdot e \in [\![\theta \cdot \{v : B \mid e_r \wedge v = e\}]\!]$ and since this holds for any fixed θ, $\forall \theta \in [\![\Gamma]\!].\theta \cdot e \in [\![\theta \cdot \{v : B \mid e_r \wedge v = e\}]\!]$.
- T-Let Assume $\Gamma \vdash \texttt{let rec}\ x : \tau_x = e_x\ \texttt{in}\ p : \tau$. By inversion $\Gamma, x : \tau_x \vdash e_x : \tau_x$ (1), $\Gamma, x : \tau_x \vdash p : \tau$ (2), and $\Gamma \vdash \tau$ (3). By IH $\forall \theta \in [\![\Gamma, x : \tau_x]\!].\theta \cdot e_x \in [\![\theta \cdot \tau_x]\!]$ (1') $\forall \theta \in [\![\Gamma, x : \tau_x]\!].\theta \cdot p \in [\![\theta \cdot \tau]\!]$ (2'). By (1') and by the type of `fix` $\forall \theta \in [\![\Gamma, x : \tau_x]\!].\theta \cdot \texttt{fix}\ x\ e_x \in [\![\theta \cdot \tau_x]\!]$. By which, (2') and (3) $\forall \theta \in [\![\Gamma]\!].\theta \cdot p[x \mapsto \texttt{fix}\ x\ e_x] \in [\![\theta \cdot \tau]\!]$.

- T-REFLECT Assume $\Gamma \vdash \texttt{reflect}\ f : \tau_f = e\ \texttt{in}\ p : \tau$. By inversion, $\Gamma \vdash \texttt{let rec}\ f : \mathsf{Reflect}(\tau_f, e) = e\ \texttt{in}\ p : \tau$. By IH, $\forall \theta \in [\![\Gamma]\!].\theta \cdot \texttt{let rec}\ f : \mathsf{Reflect}(\tau_f, e) = e\ \texttt{in}\ p \in [\![\theta \cdot \tau]\!]$. Since denotations are closed under evaluation, $\forall \theta \in [\![\Gamma]\!].\theta \cdot \texttt{reflect}\ f : \mathsf{Reflect}(\tau_f, e) = e\ \texttt{in}\ p \in [\![\theta \cdot \tau]\!]$.
- T-FIX In Theorem 8.3 from [55] (and using the textbook proofs from [42]) we proved that for each type τ, $\texttt{fix}_\tau \in [\![(\tau \to \tau) \to \tau]\!]$.

□

Theorem 5. *[Preservation] If $\emptyset \vdash p : \tau$ and $p \hookrightarrow^\star w$ then $\emptyset \vdash w : \tau$.*

Proof. In [55] proof proceeds by iterative application of Type Preservation Lemma 7. Thus, it suffices to ensure Type Preservation in λ^R, which it true by the following Lemma. □

Lemma 2. *If $\emptyset \vdash p : \tau$ and $p \hookrightarrow p'$ then $\emptyset \vdash p' : \tau$.*

Proof. Since Type Preservation in λ^U is proved by induction on the type derivation tree, we need to ensure that all the modified rules satisfy the statement.

- T-EXACT Assume $\emptyset \vdash p : \{v : B \mid e_r \wedge v = p\}$. By inversion $\emptyset \vdash p : \{v : B \mid e_r\}$. By IH we get $\emptyset \vdash p' : \{v : B \mid e_r\}$. By rule T-EXACT we get $\emptyset \vdash p' : \{v : B \mid e_r \wedge v = p'\}$. Since subtyping is closed under evaluation, we get $\emptyset \vdash \{v : B \mid e_r \wedge v = p'\} \preceq \{v : B \mid e_r \wedge v = p\}$. By rule T-SUB we get $\emptyset \vdash p' : \{v : B \mid e_r \wedge v = p\}$.
- T-LET Assume $\emptyset \vdash \texttt{let rec}\ x : \tau_x = e_x\ \texttt{in}\ p : \tau$. By inversion, $x : \tau_x \vdash e_x : \tau_x$ (1), $x : \tau_x \vdash p : \tau$ (2), and $\Gamma \vdash \tau$ (3). By rule T-FIX $x : \tau_x \vdash \texttt{fix}\ x\ e_x : \tau_x$ (1'). By (1'), (2) and Lemma 6 of [55], we get $\vdash p[x \mapsto \texttt{fix}\ x\ e_x] : \tau[x \mapsto \texttt{fix}\ x\ e_x]$. By (3) $\tau[x \mapsto \texttt{fix}\ x\ e_x] \equiv \tau$. Since $p' \equiv p[x \mapsto \texttt{fix}\ x\ e_x]$, we have $\emptyset \vdash p' : \tau$.
- T-REFLECT Assume $\emptyset \vdash \texttt{reflect}\ x : \tau_x = e_x\ \texttt{in}\ p : \tau$. By double inversion, with $\tau'_x \equiv \mathsf{Reflect}(\tau_x, e_x)$; $x : \tau'_x \vdash e_x : \tau'_x$ (1), $x : \tau'_x \vdash p : \tau$ (2), and $\Gamma \vdash \tau$ (3). By rule T-FIX $x : \tau'_x \vdash \texttt{fix}\ x\ e_x : \tau'_x$ (1'). By (1'), (2) and Lemma 6 of [55], we get $\vdash p[x \mapsto \texttt{fix}\ x\ e_x] : \tau[x \mapsto \texttt{fix}\ x\ e_x]$. By (3) $\tau[x \mapsto \texttt{fix}\ x\ e_x] \equiv \tau$. Since $p' \equiv p[x \mapsto \texttt{fix}\ x\ e_x]$, we have $\emptyset \vdash p' : \tau$.
- T-FIX This case cannot occur, as `fix` does not evaluate to any program.

