Refinement Reflection:
Parallel Legacy Languages as Theorem Provers

ANONYMOUS AUTHOR(S)

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

1 INTRODUCTION

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

Previously, SMT-based refinement types (Constable and Smith 1987; Rushby et al. 1998) have been used to retrofit so-called "shallow" verification *e.g.* array bounds checking into ML (Bengtson et al. 2008; Rondon et al. 2008; Xi and Pfenning 1998), C (Condit et al. 2007; Rondon et al. 2010), Haskell (Vazou et al. 2014), TypeScript (Vekris et al. 2016), and Racket (Kent et al. 2016). 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 permits 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

 http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn.

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 (Vazou et al. 2014), thereby converting the legacy language Haskell into a theorem prover. We demonstrate the benefits of this conversion by proving typeclass laws. Haskell's typeclass machinery has led to a suite of expressive abstractions and optimizations which, for correctness, crucially require typeclass *instances* to obey key algebraic laws. We show how reflection can be used to verify that widely used instances of the Monoid, Applicative, Functor, and Monad typeclasses satisfy the respective laws, making the use of these typeclasses safe (§ 6).

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 (§ 7).

2 OVERVIEW

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

2.1 Refinement Types

First, we recall some preliminaries about specification and verification with refinement types.

Refinement types are the source program's (here Haskell's) types decorated with logical predicates drawn from an SMT decidable logic (Constable and Smith 1987; Rushby et al. 1998). For example, we define the `Nat` type as `Int` refined by the predicate `0 ≤ v` which belongs to the quantifier free logic of linear arithmetic and uninterpreted functions (QF-UFLIA (Barrett et al. 2010)).

```
type Nat = { v:Int | 0 ≤ v }
```

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

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

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

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

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

```
type Prop = ()
```

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

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

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

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

Function types encode universally quantified propositions:

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

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

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

```
data QED = QED

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

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

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

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

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

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

A Note on Bottom: Readers familiar with Haskell's semantics may be concerned that "bottom", which inhabits all types, makes our proofs suspect. Fortunately, as described in Vazou et al. (2014), Liquid Haskell ensures that all terms with non-trivial refinements provably terminate and evaluate to (non-bottom) values, which makes our proofs sound.

2.2 Refinement Reflection

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

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

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

Step 1: Definition The annotation tells Liquid Haskell to declare an *uninterpreted function* `fib :: Int → Int` in the refinement logic. By uninterpreted, we mean that the logical fib is *not* connected to the program function fib; in the logic, fib only satisfies the *congruence axiom* $\forall n, m.\ n = m \Rightarrow \texttt{fib}\ n = \texttt{fib}\ m$. 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 VC that is discharged by the SMT solver, even though `fib` is uninterpreted:

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

Note that the verification of `pf_fib2` relies merely on the fact that `fib` was applied to (*i.e.* unfolded at) `0`, `1` and `2`. The SMT solver automatically *combines* the facts, once they are in the antecedent. The following is also verified:

```
pf_fib2' :: { fib 2 = 1 }
pf_fib2' = [ fib 0, fib 1, fib 2 ] ** QED
```

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

Reflection vs. Axiomatization An alternative *axiomatic* approach, used by Dafny (Leino 2010) and F* (Swamy et al. 2016), is to encode `fib` using a universally quantified SMT formula (or axiom): $\forall n.$ `fibP` (`fib` n) n. Axiomatization offers greater automation than reflection. Unlike Liquid Haskell, 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 axioms render checking of the VCs *undecidable*. In practice, automatic axiom instantation can easily lead to infinite "matching loops". For example, the existence of a term `fib` n in a VC can trigger the above axiom, which may then produce the terms `fib` $(n-1)$ and `fib` $(n-2)$, which may then recursively give rise to further instantiations *ad infinitum*. To prevent matching loops an expert must carefully craft "triggers" and provide a "fuel" parameter (Amin et al. 2014) 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 (Leino and Pit-Claudel 2016).

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 (Mu et al. 2009).

"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 typeclass 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.

Reflection and Refinement Types The above examples illustrate how refinement types critically allow us to restrict the domain of both reflected functions and proof terms in order to enable sound theorem proving.

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

Thus, refinement types let us encode proofs that only hold on refined types, like `Nat` while using Haskell's Integers, instead of forcing to switch to a user or library defined `Data.Natural` Haskell type. As a less elegant alternative, usually followed in dependently typed languages, `fib` could be defined as a function returning a `Maybe` – *i.e.* returning `Nothing` on negative inputs. Then the code and proofs would be littered with cases analyses required to extract `fib`'s result. Another alternative is Dafny's approach that uses code level assumptions and assertions to restrict function's domain and range. While these assumptions and assertions can be semi-automatically discharged using the SMT solver, they impose programmer overheads as they cannot be inferred unlike refinement types that are inferred using the abstract interpretation framework of liquid typing (Rondon et al. 2008).

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 (Bocchino et al. 2009)), Concurrent Revisions for .NET (Burckhardt et al. 2010) and Haskell's LVish (Kuper et al. 2014), Accelerate (McDonell et al. 2013), and REPA (Keller et al. 2010). 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 ()
```

