Liquid Proof Macros

Anonymous Author(s)

Abstract

Liquid Haskell is a popular verifier for Haskell programs, leveraging the power of SMT solvers to ease users’ burden of proof. However, this power does not come without a price: convincing Liquid Haskell that a program is correct often necessitates giving hints to the underlying solver, which can be a tedious and verbose process that sometimes requires intricate knowledge of Liquid Haskell’s inner workings.

In this paper, we present Liquid Proof Macros, an extensible metaprogramming technique and framework for simplifying the development of Liquid Haskell proofs. We describe how to leverage Template Haskell to generate Liquid Haskell proof terms, via a tactic-inspired DSL interface for more concise and user-friendly proofs, and we demonstrate the capabilities of this framework by automating a wide variety of proofs from an existing Liquid Haskell benchmark.


Keywords: Liquid Haskell, proof macros, tactics

ACM Reference Format:

1 Introduction

Liquid Haskell [22] is a popular verifier for Haskell programs, leveraging the power of SMT solvers [2] (such as Z3 [9] or CVC4 [1]) to prove the correctness of diverse applications ranging from optimizations [23] to string matching algorithms [24]. Specifications for these applications are written in the form of refinement types [11], boolean predicates over program values.

For concreteness, consider the following min function that computes the minimum of two natural numbers, defined inductively:

```haskell
data N = Z | S N

min :: N → N → N
min Z _ = Z
min _ Z = Z
min (S m) (S n) = S (min m n)
```

Naturally, we would expect such a function to be associative, that is:

$$\forall a \ b \ c. \ \text{min} \ (\text{min} \ a \ b) \ c = \text{min} \ a \ (\text{min} \ b \ c)$$

In Liquid Haskell, we can specify associativity by defining a refinement type to encode this property, and we can prove associativity by defining a term of that type:

- assocMin :: a:N → b:N → c:N → _:
  \{(\_::() | min (min a b) c == min a (min b c))\} @-

To Haskell, the type of assocMin is simply a function with three natural number arguments that returns a Proof, which is just a type synonym for (). To Liquid Haskell, however, the type of assocMin is much more interesting: its return type does not only specify that the output is a unit, but refines it so that associativity of min holds for its input arguments. In other words, the only interesting thing about the result of this function is its refinement, which constitutes an “extrinsic style” proof of associativity. This is a common enough pattern that Liquid Haskell supports dropping the “\_::()” part of the refinement for brevity, as we will also do in the remainder of this paper.

But how does Liquid Haskell decide if the refinement type is true? By reducing typechecking to verification conditions that SMT solvers reason about. However, while SMT solvers are pre-programmed with a wide assortment of facts about various domains such as integer arithmetic and boolean logic, they don’t really know anything about user-defined data types like N or user-defined functions like min. While a direct encoding of such features to SMT is possible in principle [26], it leads to unpredictable verification, also known as the “butterfly effect” [13]. To that end, Liquid Haskell lifts user-defined data types and functions into a representation that can be handled symbolically by SMT solvers [25]. Still, many true properties of user-defined data types and functions remain not automatically verifiable: users must guide, via refined Haskell code, the SMT solver to simpler cases that can be checked automatically.

Unfortunately, given the lack of interactivity of Liquid Haskell, it is not always clear what the gap in understanding between the user and the SMT solver is, which often makes writing such refined code a tedious and frustrating process. Consider again associativity for the min function. On paper, we can informally reason that associativity holds by induction on the natural numbers that are inputs to min, due to its simple recursive structure. In Liquid Haskell, the refined code that finally convinces the SMT solver that the program typechecks is shown in Figure 1.

All of the branches of pattern matching on a, b, and c must be written out explicitly. Otherwise, the SMT solver would not know how to simplify the min expressions in the
{-@ assocMin :: a:N -> b:N -> c:N ->
  { min (min a b) c == min a (min b c) } @-}

assocMin :: N -> N -> N -> Proof
assocMin = \a b c ->
case a of
  Z ->
    case b of
      Z ->
        case c of
          Z -> trivial
          S c' -> trivial
          S b' ->
        case c of
          Z -> trivial
          S c' -> trivial
      S a' ->
        case b of
          Z ->
            case c of
              Z -> trivial
              S c' -> trivial
              S b' ->
            case c of
              Z -> trivial
              S c' -> assocMin a' b' c'

Figure 1. Liquid Haskell proof term for associativity of \(\text{min}\)

refinement—the only facts it knows are the three equations
that were used in \(\text{min}\)'s definition: \(\min Z = Z, \min _ Z = Z, \text{and} \min (S x) (S y) = S (\min x y)\). Liquid
Haskell understands the constraints introduced by pattern
matching, and takes them into account in order to discharge
most cases—the non-recursive ones that involve at least one
\(Z\). The proof conclusion in such cases is trivial, which is
again just a synonym for the term-level (\()\).