□

D. 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.

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

Figure 11. Syntax of λ^S

D.1 The SMT logic λ^S

Syntax: Terms & Sorts Figure 11 summarizes the syntax of λ^S, the *sorted* (SMT-) decidable logic of quantifier-free equality, uninterpreted functions and linear arithmetic (QF-EUFLIA) [6, 38]. 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 $\texttt{Fun}\ s\ s$. 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" [38]. A predicate r *is satisfiable* if there exists $\sigma \models r$. A predicate r *is valid* if for all assignments $\sigma \models r$.

D.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 12.

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

$$(\!|\texttt{Int}|\!) \doteq \texttt{Int} \qquad (\!|T|\!) \doteq \texttt{U}$$
$$(\!|\texttt{Bool}|\!) \doteq \texttt{Bool} \qquad (\!|x : \tau_x \to \tau|\!) \doteq \texttt{Fun}\ (\!|\tau_x|\!)\ (\!|\tau|\!)$$

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

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

$$\frac{}{\Gamma \vdash b \rightsquigarrow b}\ \text{T-Bool} \qquad \frac{}{\Gamma \vdash n \rightsquigarrow n}\ \text{T-Int}$$

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

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

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

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

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

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

$$\frac{\Gamma \vdash e \rightsquigarrow r \quad \Gamma \vdash e_i[\overline{y_i} \mapsto \overline{\mathtt{sel}_{D_i}\ x}][x \mapsto e] \rightsquigarrow r_i}{\Gamma \vdash \mathtt{case}\ x = e\ \mathtt{of}\ \{D_i\ \overline{y_i} \to e_i\} \rightsquigarrow \mathtt{if}\ \mathsf{app}\ \mathtt{is}_{D_1}\ r\ \mathtt{then}\ r_1\ \mathtt{else}\ \ldots\ \mathtt{else}\ r_n}\ \text{T-Case}$$

Figure 12. Transforming λ^R terms into λ^S.

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 `lam` and `app`. We embed λ-abstractions using `lam` as shown in rule T-Fun. The term $\lambda x.e$ of type $\tau_x \to \tau$ is transformed to $\mathtt{lam}^{s_x}_{s}\ x\ r$ of sort `Fun` s_x s, where s_x and s are respectively $(\!|\tau_x|\!)$ and $(\!|\tau|\!)$, $\mathtt{lam}^{s_x}_{s}$ is a special uninterpreted function of sort $s_x \to s \to$ `Fun` s_x s, and x of sort s_x and r of sort s are the embedding of the binder and body, respectively. As `lam` is 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 [41] using an uninterpreted *apply* function `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 $\mathsf{app}^{s_x}_{s}\ r\ r' : s$ where s and s_x are respectively $(\!|\tau|\!)$ and $(\!|\tau_x|\!)$, the $\mathsf{app}^{s_x}_{s}$ is a special uninterpreted function of sort `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 `Fun` $(\!|\tau|\!)$ `Bool`, and the *select function* $\mathtt{sel}_{D_{i,j}}$ has the sort `Fun` $(\!|\tau|\!)$ $(\!|\tau_{i,j}|\!)$. Rule T-Case translates case-expressions of λ^R into nested `if` terms in λ^S, by using the check functions in the guards, and the select functions for the binders of each case. 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:

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

We favor selectors to the axiomatic translation of HALO [58] and F* [50] to avoid universally quantified formulas and the resulting instantiation unpredictability.