which returns an (effectful) `Par` computation that 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. However, the interface has an `Ord` constraint, which means this polymorphic function works with user-defined data types and thus, orderings. What if the user fails to implement a total order? Then, even a correct implementation of, *e.g.* a concurrent skiplist (Fomitchev and Ruppert 2004), can reveal different insertion orders due to concurrency, thus wrecking the determinism guarantee.

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. Fortunately, 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 }
```

Verified Monoids Similarly, we can refine the `Monoid` typeclass with refinements expressing the `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` and `Double` are trivial to write; they just appeal to the SMT solver's built-in theories. For example, the following proves totality (of ordering) for `Int`.

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

Verified instances for algebraic datatypes We use structural induction to prove the class laws for user defined algebraic datatypes. For example, we can inductively define `Peano` numerals and then write a function to compare two `Peano` numbers:

```
data Peano = Z | S Peano

reflect leq :: Peano → Peano → Bool
leq Z _         = True
leq (S _) 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 (Vazou et al. 2014), 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
                       ∵ 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 (§ 3). We can write the rest of the `VerifiedOrd` proof methods in a fashion similar to `totalPeano` and use them to create verified instances:

```
instance Ord Peano where              instance VerifiedOrd Peano where
  (≤) = leq                            total = totalPeano
```

Proving all the four `VerifiedOrd` laws is a burden on the programmer. Since `Peano` is isomorphic to `Nat`s, 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 (Barthe and Pons 2001). We design a typeclass `Iso` to witnesses type isomorphism.

```
class Iso a b where
  to     :: a → b
  from   :: b → a
  toFrom :: x:a → {to (from x) = x}
  fromTo :: 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 b using the VerifiedOrd totality on a as:

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

Now, we can simply use Haskell's instances, to obtain a VerifiedOrd instance for Peano by creating an Iso Nat instance for 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, totality:

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

Haskell's typeclass machinery will now let us obtain a VerifiedOrd instance for (Peano, Peano) from the VerifiedOrd instance for Peano. 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 GHC's support for generics (Magalhães et al. 2010) to fully automate the derivation of verified instances for user datatypes. In §7, we use these ideas to develop safe interfaces to LVish modules and to verify patterns from DPJ.

3 REFINEMENT REFLECTION: λ^R

We formalize refinement reflection in two steps. First, we develop a core calculus λ^R with an *undecidable* type system based on denotational semantics. We show how the soundness of the type system allows us to *prove theorems* using λ^R. Next, in § 4 we define a language λ^S that soundly approximates λ^R while enabling decidable SMT-based type checking.

3.1 Syntax

Figure 1 summarizes the syntax of λ^R, which is essentially the calculus λ^U (Vazou et al. 2014) with explicit recursion and a special `reflect` binding to denote terms that are reflected into the refinement logic. The elements of λ^R are layered into primitive constants, values, expressions, binders and programs.

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

Fig. 1. **Syntax of λ^R**

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

Fig. 2. **Syntax of λ^S**

Constants The primitive constants of λ^R include primitive relations $\odot$, 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, variables x, applications $e\ e$, and `case` expressions.

Binders & Programs A *binder* b is a series of possibly recursive `let` definitions, followed by an expression. A *program* p is a series of `reflect` definitions, each of which names a function that is reflected into the refinement logic, followed by a binder. The stratification of programs via binders is required so that arbitrary recursive definitions are allowed in the program but cannot be inserted into the logic via refinements or reflection. (We *can* allow non-recursive `let` binders in expressions e, but omit them for simplicity.)

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 as recursive `let` bindings, with extra termination-check constraints imposed by the type system:

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

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

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

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

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. As discussed by Vazou et al. (2014) termination labels can be checked using refinement types and are used to ensure that refinements cannot diverge as required for soundness of type checking under lazy evaluation. Finally, we have dependent *function types* $x : \tau_x \to \tau$ 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.

Denotations Each type τ *denotes* a set of expressions $[\![\tau]\!]$, that is defined via the operational semantics (Knowles and Flanagan 2010). Let $\lfloor\tau\rfloor$ be the type we get if we erase all refinements from τ and $e : \lfloor\tau\rfloor$ be the standard typing relation for the typed lambda calculus. Then, we define the denotation of types as:

