Towards a Translation from Liquid Haskell to Coq

Lykourgos Mastorou¹, Niki Vazou¹, and Michael Greenberg²

¹ IMDEA Software Institute, Spain  ² Stevens Institute of Technology, USA

Liquid Haskell [7] is a refinement type checker for Haskell programs that can also be used as a theorem prover to mechanically check user-provided proofs. For example, it has been used to mechanize proofs about equational reasoning [6], relational properties [5], and program security [3]. These case studies demonstrate that mechanically checking theorems using Liquid Haskell is possible, but they illustrate two main disadvantages. First, the foundations of Liquid Haskell as a theorem prover have neither been well studied nor formalized. Second—and more pressingly—although Liquid Haskell can check proofs, it does not assist in their development. That is, when a fully automatic the proof fails, the proof engineer can’t directly inspect the proof at the point of failure—making it difficult to further develop the proof. We can address both disadvantages at once by translating Liquid Haskell proofs to Coq. To understand this translation, we started by encoding the Ackermann function and related arithmetic theorems in Liquid Haskell and in (to-be-automatically-derived) Coq. In this abstract, we present how we aim to translate the four main ingredients of Liquid Haskell functions and proofs.

1. Refinement Types $\rightarrow$ Subset Types  Refinement types are types refined with logical predicates. For example, $\text{Nat} = \{v : \text{Int} | 0 \leq v\}$ is the type of integers refined to be natural numbers. Liquid Haskell’s refinements are dependent types: $n : \text{Int} \rightarrow \{v : \text{Int} | n \leq v\}$ is the type of a function that increases its integer argument. We could encode refinement types in Coq two ways: inductive predicates or subset types. While Coq works more easily with inductive predicates, they are not a good choice for us: they do not permit “breaking the invariants”. Suppose $n : \text{Nat}$ and $m : \{v : \text{Int} | 1 \leq v\}$. Refinement types can easily show that $(n-1)+m : \text{Nat}$, even though $n-1$ can be negative. In order to separate values and operations from their properties, we encode refinement types as subset types.

In Coq, an element of the subset type $\{v : b | p v\}$ is a pair $(e, q)$ of an expression $e$ and a proof that $p e$ holds. In Liquid Haskell, an element of a subset type is just the expression $e$. Our translation’s primary challenge is to fill in the proof terms.

2. Reflection & Termination Metrics $\rightarrow$ Equations & Induction Principles  Liquid Haskell uses a termination checker to ensure user defined functions terminate—Liquid Haskell can only reason safely about terminating functions. When termination is not structurally obvious, termination metrics let us check semantic termination. For example, the metric $/[m,n]$ below expresses that $\text{ack} m n$ should use lexicographic ordering on its arguments; Liquid Haskell will check that the metric is well founded.

{⁻@ ack :: m:Nat -> n:Nat -> Nat / [m,n] @⁻}

$\text{ack} 0 n = n + 1$

$\text{ack} m n = \text{if} \ n == 0 \ \text{then} \ \text{ack} (m-1) 1 \ \text{else} \ \text{ack} (m-1) (\text{ack} (m-1))$

Terminating functions can be reflected [8] in the refinement logic. The annotation $\{-@ \text{reflect}\ ack \ @⁻\}$ reflects the Ackermann function, which in practice means that $\text{ack}$ can appear in the refinements and that the logic “knows” the function’s definition—so we can more easily write and prove theorems about $\text{ack}$. For example, the theorems below state that $\text{ack}$ is monotonic; their (omitted) proofs are Haskell functions that inhabit the types. (The type $\{p\}$ is really $\{v : (\bot) | p\}$, a refinement of unit used as a notion of proposition.)

{⁻@ monotonic_one :: m:Nat -> n:Nat -> (\text{ack} m n < \text{ack} m (n+1)) / [n] @⁻}

{⁻@ monotonic :: m:Nat -> n:Nat -> p:{Nat | p < n} -> (\text{ack} m p < \text{ack} m n) / [n] @⁻}
Towards a Translation from Liquid Haskell to Coq

Mastorou, Vazou, and Greenberg

We use Coq’s Equations [4] to define functions that are not obviously terminating. Coq’s Fixpoint only permits structural induction, while Program Fixpoint provides opaque functions. We use Equations to encode functions like Ackermann’s in Coq, yielding both the function’s definition and its Equations-generated induction principle.