D.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` *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* `lam` *and* `app` *satisfy the axioms of β-equivalence:*

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

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 [54]. We connect the semantics of λ^R and translated λ^S via the following theorems:

Theorem 6. *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 $\sigma \in \theta^\perp$, $\theta^\perp \cdot e \hookrightarrow^\star$ `True` iff $\sigma^\beta \models r$.*

Refined Types $\quad \tau \quad ::= \quad \{v : B^{[\Downarrow]} \mid e\} \mid x : \tau \to \tau$

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

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

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

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

Figure 13. Algorithmic Typing (other rules in Figs 8 and 10.)

D.4 Decidable Type Checking

Figure 13 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 [54]. 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 [54].

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 terminating binders [54]:

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

where we embed each binder as

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

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 2 to prove soundness of subtyping.

Lemma 3. *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 3 directly implies the soundness of λ^S.

Theorem 7 (Soundness of λ^S)**.** *If* $\Gamma \vdash_S e : \tau$ *then* $\Gamma \vdash e : \tau$.

E. Soundness of Algorithmic Verification

In this section we prove soundness of Algorithmic verification, by proving the theorems of § D by referring to the proofs in [55].

E.1 Transformation

Definition 2 (Initial Environment)**.** *We define the initial SMT environment* Δ_0 *to include*

$\mathtt{s}_c$	:	$(\!|\mathsf{Ty}(c)|\!)$	$\forall c \in \lambda^R$
$\mathtt{lam}_s^{s_x}$	:	$s_x \to s \to \mathtt{Fun}\ s_x\ s$	$\forall s_x, s \in \lambda^S$
$\mathtt{app}_s^{s_x}$	:	$\mathtt{Fun}\ s_x\ s \to s_x \to s$	$\forall s_x, s \in \lambda^S$
$\mathtt{s}_D$	:	$(\!|\mathsf{Ty}(D)|\!)$	$\forall D \in \lambda^R$
$\mathtt{is}_D$	:	$(\!|T \to \mathtt{Bool}|\!)$	$\forall D \in \lambda^R$ *of data type* T
$\mathtt{sel}_{D_i}$	:	$(\!|T \to \tau_i|\!)$	$\forall D \in \lambda^R$ *of data type* T *and* i*-th argument* τ_i
x_i^s	:	s	$\forall s \in \lambda^S$ *and* $1 \le i \le M_\lambda$

Where x_i^s *are* M_λ *global names that only appear as lambda arguments.*

We modify the T-FUN rule to ensure that logical abstraction is performed using the minimum globally defined lambda argument that is not already abstracted. We do so, using the helper function $\mathtt{MaxLam}(s, r)$:

$$\begin{aligned} \mathtt{MaxLam}(s, \mathtt{lam}_{s'}^{s}\ x_i^s\ r) &= \mathtt{max}(i, \mathtt{MaxLam}(s, r)) \\ \mathtt{MaxLam}(s, r\ \overline{r}) &= \mathtt{max}(\mathtt{MaxLam}(s, r, \overline{r})) \\ \mathtt{MaxLam}(s, r_1 \oplus_2 r_2) &= \mathtt{max}(\mathtt{MaxLam}(s, r_1, r_2)) \\ \mathtt{MaxLam}(s, \oplus_1 r) &= \mathtt{MaxLam}(s, r) \\ \mathtt{MaxLam}(s, \mathtt{if}\ r\ \mathtt{then}\ r_1\ \mathtt{else}\ r_2) &= \mathtt{max}(\mathtt{MaxLam}(s, r, r_1, r_2)) \\ \mathtt{MaxLam}(s, r) &= 0 \end{aligned}$$

$$\frac{i = \mathtt{MaxLam}((\!|\tau_x|\!), r) \quad i < M_\lambda \quad y = x_{i+1}^{(\!|\tau_x|\!)} \quad \Gamma, y : \tau_x \vdash e[x \mapsto y] \rightsquigarrow r \quad \Gamma \vdash (\lambda x.e) : (x : \tau_x \to \tau)}{\Gamma \vdash \lambda x.e \rightsquigarrow \mathtt{lam}_{(\!|\tau|\!)}^{(\!|\tau_x|\!)}\ y\ r} \text{ T-FUN}$$

Lemma 4 (Type Transformation)**.** *If* $\Gamma \vdash e \rightsquigarrow p$, *and* $\Gamma \vdash e : \tau$, *then* $\Delta_0, (\!|\Gamma|\!) \vdash_S p{:}(\!|\tau|\!)$.