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

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

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

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

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

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

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

Termination Checking We defined $\mathsf{Reflect}(\cdot, \cdot)$ to be a *partial* function that only reflects provably terminating expressions, *i.e.* expressions whose result type is marked with $\Downarrow$. If a non-provably terminating function is reflected in an λ^R expression then type checking will fail (with a reflection type error in the implementation). This restriction is crucial for soundness, as diverging expressions can lead to inconsistencies. For example, reflecting the diverging `f x = 1 + f x` into the logic leads to an inconsistent system that is able to prove `0 = 1`.

Automatic Reflection Reflection of λ^R expressions into the refinements happens automatically by the type system, not manually by the user. The user simply annotates a function f as `reflect` f. Then, the rule T-Refl in Figure 3 is used to type check the reflected function by strengthening the f's result via $\mathsf{Reflect}(\cdot, \cdot)$. Finally, the rule T-Let is used to check that the automatically strengthened type of f satisfies f's implementation.

Consequences for Verification Reflection has a major consequence for verification: instead of being tethered to quantifier instantiation heuristics or having to program "triggers" as in Dafny (Leino 2010) or F* (Swamy et al.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 3. Typing of λ^R and algorithmic subtyping of λ^S

2016), the programmer can predictably "unfold" the definition of the function during a proof simply by "calling" the function, which, as discussed in § 6, we have found to be a very natural way of structuring proofs.

3.5 Typing Rules

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

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

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

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 (Knowles and Flanagan 2010; Vazou et al. 2014). First, rule T-Refl is used to strengthen the type of each reflected binder with its definition, as described previously in § 3.4. Second, rule T-Exact strengthens the expression with a singleton type equating the value and the expression (*i.e.* reflecting the expression in the type). This is a generalization of the "selfification" rules from (Knowles and Flanagan 2010; Ou et al. 2004) and is required to equate the reflected functions with their definitions. For example, the application `fib 1` is typed as $\{v : \texttt{Int}^{\Downarrow} \mid \texttt{fibP}\ v\ 1 \wedge v = \texttt{fib}\ \ 1\}$ where the first conjunct comes from the (reflection-strengthened) output refinement of `fib` § 2 and the second comes from rule T-Exact.

Well-formedness A judgment $\Gamma \vdash \tau$ states that the refinement type τ is well-formed in the environment Γ. Following Vazou et al. (2014), τ is well-formed if all the refinements in τ are `Bool`-typed, provably terminating expressions in Γ.

Subtyping A judgment $\Gamma \vdash \tau_1 \preceq \tau_2$ states that the type τ_1 is a subtype of τ_2 in the environment Γ. Informally, τ_1 is a subtype of τ_2 if, when the free variables of τ_1 and τ_2 are bound to expressions described by Γ, the denotation of τ_1 is *contained in* the denotation of τ_2. Subtyping of basic types reduces to denotational containment checking, shown in rule $\preceq$-Base-λ^R. That is, τ_1 is a subtype of τ_2 under Γ if for any closing substitution θ in the denotation of Γ, $[\![\theta \cdot \tau_1]\!]$ is contained in $[\![\theta \cdot \tau_2]\!]$.

Soundness Following λ^U (Vazou et al. 2014), in Supplementary-Material (2017) we prove that evaluation preserves typing and typing implies denotational inclusion.

Theorem 3.1. *[Soundness of λ^R]*

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

Theorem 3.1 lets us interpret well typed programs as proofs of propositions. For example, in § 2 we verified that the term `fibUp` proves

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

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

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

4 ALGORITHMIC CHECKING: λ^S

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

Syntax: Terms & Sorts Figure 2 summarizes the syntax of λ^S, the *sorted* (SMT-) decidable logic of quantifier-free equality, uninterpreted functions and linear arithmetic (QF-EUFLIA) (Barrett et al. 2010; Nelson 1981). The *terms* of λ^S include integers n, booleans b, variables x, data constructors D (encoded as constants), fully applied unary $\oplus_1$ and binary $\oplus_2$ operators, and application $x\ \overline{r}$ 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 $s \rightarrow s$. Other functional values in λ^R, *e.g.* reflected λ^R functions and λ-expressions, are represented as first-order values with the uninterpreted sort `Fun` s s. The universal sort `U` represents all other values.

Semantics: Satisfaction & Validity An assignment σ maps 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 $\sigma \cdot r$ "is true" (Nelson 1981). A predicate r *is satisfiable* if there exists $\sigma \models r$. A predicate r *is valid* if for all assignments $\sigma \models r$.

4.1 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. Most of the transformation rules are identity and can be found in (Supplementary-Material 2017). Here we discuss the non-identity ones.

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

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