3. Implicit Semantic Subtyping ⇒ Custom Tactics

Liquid Haskell implicitly uses subtyping to weaken the types of expressions to their appropriate subtypes. Such subtyping occurs in two program locations: join points and function applications. For example, to type if p then 2 else 4 as Nat, the singleton branch types \{ v : Int | v = 2 \} and \{ v : Int | v = 4 \} will both be weakened to Nat via implicit subtyping. Similarly, typing of f 4, where f : Nat → Int, succeeds because of implicit subtyping in the argument.

We created ref_tacts, a new suite of tactics, to simulate Liquid Haskell’s implication checking. At join points, we use the my_trivial tactic emulates what is trivial for Liquid Haskell’s logic, including destruction and lia’s arithmetic. At function applications, the reft_pose tactic does a bit more: (1) it grabs the function’s preconditions, (2) it proves that the arguments satisfy them, and, critically (3) it makes the proof term inside the argument opaque so that it does not clutter the proof environment. Note that functions arguments have correct subset types—but those types may not be the function’s domain type. Part of reft_pose’s job is to ‘upcast’ arguments to satisfy the function domain’s preconditions.

4. SMT & Proof by Logical Evaluation (PLE) ⇒ Sniper

How do we define ref_tacts? Liquid Haskell resolves semantic subtyping’s implications by an SMT solver. SMT solvers know all kinds of things—notably, linear arithmetic. Consider this proof of monotonic:

```
monotonic m n p | n == p + 1 = monotonic_one m p
| otherwise = ack m p ? monotonic m (n-1) p
    <<= ack m (n-1) ? monotonic_one m (n-1)
    <<= ack m n  *** QED
```

The goal is to prove that \( \text{ack} \ m \ p < \text{ack} \ m \ n \). In the base case, where \( n == p + 1 \), the proof concludes by \( \text{ack} \ m \ p < \text{ack} \ m \ (p+1) \). In the inductive case, we use Liquid Haskell’s combinators to build the proof. First, we call the inductive hypothesis \( \text{monotonic} \ m \ (n-1) \ p \) to find \( \text{ack} \ m \ p < \text{ack} \ m \ (n-1) \). Next, \( \text{monotonic_one} \) lets us find \( \text{ack} \ m \ (n-1) < \text{ack} \ m \ n \). The proof concludes by linear arithmetic and transitivity of \(<\), which SMT knows.

Using lia to translate the proof above to Coq is easy. Unfortunately, we must explicitly use transitivity of \(<\) even though transitivity is ‘free’ in SMT!

Proof translation becomes still more challenging Proof by Logical Evaluation [8] (PLE). PLE evaluates expressions in the SMT solver itself, substantially shrinking Liquid Haskell proofs—one need only invoke lemmas. For example, with PLE, the inductive case of monotonic could be \( \text{monotonic} \ m \ (n-1) \ p \ ? \text{monotonic} \ m \ (n-1) \). Translating this simpler proof calls for proof search, since intermediate term for transitivity \( \text{ack} \ m \ (n-1) \) has vanished.

Happily, the recently developed Sniper [2] tactic gracefully combines both SMT knowledge and proof search. Sniper provides general proof automation and combines SMTCoq [1] with general Coq tactics. We conjecture that sniper would be ideal for our Liquid Haskell to Coq translation. In order to apply Sniper in our setting, we must extend it to support equations and subset types that—critical parts of our translation.

Conclusion

We aim to translate Liquid Haskell to Coq. Early experiments have produced workable translations of types, functions, subtyping, and Liquid Haskell’s refinement logic. Sniper [2] offers a promising way forward, once it can reason properly about subset types.
Acknowledgments

We thank Aliaksandr Hryzlou and Chantal Keller for their feedback and discussions. This work is partially founded by the Horizon Europe ERC Starting Grant CRETE (GA: 101039196), the Attracción de Talento grant (Ref. 2019-T2/TIC-13455), and the Juan de la Cierva grant (IJC2019-041599-I).

References