Proof. We proceed by induction on the translation

- T-BOOL: Since $(\!|\mathtt{Bool}|\!) = \mathtt{Bool}$, If $\Gamma \vdash b : \mathtt{Bool}$, then $\Delta_0, (\!|\Gamma|\!) \vdash_S b{:}(\!|\mathtt{Bool}|\!)$.
- T-INT: Since $(\!|\mathtt{Int}|\!) = \mathtt{Int}$, If $\Gamma \vdash n : \mathtt{Int}$, then $\Delta_0, (\!|\Gamma|\!) \vdash_S n{:}(\!|\mathtt{Int}|\!)$.
- T-UN: Since $\Gamma \vdash \neg\, e : \tau$, then it should be $\Gamma \vdash e : \mathtt{Bool}$ and $\tau \equiv \mathtt{Bool}$. By IH, $\Delta_0, (\!|\Gamma|\!) \vdash_S r{:}(\!|\mathtt{Bool}|\!)$, thus $\Delta_0, (\!|\Gamma|\!) \vdash_S \neg r{:}(\!|\mathtt{Bool}|\!)$.

- T-BIN Assume $\Gamma \vdash e_1 \oplus_2 e_2 \rightsquigarrow r_1 \oplus_2 r_2$. By inversion $\Gamma \vdash e_1 \rightsquigarrow r_1$, and $\Gamma \vdash e_2 \rightsquigarrow r_2$. Since $\Gamma \vdash e_1 \oplus_2 e_2 : \tau$, then $\Gamma \vdash e_1 : \tau_1$ and $\Gamma \vdash e_1 : \tau_2$. By IH, $\Delta_0, (\!|\Gamma|\!) \vdash_S r_1{:}(\!|\tau_1|\!)$ and $\Delta_0, (\!|\Gamma|\!) \vdash_S r_2{:}(\!|\tau_2|\!)$. We split cases on $\oplus_2$
  - If $\oplus_2 \equiv =$, then $\tau_1 = \tau_2$, thus $(\!|\tau_1|\!) = (\!|\tau_2|\!)$ and $(\!|\tau|\!) = \tau = \texttt{Bool}$.
  - If $\oplus_2 \equiv <$, then $\tau_1 = \tau_2 = \texttt{Int}$, thus $(\!|\tau_1|\!) = (\!|\tau_2|\!) = \texttt{Int}$ and $(\!|\tau|\!) = \tau = \texttt{Bool}$.
  - If $\oplus_2 \equiv \wedge$, then $\tau_1 = \tau_2 = \texttt{Bool}$, thus $(\!|\tau_1|\!) = (\!|\tau_2|\!) = \texttt{Bool}$ and $(\!|\tau|\!) = \tau = \texttt{Bool}$.
  - If $\oplus_2 \equiv +$ or $\oplus_2 \equiv -$, then $\tau_1 = \tau_2 = \texttt{Int}$, thus $(\!|\tau_1|\!) = (\!|\tau_2|\!) = \texttt{Int}$ and $(\!|\tau|\!) = \tau = \texttt{Int}$.
- T-VAR: Assume $\Gamma \vdash x \rightsquigarrow x$ Then $\Gamma \vdash x : \Gamma(x)$ and $\Delta_0, (\!|\Gamma|\!) \vdash_S x{:}(\!|\Gamma|\!)(x)$. But by definition $((\!|\Gamma|\!))(x) = (\!|\Gamma(x)|\!)$.
- T-OP: Assume $\Gamma \vdash c \rightsquigarrow \mathtt{s}_c$ Also, $\Gamma \vdash c : \mathsf{Ty}(c)$ and $\Delta_0, (\!|\Gamma|\!) \vdash_S \mathtt{s}_c{:}\Delta_0(\mathtt{s}_c)$. But by Definition 2 $\Delta_0(\mathtt{s}_c) = (\!|\mathsf{Ty}(c)|\!)$.
- T-DC: Assume $\Gamma \vdash D \rightsquigarrow \mathtt{s}_D$ Also, $\Gamma \vdash D : \mathsf{Ty}(D)$ and $\Delta_0, (\!|\Gamma|\!) \vdash_S \mathtt{s}_D{:}\Delta_0(\mathtt{s}_D)$. But by Definition 2 $\Delta_0(\mathtt{s}_D) = (\!|\mathsf{Ty}(c)|\!)$.