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

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

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

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

Embedding Applications We embed applications via defunctionalization (Reynolds 1972) using the uninterpreted `app`:

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

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

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

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

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

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

To generalize, let D_i be a data constructor such that

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

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 $\texttt{Fun}\ (\!|\tau|\!)\ (\!|\tau_{i,j}|\!)$.

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

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

For example, the body of the list append function

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

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

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

Informally, the translation relation $\Gamma \vdash e \leadsto r$ 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 4.1 (β-Model). A β−model σ^β is an extension of a model σ where `lam` and `app` satisfy the axioms of β-equivalence:

$$\forall\, x\ y\ e.\ \texttt{lam}\ x\ e \ =\ \texttt{lam}\ y\ (e[x \mapsto y]) \qquad\qquad \forall\, x\ e_x\ e.\ \texttt{app}\ (\texttt{lam}\ x\ e)\ e_x \ =\ e[x \mapsto e_x]$$

Semantics Preservation We define the translation of a λ^R term into λ^S under the empty environment as $(\!|e|\!) \doteq r$ *iff* $\emptyset \vdash e \leadsto r$. 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 (Vazou et al. 2014). We connect the semantics of λ^R and λ^S via the following:

Theorem 4.2. *If $\Gamma \vdash e \leadsto r$, 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 4.3. *If $\Gamma \vdash e :$ `Bool`, e reduces to a value and $\Gamma \vdash e \leadsto r$, then for every $\theta \in [\![\Gamma]\!]$ and every $\sigma \in \theta^\perp$, $\theta^\perp \cdot e \hookrightarrow^\star$ `True` iff $\sigma^\beta \models r$.*

4.3 Decidable Type Checking

We make the type checking from Figure 3 decidable by checking subtyping via an SMT solver.

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 (Vazou et al. 2014):

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

Subtyping via Verification Conditions We make subtyping, and hence, typing decidable, by replacing the denotational base subtyping rule $\preceq$-Base-λ^R with the conservative, algorithmic version $\preceq$-Base-λ^S of Figure 3 that uses an SMT solver to check the validity of the subtyping VC. We use Corollary 4.3 to prove soundness of subtyping.

Lemma 4.4. *If $\Gamma \vdash_S \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}$ then $\Gamma \vdash \{v : B \mid e_1\} \preceq \{v : B \mid e_2\}$.*

Soundness of λ^S We write $\Gamma \vdash_S e : \tau$ for the judgments that can be derived using the algorithmic subtyping rule $\preceq$-Base-λ^S instead of the denotational rule $\preceq$-Base-λ^R. Lemma 4.4 implies the soundness of λ^S.

Theorem 4.5 (Soundness of λ^S). *If $\Gamma \vdash_S e : \tau$ then $\Gamma \vdash e : \tau$.*

Note that we *do not* require the β-equivalence axioms for soundness: if a VC is valid *without* the β-equivalence axioms, *i.e.* when `lam` and `app` are uninterpreted, then the VC is trivially valid *with* the axioms, and hence, by Theorem 4.5 the program is safe.

5 REASONING ABOUT LAMBDAS

Encoding of λ-abstractions and applications via uninterpreted functions, while sound, is *imprecise* as it makes it hard to prove theorems that require α- and β-equivalence or extensional equality. Using the universally quantified α- and β- equivalence axioms would let the type checker accept more programs, but would render validity, and hence, type checking undecidable. Next, we identify a middle ground by describing an not provably complete, but sound and decidable approach to increase the precision of type checking by strengthening the VCs with instances of the α- and β- equivalence axioms § 5.1 and by introducing a combinator for safely asserting extensional equality § 5.2. In the sequel, we omit `app` when it is clear from the context.

5.1 Equivalence

As soundness relies on satisfiability under a σ^{β} (see Definition 4.1), we can safely *instantiate* the axioms of α- and β-equivalence on any set of terms of our choosing and still preserve soundness (Theorem 4.5). 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, this VC is needed to prove the applicative laws for `Reader`:

$$d = \texttt{lam}\ x_1\ (x\ x_1) \quad \Rightarrow \quad \texttt{lam}\ x_2\ ((\texttt{lam}\ x_1\ (x\ x_1))\ x_2) = \texttt{lam}\ x_1\ (d\ x_1)$$

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 a *β-equivalence instance predicate* (or *β-instance*):

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

The λ-free restriction is 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.

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

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:

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

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

$$\texttt{lam}\ x_2\ (\texttt{lam}\ x_1\ w)\ =_\alpha\ \texttt{lam}\ x_3\ (\texttt{lam}\ x_1\ w)\ =_\beta\ \texttt{lam}\ x_3\ (\texttt{app}\ (\texttt{lam}\ x_2\ (\texttt{lam}\ x_1\ w))\ w)$$

5.2 Extensionality

Often, we need to prove that two functions are equal, given the definitions of reflected binders. 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}
```