However, in the recursive case of \(a\), the Liquid Haskell
typechecker needs additional help, in the form of a recursive
call to assocMin \(\text{a' b' c'}\), which brings its refinement in
scope for the SMT solver and allows it to conclude that the
induced verification condition holds. Crucially, this refine-
ment is again the only thing that matters: while the structure
of the term gives the appearance of a proof term in the style
of Coq or Agda, the actual return value doesn’t matter. We
could just as well have written something like

\[
\text{snd } \text{assocMin } a' b' c', ()
\]

and Liquid Haskell would still gladly accept the definition.
In fact, Liquid Haskell’s conjunction operator (&&&) is de-

defined exactly this way: it takes two Proofs and returns the
second one—its only effect is making the refinement of both
arguments visible to the SMT solver.

{-@ assocMin :: a:N -> b:N -> c:N ->
  { min (min a b) c == min a (min b c) } @-}

[tactic]
assocMin :: N -> N -> N -> Proof
assocMin a b c = induct a; induct b; induct c

Figure 2. Associativity of \(\text{min}\) using Liquid Proof Macros

Even in this simple example of associativity of \(\text{min}\), the
full verbosity required is cumbersome and obscures the fact
that the underlying argument is a straightforward induc-
tion. In larger developments where the SMT solver might
need to rely on helper lemmas, this problem only becomes
more pronounced. Other proof assistants, such as Coq [21],
Lean [15], or Isabelle [16], rely on interactive tactics in these
situations to aid users’ proof efforts. But developers of these
tactic languages enjoy a transparent API to interact with the
current proof state, and an essentially clean slate to design
metaprogramming capabilities, which has been exploited to
the great benefit of proof assistant users [10, 19, 27].

On the other hand, Liquid Haskell interacts with the SMT
solver in a very opaque manner, and within the Haskell
ecosystem metaprogramming capabilities are already well
established in the form of Template Haskell—but not really
designed with SMT-based verification in mind. So then, what
can we do within the confines of this mature Haskell ecosystem
to aid users? Without interactivity, an interface to concise
proof generators must expand to a proof term all at once
i.e. it must behave like a macro. Therefore, we developed
a macro system for generating Liquid Haskell proof terms,
using the existing metaprogramming tools for Haskell.

Liquid Proof Macros. In this paper, we show how to
leverage the power of Template Haskell to automate proof
term generation for Liquid Haskell. We develop Liquid Proof
Macros, an extensible DSL in which users can write intuitive
proofs that resemble automated tactics of more traditional
proof assistants, including case analysis, induction, condi-
tioning, and proof search. For example, the same proof of
associativity of \(\text{min}\) using Liquid Proof macros can be seen in
Figure 2.

These macros are expanded to a subset of Haskell that
resembles, or rather is even more complicated than, the one
used in Figure 1. To facilitate typechecking of larger Liquid
Haskell developments, we also augment this subset with
metadata information, and provide a pruning algorithm re-
miniscen of shrinking in property-based testing [6], simpli-
fying away any unnecessary components that result from
proof search.

\[\text{We refrain calling our DSL tactics, as that suggests a notion of interactivity}
\text{that is impossible in the current version of Liquid Haskell.}\]
In this paper we make the following contributions:

- We describe a methodology for using Template Haskell to automatically construct Liquid Haskell proof terms, and we develop an extensible framework using this methodology for automating inductive proofs in Liquid Haskell (Section 2).
- We evaluate our framework against an existing benchmark containing a wide variety of Liquid Haskell properties, and found that our Liquid Proof Macros can be used to automate all of these properties, leading to a 1.57x reduction in code on average (Section 3).

We then discuss related work (Section 4), before concluding with a discussion of the limitations of our framework and directions for future work (Section 5).

2. Liquid Proof Macros

Going from a proof macro like the one we saw earlier in Figure 2 to a low-level Liquid Haskell proof such as the one in Figure 1 is a multi-stage process, which we will describe in detail in this section.

First we introduce the *proto-proof language* (Section 2.1), a subset of (surface) Haskell with some annotations that are necessary for simplification, but which can be straightforwardly erased to obtain valid Liquid Haskell terms. This will serve as the language that Liquid Proof macros expand to.

Second, we formally introduce the *proof macro language* in which users write proofs (Section 2.2), an extensible collection of high-level constructs (such as *induct*) that facilitate SMT reasoning. We show how a proof macro can be expanded into a proto-proof term, which is then cached, embedded in Haskell, and spliced in place of the original proof macro.

Then, we extend this language to allow for binding-based conditional expansion of macros (Section 2.3), a way of organizing branches proofs based on variables introduced during the expansion process.

Finally, we describe how cached terms are repeatedly pruned by using the metadata annotations in the proto-proof language syntax (Section 2.4), removing potentially unnecessary proof terms by using Liquid Haskell as the validity oracle.

2.1 The proto-proof language

The proto-proof language is at its core a subset of Haskell expressions with some additional metadata. Figure 3 depicts its syntax, which contains lambdas, pattern matching with case, if statements, Liquid Haskell’s conjunction (&&&) and trivial, as well as a special construct Auto that keeps track of three lists of arbitrary Haskell expressions for simplification purposes.