- T-FUN: Assume $\Gamma \vdash \lambda x.e \rightsquigarrow \mathtt{lam}^{(\!|\tau_x|\!)}_{(\!|\tau|\!)}\ x_i^{(\!|\tau_x|\!)}\ r$. By inversion $i \leq M_\lambda$ $\Gamma, x_i^{(\!|\tau_x|\!)} : \tau_x \vdash e[x \mapsto x_i^{(\!|\tau_x|\!)}] \rightsquigarrow r$, and $\Gamma \vdash (\lambda x.e) : (x : \tau_x \to \tau)$. By the Definition 2 on $\mathtt{lam}$, x_i^s and induction, we get $\Delta_0, (\!|\Gamma|\!) \vdash_S \mathtt{lam}^{(\!|\tau_x|\!)}_{(\!|\tau|\!)}\ x_i^{(\!|\tau_x|\!)}\ r{:}\mathtt{Fun}\ (\!|\tau_x|\!)\ (\!|\tau|\!)$. By the definition of the type embeddings we have $(\!|x \to \tau_x \tau|\!) \doteq \mathtt{Fun}\ (\!|\tau_x|\!)\ (\!|\tau|\!)$.
- T-APP: Assume $\Gamma \vdash e\ e' \rightsquigarrow \mathsf{app}^{(\!|\tau_x|\!)}_{(\!|\tau|\!)}\ r\ r'$. By inversion, $\Gamma \vdash e' \rightsquigarrow r'$, $\Gamma \vdash e \rightsquigarrow r$, $\Gamma \vdash e : x : \tau_x \to \tau$. By IH and the type of app we get that $\Delta_0, (\!|\Gamma|\!) \vdash_S \mathsf{app}^{(\!|\tau_x|\!)}_{(\!|\tau|\!)}\ r\ r'{:}(\!|\tau|\!)$.
- T-IF: Assume $\Gamma \vdash \mathtt{case}\ x = e\ \mathtt{of}\ \{\mathtt{True} \to e_1; \mathtt{False} \to e_2\} \rightsquigarrow \mathtt{if}\ r\ \mathtt{then}\ r_1\ \mathtt{else}\ r_2$ Since $\Gamma \vdash \mathtt{case}\ x = e\ \mathtt{of}\ \{\mathtt{True} \to e_1; \mathtt{False} \to e_2\} : \tau$, then $\Gamma \vdash e : \mathtt{Bool}$, $\Gamma \vdash e_1 : \tau$, and $\Gamma \vdash e_2 : \tau$. By inversion and IH, $\Delta_0, (\!|\Gamma|\!) \vdash_S r{:}\mathtt{Bool}$, $\Delta_0, (\!|\Gamma|\!) \vdash_S r_1{:}(\!|\tau|\!)$, and $\Delta_0, (\!|\Gamma|\!) \vdash_S r_2{:}(\!|\tau|\!)$. Thus, $\Delta_0, (\!|\Gamma|\!) \vdash_S \mathtt{if}\ r\ \mathtt{then}\ r_1\ \mathtt{else}\ r_2{:}(\!|\tau|\!)$.
- T-CASE: Assume $\Gamma \vdash \mathtt{case}\ x = e\ \mathtt{of}\ \{D_i\ \overline{y_i} \to e_i\} \rightsquigarrow \mathtt{if}\ \mathtt{is}_{D_1}\ r\ \mathtt{then}\ r_1\ \mathtt{else}\ \ldots\ \mathtt{else}\ r_n$ and $\Gamma \vdash \mathtt{case}\ x = e\ \mathtt{of}\ \{D_i\ \overline{y_i} \to e_i\} : \tau$. By inversion we get $\Gamma \vdash e \rightsquigarrow r$ and $\Gamma \vdash e_i[\overline{y_i} \mapsto \overline{\mathtt{sel}_{D_i}\ x}][x \mapsto e] \rightsquigarrow r_i$. By IH and the Definition 2 on the checkers and selectors, we get $\Delta_0, (\!|\Gamma|\!) \vdash_S \mathtt{if}\ \mathtt{is}_{D_1}\ r\ \mathtt{then}\ r_1\ \mathtt{else}\ \ldots\ \mathtt{else}\ r_n{:}(\!|\tau|\!)$.