6 EVALUATION

We have implemented refinement reflection in Liquid Haskell. In this section, we evaluate our approach by using Liquid Haskell to verify a variety of deep specifications of Haskell functions drawn from the literature and categorized in Figure 4, totalling about 4,000 lines of specifications and proofs. Verification is fast: from 1 to 5s per file (module), which is about twice as much time as GHC takes to compile the corresponding module. Overall there is about one line of type specification (theorems) for three lines of code (implementations and proof). Next, we detail each of the five classes of specifications, illustrate how they were verified using refinement reflection, and discuss the strengths and weaknesses of our approach. *All* of these proofs require refinement reflection, *i.e.* are beyond the scope of shallow refinement typing.

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

- ***Un/folding*** definitions of a function `f` at arguments `e1...en`, which due to refinement reflection, happens whenever the term `f e1 ... en` appears in a proof. For exposition, we render the function whose un/folding is relevant as **`f`**;

CATEGORY		LOC
I. Arithmetic		
Fibonacci	§ 2	48
Ackermann	(Tourlakis 2008)	280
II. Algebraic Data Types		
Fold Universal	(Mu et al. 2009)	105
III. Typeclasses	Fig 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	(Casinghino et al. 2014)	133
Unification	(Sjöberg and Weirich 2015)	200
V. Deterministic Parallelism		
Conc. Sets	§ 7.1	906
n-body	§ 7.2	930
Par. Reducers	§ 7.3	55
TOTAL		4155

Fig. 4. **Summary of Case Studies**

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

Fig. 5. **Summary of Verified Typeclass Laws**

- ***Lemma Application*** which is carried out by using the "because" combinator (∵) to instantiate some fact at some inputs;
- ***SMT Reasoning*** in particular, *arithmetic*, *ordering* and *congruence closure* which kicks in automatically (and predictably!), allowing us to simplify proofs by not having to specify, *e.g.* which subterms to rewrite.

6.1 Arithmetic Properties

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

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

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

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

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

6.2 Algebraic Data Properties

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

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

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

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

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

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

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

6.3 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 ◇ (`mappend`) and an *identity* element `mempty`. We prove that `Peano` (with a suitable `add` and `Z`), `Maybe` (with a suitable `mappend` and `Nothing`), and `List` (with append ++ and `[]`) satisfy the monoid laws.