□

Theorem 8. *If $\Gamma \vdash e \rightsquigarrow r$, then for every substitution $\theta \in [\![\Gamma]\!]$ and every model $\sigma \in [\![\theta^\perp]\!]$, if $\theta^\perp \cdot e \hookrightarrow^\star v$ then $\sigma^\beta \models r = [\![v]\!]$.*

Proof. We proceed using the notion of tracking substitutions from Figure 8 of [55]. Since $\theta^\perp \cdot e \hookrightarrow^\star v$, there exists a sequence of evaluations via tracked substitutions,

$$\langle \theta_1^\perp; e_1 \rangle \hookrightarrow \ldots \langle \theta_i^\perp; e_i \rangle \ldots \hookrightarrow \langle \theta_n^\perp; e_n \rangle$$

with $\theta_1^\perp \equiv \theta^\perp$, $e_1 \equiv e$, and $e_n \equiv v$. Moreover, each e_{i+1} is well formed under Γ, thus it has a translation $\Gamma \vdash e_{i+1} \rightsquigarrow r_{i+1}$. Thus we can iteratively apply Lemma 5 $n-1$ times and since v is a value the extra variables in $\theta_n^\perp$ are irrelevant, thus we get the required $\sigma^\beta \models r = [\![v]\!]$. □

For Boolean expressions we specialize the above to

Corollary 2. *If $\Gamma \vdash e : \mathtt{Bool}^\downarrow$ and $\Gamma \vdash e \rightsquigarrow r$, then for every substitution $\theta \in [\![\Gamma]\!]$ and every model $\sigma \in [\![\theta^\perp]\!]$, $\theta^\perp \cdot e \hookrightarrow^\star \mathtt{True} \iff \sigma^\beta \models r$*

Proof. We prove the left and right implication separately:

- $\Rightarrow$ By direct application of Theorem 8 for $v \equiv \mathtt{True}$.
- $\Leftarrow$ Since e is terminating, $\theta^\perp \cdot e \hookrightarrow^\star v$. with either $v \equiv \mathtt{True}$ or $v \equiv \mathtt{False}$. Assume $v \equiv \mathtt{False}$, then by Theorem 8, $\sigma^\beta \models \neg r$, which is a contradiction. Thus, $v \equiv \mathtt{True}$.

□

Lemma 5 (Equivalence Preservation)**.** *If $\Gamma \vdash e \rightsquigarrow r$, then for every substitution $\theta \in [\![\Gamma]\!]$ and every model $\sigma \in [\![\theta^\perp]\!]$, if $\langle \theta^\perp; e \rangle \hookrightarrow \langle \theta_2^\perp; e_2 \rangle$ and for $\Gamma \subseteq \Gamma_2$ so that $\theta_2^\perp \in [\![\Gamma_2]\!]$ and $\sigma_2^\beta \in [\![\theta_2^\perp]\!]$, $\Gamma_2 \vdash e_2 \rightsquigarrow r_2$ then $\sigma^\beta \cup (\sigma_2^\beta \setminus \sigma^\beta) \models r = r_2$.*

Proof. We proceed by case analysis on the derivation $\langle \theta^\perp; e \rangle \hookrightarrow \langle \theta_2^\perp; e_2 \rangle$.