Functor Laws A type is a functor if it has a function `fmap` that satisfies the *identity* and *distribution* (or fusion) laws in Figure 5. For example, consider the proof of the `map` distribution law for the lists, also known as "map-fusion", which is the basis for important optimizations in GHC (Wiki 2008). We reflect the definition of `map` for lists and use it to specify fusion and verify it by an inductive proof:

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

Monad Laws The monad laws, which relate the properties of the two operators ≫= and `return` (Figure 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
assoc [] f g     =  [] >>= f >>= g
                 =. [] >>= g
                 =. []
                 =. [] >>= (\x → f x >>= g) ** QED
assoc (x:xs) f g =  (x:xs) >>= f  >>= g
                 =. (f x ++ xs >>= f) >>= g
                 =. (f x >>= g) ++ (xs >>= f >>= g) ∵ bind_append (f x) (xs >>= f) g
                 =. (f x >>= g) ++ (xs >>= \y → f y >>= g) ∵ assoc xs f g
                 =. (\y → f y >>= g) x ++ (xs >>= \y → f y >>= g) ∵ βeq f g x
                 =. (x:xs) >>= (\y → f y >>= g) ** QED
```

Where the `bind_append` fusion lemma states that:

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

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

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

6.4 Functional Correctness

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

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

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

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

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

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

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

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

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

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

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

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

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

7 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. 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.

7.1 LVish: Concurrent Sets

First, we 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) as described in Kuper et al. (2014).

7.2 `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 (Marlow et al. 2011).

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 § 2.4.

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

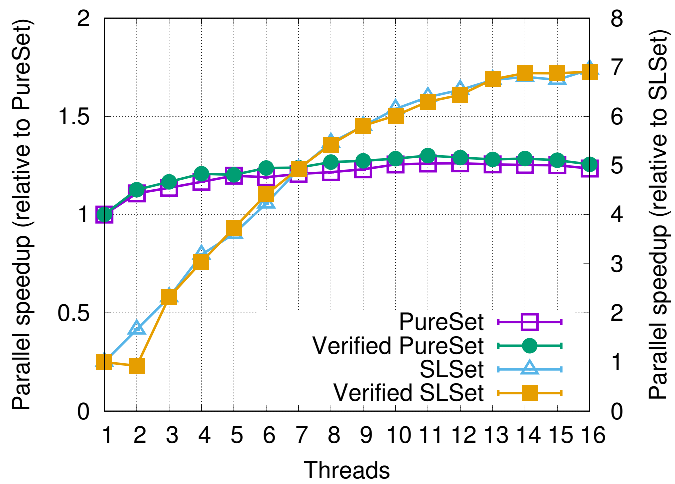

Fig. 6. Parallel speedup for doing 1 million parallel inserts over 10 iterations, verified and unverified, relative to the unverified version, for PureSet and SLSet.

To enforce the determinism of an *n*-body simulation, we need to provide a `VerifiedMonoid` instance for `Accel`. We can 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 will not be slowed down by an unacceptable amount.

7.3 DPJ: Parallel Reducers

The Deterministic Parallel Java (DPJ) project provides a deterministic-by-default semantics for the Java programming language (Bocchino et al. 2009). 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:

```
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 ⇒ VerifiedCommutativeMonoid 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 (Kuper et al. 2014) to encode a reducer variable with a value `a` and a region `s` as `RVar s a`.

[1]Floating point numbers notoriously violate associativity, but we use this approximation because Haskell does net yet have an implementation of *superaccumulators* (Collange et al. 2014).

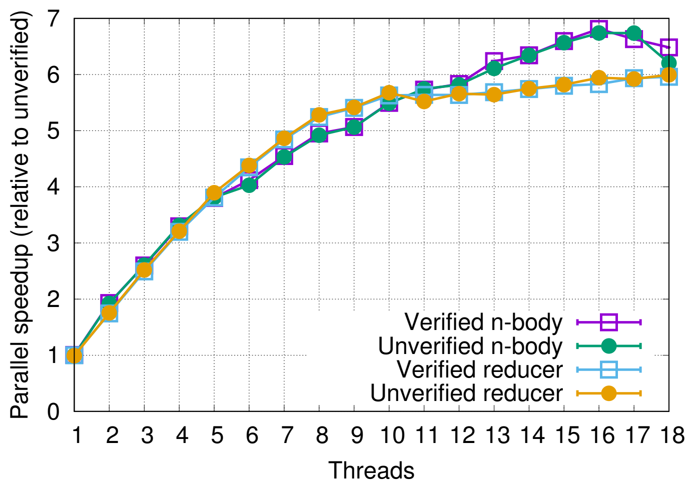

Fig. 7. Parallel speedup for doing a parallel n-body simulation and parallel array reduction. The speedup is relative to the unverified version of each respective class of program.

```
newtype RVar s a
```

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

```
updateRVar :: VerifiedCommutativeMonoid 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 $3\text{x}10^9$ 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.

8 RELATED WORK

SMT-Based Verification SMT-solvers have been extensively used to automate program verification via Floyd-Hoare logics (Nelson 1981). Our work is inspired by Dafny's Verified Calculations (Leino and Polikarpova 2016), a framework for proving theorems in Dafny (Leino 2010), but differs in (1) our use of reflection instead of axiomatization and (2) our use of refinements to compose proofs. Dafny, and the related F* (Swamy et al. 2016) which like Liquid Haskell, uses types to compose proofs, offer more automation by translating recursive functions to SMT axioms. However, unlike reflection, this axiomatic approach renders typechecking and verification undecidable (in theory) and leads to unpredictability and divergence (in practice) (Leino and Pit-Claudel 2016).

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

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

Proving Equational Properties Several authors have proposed tools for proving (equational) properties of (functional) programs. Systems Sousa and Dillig (2016) and Asada et al. (2015) extend classical safety verification algorithms, respectively based on Floyd-Hoare logic and Refinement Types, to the setting of relational or k-safety properties that are assertions over k-traces of a program. Thus, these methods can automatically prove that certain functions are associative, commutative *etc.*. but are restricted to first-order properties and are not programmer-extensible. Zeno (Sonnex et al. 2012) generates proofs by term rewriting and Halo (Vytiniotis et al. 2013) uses an axiomatic encoding to verify contracts. Both the above are automatic, but unpredictable and not programmer-extensible, hence, have been limited to far simpler properties than the ones checked here. HERMIT (Farmer et al. 2015) proves equalities by rewriting the GHC core language, guided by user specified scripts. In contrast, our proofs are simply Haskell programs, we can use SMT solvers to automate reasoning, and, most importantly, we can connect the validity of proofs with the semantics of the programs.

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

9 CONCLUSIONS AND FUTURE WORK

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

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

Second, while Haskell's separation of pure and effectful code undoubtedly makes it easier to implement refinement reflection, we believe that the technique, like refinement typing in general, is orthogonal to purity. In particular, by carefully mediating between the interaction of pure and impure code using methods like permissions, uniqueness *etc.* refinement type systems have been developed for legacy languages like C (Rondon et al. 2010), JavaScript (Chugh et al. 2012; Vekris et al. 2016), and Racket (Kent et al. 2016) and so it would be interesting to see how to extend refinement reflection to other legacy languages.

REFERENCES

N. Amin, K. R. M. L., and T. Rompf. 2014. Computing with an SMT Solver. In *TAP*.

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

K. Asada, R. Sato, and N. Kobayashi. 2015. Verifying Relational Properties of Functional Programs by First-Order Refinement. In *PEPM*.

L. Augustsson. 1998. Cayenne - a Language with Dependent Types.. In *ICFP*.

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

G. Barthe and O. Pons. 2001. Type isomorphisms and proof reuse in dependent type theory. In *International Conference on Foundations of Software Science and Computation Structures*. Springer.

J. Bengtson, K. Bhargavan, C. Fournet, A.D. Gordon, and S. Maffeis. 2008. Refinement Types for Secure Implementations. In *CSF*.

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

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

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

Jr. Bocchino, L. Robert, V. S. Adve, S. V. Adve, and M. Snir. 2009. Parallel Programming Must Be Deterministic by Default *(HotPar)*.

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

S. Burckhardt, A. Baldassin, and D. Leijen. 2010. Concurrent Programming with Revisions and Isolation Types. In *OOPSLA*.

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

R. Chugh, D. Herman, and R. Jhala. 2012. Dependent Types for JavaScript. In *OOPLSA*.