- Assume $\langle \theta^\perp; e_1\ e_2 \rangle \hookrightarrow \langle \theta_2^\perp; e_1'\ e_2 \rangle$. By inversion $\langle \theta^\perp; e_1 \rangle \hookrightarrow \langle \theta_2^\perp; e_1' \rangle$. Assume $\Gamma \vdash e_1 \rightsquigarrow r_1$, $\Gamma \vdash e_2 \rightsquigarrow r_2$, $\Gamma_2 \vdash e_1' \rightsquigarrow r_1'$. By IH $\sigma^\beta \cup (\sigma_2^\beta \setminus \sigma^\beta) \models r_1 = r_1'$, thus $\sigma^\beta \cup (\sigma_2^\beta \setminus \sigma^\beta) \models \mathsf{app}\ r_1\ r_2 = \mathsf{app}\ r_1'\ r_2$.
- Assume $\langle \theta^\perp; c\ e \rangle \hookrightarrow \langle \theta_2^\perp; c\ e' \rangle$. By inversion $\langle \theta^\perp; e \rangle \hookrightarrow \langle \theta_2^\perp; e' \rangle$. Assume $\Gamma \vdash e \rightsquigarrow r$, $\Gamma \vdash e' \rightsquigarrow r'$. By IH $\sigma^\beta \cup (\sigma_2^\beta \setminus \sigma^\beta) \models r = r'$, thus $\sigma^\beta \cup (\sigma_2^\beta \setminus \sigma^\beta) \models \mathsf{app}\ c\ r = \mathsf{app}\ c\ r'$.
- Assume $\langle \theta^\perp; \mathtt{case}\ x = e\ \mathtt{of}\ \{D_i\ \overline{y_i} \to e_i\} \rangle \hookrightarrow \langle \theta_2^\perp; \mathtt{case}\ x = e'\ \mathtt{of}\ \{D_i\ \overline{y_i} \to e_i\} \rangle$. By inversion $\langle \theta^\perp; e \rangle \hookrightarrow \langle \theta_2^\perp; e' \rangle$. Assume $\Gamma \vdash e \rightsquigarrow r$, $\Gamma \vdash e' \rightsquigarrow r'$. $\Gamma \vdash e_i[\overline{y_i} \mapsto \overline{\mathtt{sel}_{D_i}\ x}][x \mapsto e] \rightsquigarrow r_i$. By IH $\sigma^\beta \cup (\sigma_2^\beta \setminus \sigma^\beta) \models r = r'$, thus $\sigma^\beta \cup (\sigma_2^\beta \setminus \sigma^\beta) \models \mathtt{if}\ \mathtt{is}_{D_1}\ r\ \mathtt{then}\ r_1\ \mathtt{else}\ \ldots\ \mathtt{else}\ r_n(\!|\tau|\!) = \mathtt{if}\ \mathtt{is}_{D_1}\ r'\ \mathtt{then}\ r_1\ \mathtt{else}\ \ldots\ \mathtt{else}\ r_n(\!|\tau|\!)$.
- Assume $\langle \theta^\perp; D\ \overline{e_i}\ e\ \overline{e_j} \rangle \hookrightarrow \langle \theta_2^\perp; D\ \overline{e_i}\ e'\ \overline{e_j} \rangle$. By inversion $\langle \theta^\perp; e \rangle \hookrightarrow \langle \theta_2^\perp; e' \rangle$. Assume $\Gamma \vdash e \rightsquigarrow r$, $\Gamma \vdash e_i \rightsquigarrow r_i$, $\Gamma \vdash e' \rightsquigarrow r'$. By IH $\sigma^\beta \cup (\sigma_2^\beta \setminus \sigma^\beta) \models r = r'$, thus $\sigma^\beta \cup (\sigma_2^\beta \setminus \sigma^\beta) \models \mathsf{app}\ D\ \overline{r_i}\ r\ \overline{r_j} = \mathsf{app}\ D\ \overline{r_i}\ r'\ \overline{r_j}$.
- Assume $\langle \theta^\perp; c\ w \rangle \hookrightarrow \langle \theta^\perp; \delta(c, w) \rangle$. By the definition of the syntax, $c\ w$ is a fully applied logical operator, thus $\sigma^\beta \cup (\sigma_2^\beta \setminus \sigma^\beta) \models c\ w = [\![\delta(c, w)]\!]$