S. Collange, D. Defour, S. Graillat, and R. Iakymchuk. 2014. Full-Speed Deterministic Bit-Accurate Parallel Floating-Point Summation on Multi- and Many-Core Architectures. (2014). https://hal.archives-ouvertes.fr/hal-00949355 working paper or preprint.

J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula. 2007. Dependent Types for Low-Level Programming. In *ESOP*.

R. L. Constable and S. F. Smith. 1987. Partial Objects In Constructive Type Theory. In *LICS*.

R. A. Eisenberg and J. Stolarek. 2014. Promoting functions to type families in Haskell. In *Haskell*.

A. Farmer, N. Sculthorpe, and A. Gill. 2015. Reasoning with the HERMIT: Tool Support for Equational Reasoning on GHC Core Programs *(Haskell)*.

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

M. Fomitchev and E. Ruppert. 2004. Lock-free linked lists and skip lists. In *Principles of distributed computing*.

G. Kahn. 1974. The semantics of a simple language for parallel programming. In *Information Processing*.

G. Keller, M. M.T. Chakravarty, R. Leshchinskiy, S. Peyton-Jones, and B. Lippmeier. 2010. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *ICFP*.

A. M. Kent, D. Kempe, and S. Tobin-Hochstadt. 2016. Occurrence typing modulo theories. In *PLDI*.

K.W. Knowles and C. Flanagan. 2010. Hybrid type checking. *TOPLAS* (2010).

Lindsey Kuper, Aaron Turon, Neelakantan R Krishnaswami, and Ryan R Newton. 2014. Freeze after writing: quasi-deterministic parallel programming with LVars. In *POPL*.

E. A. Lee and D. G. Messerschmitt. 1987. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. Comput.* (1987).

K. R. M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness *(LPAR)*.

K. R. M. Leino and C. Pit-Claudel. 2016. Trigger selection strategies to stabilize program verifiers. In *CAV*.

K. R. M. Leino and N. Polikarpova. 2016. Verified Calculations. In *VSTTE*.

J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. 2010. A Generic Deriving Mechanism for Haskell. In *Haskell*.

S. Marlow, R. Newton, and S. Peyton-Jones. 2011. A Monad for Deterministic Parallelism. In *Haskell*.

S. Marlow, S. Peyton-Jones, and S. Singh. 2009. Runtime support for multicore Haskell. In *ICFP*.

T. L. McDonell, M. M.T. Chakravarty, G. Keller, and B. Lippmeier. 2013. Optimising Purely Functional GPU Programs. In *ICFP*.

S. C. Mu, H. S. Ko, and P. Jansson. 2009. Algebra of Programming in Agda: Dependent Types for Relational Program Derivation. *J. Funct. Program.* (2009).

K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. 2012. A Type System for Borrowing Permissions. In *POPL*.

G. Nelson. 1981. *Techniques for program verification.* Technical Report CSL81-10. Xerox Palo Alto Research Center.

U. Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph.D. Dissertation. Chalmers.

X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. 2004. Dynamic Typing with Dependent Types. In *IFIP TCS*.

J. C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *25th ACM National Conference*.

P. Rondon, M. Kawaguchi, and R. Jhala. 2008. Liquid Types. In *PLDI*.

P. Rondon, M. Kawaguchi, and R. Jhala. 2010. Low-Level Liquid Types. In *POPL*.

J. Rushby, S. Owre, and N. Shankar. 1998. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE TSE* (1998).

V. Sjöberg and S. Weirich. 2015. Programming Up to Congruence. *POPL* (2015).

W. Sonnex, S. Drossopoulou, and S. Eisenbach. 2012. Zeno: An Automated Prover for Properties of Recursive Data Structures. In *TACAS*.

M. Sousa and I. Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *PLDI*.

Non-Anonymous Supplementary-Material. 2017. Supplementary Material. (2017).

N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Y. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *POPL*.

L. G. Tesler and H. J. Enea. 1968. A language design for concurrent processes. In *AFIPS*.

G. Tourlakis. 2008. Ackermannfis Function. (2008). http://www.cs.yorku.ca/~gt/papers/Ackermann-function.pdf.

N. Vazou, A. Bakst, and R. Jhala. 2015. Bounded refinement types. In *ICFP*.

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

P. Vekris, B. Cosman, and R. Jhala. 2016. Refinement types for TypeScript. In *PLDI*.

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

E. Westbrook, J. Zhao, Z. Budimlić, and V. Sarkar. 2012. *Practical Permissions for Race-Free Parallelism.*

GHC Wiki. 2008. GHC Optimisations. (2008). https://wiki.haskell.org/GHC_optimisations.

H. Xi and F. Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types.. In *PLDI*.