- Assume $\langle \theta^\bot; (\lambda x.e) e_x \rangle \hookrightarrow \langle \theta^\bot; e[x \mapsto e_x] \rangle$. Assume $\Gamma \vdash e \rightsquigarrow r$, $\Gamma \vdash e_x \rightsquigarrow r_x$. Since σ^β is defined to satisfy the β-reduction axiom, $\sigma^\beta \cup (\sigma_2^\beta \setminus \sigma^\beta) \models \texttt{app}\ (\texttt{lam}\ x\ e)\ r_x = r[x \mapsto r_x]$.
- Assume $\langle \theta^\bot; \texttt{case}\ x = D_j\ \overline{e}\ \texttt{of}\ \{D_i\ \overline{y_i} \to e_i\} \rangle \hookrightarrow \langle \theta^\bot; e_j[x \mapsto D_j\ \overline{e}][y_i \mapsto \overline{e}] \rangle$. Also, let $\Gamma \vdash e \rightsquigarrow r$, $\Gamma \vdash e_i[x \mapsto D_j\ \overline{e}][y_i \mapsto \overline{e_i}] \rightsquigarrow r_i$. By the axiomatic behavior of the measure selector $\texttt{is}_{D_j}\ \overline{r}$, we get $\sigma^\beta \models \texttt{is}_{D_j}\ \overline{r}$. Thus, $\sigma^\beta \texttt{if}\ \texttt{is}_{D_1}\ r\ \texttt{then}\ r_1\ \texttt{else}\ \ldots\ \texttt{else}\ r_n = r_j$.
- Assume $\langle (x, e_x)\theta^\bot; x \rangle \hookrightarrow \langle (x, e'_x)\theta_2^\bot; x \rangle$. By inversion $\langle \theta^\bot; e_x \rangle \hookrightarrow \langle \theta_2^\bot; e'_x \rangle$. By identity of equality, $(x, r_x)\sigma^\beta \cup (\sigma_2^\beta \setminus \sigma^\beta) \models x = x$.
- Assume $\langle (y, e_y)\theta^\bot; x \rangle \hookrightarrow \langle (y, e_y)\theta_2^\bot; e_x \rangle$. By inversion $\langle \theta^\bot; x \rangle \hookrightarrow \langle \theta_2^\bot; e_x \rangle$. Assume $\Gamma \vdash e_x \rightsquigarrow r_x$. By IH $\sigma^\beta \cup (\sigma_2^\beta \setminus \sigma^\beta) \models x = r_x$. Thus $(y, r_y)\sigma^\beta \cup (\sigma_2^\beta \setminus \sigma^\beta) \models x = r_x$.
- Assume $\langle (x, w)\theta^\bot; x \rangle \hookrightarrow \langle (x, w)\theta^\bot; w \rangle$. Thus $(x, [\![w]\!])\sigma^\beta \models x = [\![w]\!]$.
- Assume $\langle (x, D\ \overline{y})\theta^\bot; x \rangle \hookrightarrow \langle (x, D\ \overline{y})\theta^\bot; D\ \overline{y} \rangle$. Thus $(x, \texttt{app}\ D\ \overline{y})\sigma^\beta \models x = \texttt{app}\ D\ \overline{y}$.
- Assume $\langle (x, D\ \overline{e})\theta^\bot; x \rangle \hookrightarrow \langle (x, D\ \overline{y}), \overline{(y_i, e_i)}\theta^\bot; D\ \overline{y} \rangle$. Assume $\Gamma \vdash e_i \rightsquigarrow r_i$. Thus $(x, \texttt{app}\ D\ \overline{y}), \overline{(y_i, r_i)}\sigma^\beta \models x = \texttt{app}\ D\ \overline{y}$.

□

E.2 Soundness of Approximation

Theorem 9 (Soundness of Algorithmic)**.** *If* $\Gamma \vdash_S e : \tau$ *then* $\Gamma \vdash e : \tau$.

Proof. To prove soundness it suffices to prove that subtyping is appropriately approximated, as stated by the following lemma. □

Lemma 6. *If* $\Gamma \vdash_S \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}$ *then* $\Gamma \vdash \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}$.

Proof. By rule $\preceq$-BASE, we need to show that $\forall \theta \in [\![\Gamma]\!].[\![\theta \cdot \{v : B \mid e_1\}]\!] \subseteq [\![\theta \cdot \{v : B \mid e_2\}]\!]$. We fix a $\theta \in [\![\Gamma]\!]$. and get that forall bindings $(x_i : \{v : B^\downarrow \mid e_i\}) \in \Gamma$, $\theta \cdot e_i[v \mapsto x_i] \hookrightarrow^\star \texttt{True}$.

Then need to show that for each e, if $e \in [\![\theta \cdot \{v : B \mid e_1\}]\!]$, then $e \in [\![\theta \cdot \{v : B \mid e_2\}]\!]$.

If e diverges then the statement trivially holds. Assume $e \hookrightarrow^\star w$. We need to show that if $\theta \cdot e_1[v \mapsto w] \hookrightarrow^\star \texttt{True}$ then $\theta \cdot e_2[v \mapsto w] \hookrightarrow^\star \texttt{True}$.

Let $\theta^\bot$ the lifted substitution that satisfies the above. Then by Lemma 2 for each model $\sigma^\beta \in [\![\theta^\bot]\!]$, $\sigma^\beta \models r_i$, and $\sigma^\beta \models q_1$ for $\Gamma \vdash e_i[v \mapsto x_i] \rightsquigarrow r_i$ $\Gamma \vdash e_i[v \mapsto w] \rightsquigarrow q_i$. Since $\Gamma \vdash_S \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}$ we get

$$\bigwedge_i r_i \Rightarrow q_1 \Rightarrow q_2$$

thus $\sigma^\beta \models q_2$. By Theorem 2 we get $\theta \cdot e_2[v \mapsto w] \hookrightarrow^\star \texttt{True}$. □

F. 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 § F.1, and the latter by introducing a combinator for safely asserting extensional equality § F.2. In the rest of this section, for clarity we omit app when it is clear from the context.

F.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 7). 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 $\texttt{lam}\ x_i\ e$, the index i is greater than any lambda argument appearing in e.

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

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

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`:

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

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

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

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

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.

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

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:

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

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

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

F.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.

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}
```