With the exception of Auto, terms in the proto-proof language can be directly embedded into Haskell. In turn, Auto can be embedded by translating the kept and init lists of expressions to sequences of Liquid Haskell conjunctions—we will expand on this when discussing pruning later in this section. That is:

```
Auto { init = [exp1, ..., expn],
      kept = [exp1, ..., expn],
      pruned = []
}
```

is embedded into Haskell as:

```
a1 &&& ... &&& am &&& b1 &&& ... &&& bn
```

2.2 The proof macro language

The proof macro language defines a collection of proof macros that aim to concisely describe the high-level structure of a Liquid Haskell proof. This collection is designed to be extensible, so that new proof macros can be added easily by adding a new constructor to the proof macro language and then defining its expansion. The syntax for proof macro language appears in Figure 4, and consists of atomic macros that can be sequenced together. Liquid Haskell users can write such sequences of proof macros, like the three *inducts* in Figure 2, which are then expanded into the proto-proof language.

To expand each proof macro, we need to take into account the expansion of any macros that preceded it in the sequence. To that end, we introduce two contexts, a typing context Γ which associates variables to Haskell types, and a recursion context P which associates arguments of the top-level declaration to a (potentially empty) set of subterms that can be used to instantiate recursive calls without triggering

```
decl-proto ::= f :: typ
            | f = expr-proto;

expr-proto ::= \name \to expr-proto
             | case expr of pat \to expr-proto
             | if expr then expr-proto else expr-proto
             | expr &&& expr-proto
             | trivial
             | Auto {
                  init = [exp],
                  kept = [exp],
                  pruned = []
             }

exp ::= Haskell expression
pat ::= Haskell pattern
typ ::= Haskell type (monomorphic)
f, x ::= Haskell name
n ∈ N
```

Figure 3. Syntax of the proto-proof language
exp-macro ::= \textit{induct} \textit{x} \\
| \textit{destruct} \textit{exp} \\
| \textit{assert} \textit{exp} \\
| \textit{dismiss} \textit{exp} \\
| \textit{condition} \textit{exp} \\
| \textit{auto} \begin{scriptsize}\{\textit{\pi}\end{scriptsize} \textit{n} \\
| \textit{use} \textit{exp} \\
| \textit{trivial}

\text{Figure 4. The proof macro language}

\begin{align*}
\text{decl-macro} & ::= \textit{f ::} \ \textit{typ} \\
f \ y_i & = \textit{exp-macro}; \\
\text{exp-macro} & ::= \textit{induct} \textit{x} \\
| & \textit{destruct} \textit{exp} \\
| & \textit{assert} \textit{exp} \\
| & \textit{dismiss} \textit{exp} \\
| & \textit{condition} \textit{exp} \\
| & \textit{auto} \begin{scriptsize}\{\textit{\pi}\end{scriptsize} \textit{n} \\
| & \textit{use} \textit{exp} \\
| & \textit{trivial}
\end{align*}

\text{exp} ::= \textit{Haskell expression} \\
\text{typ} ::= \textit{Haskell type (monomorphic)} \\
f, x, y_i ::= \textit{Haskell name} \\
n \in \mathbb{N}

\begin{itemize}
\item \textit{condition} \textit{e}, \textit{assert} \textit{e}, \textit{dismiss} \textit{e}:
These three macros all expand into \texttt{if} statements with the given boolean expression \textit{e} as its condition. The difference lies in the holes produced in this expansion: \textit{condition} creates a hole \texttt{\Gamma\_P} for both the then and the else branches; \textit{assert} only creates such a hole for the then branch with the else branch being \texttt{trivial}; and \textit{dismiss} is the dual of \textit{assert}.

\textbf{Evidence Macros}. Evidence macros are processed into terms that provide evidence to the Liquid Haskell typechecker, such as introducing a lemma to the refinement context.
\begin{itemize}
\item \texttt{trivial}:
This macro expands into Liquid Haskell’s \texttt{trivial} with type \texttt{Proof in the resulting Haskell term. Since \texttt{trivial} = () and \texttt{Proof} = (), using this macro effectively means that the SMT solver can discharge any remaining obligations.
\item \texttt{use} \textit{exp}:
This macro makes the refinement type of the expression available to the SMT solver using Liquid Haskell’s conjunction \texttt{&&}, similar to how, in Figure 1, a call to \texttt{assocMin a’ b’ c’} was needed to conclude the proof.
\item \texttt{auto} \begin{scriptsize}\{\textit{\pi}\end{scriptsize} \textit{n}:
Finally, the \texttt{auto} macro is the core of our framework’s automation. It takes two optional parameters, a sequence \texttt{\pi} of \texttt{hints}, and a natural number \textit{n}, and it generates all well-typed neutral forms of type \texttt{Proof up to height \textit{n} that use variables from the current context or the hints}. To ensure recursive calls are terminating, we keep track of a separate recursion context that is specially constructed in the rest of the proof.
\end{itemize}

Armed with the expansion relation for a single proof macro, we can formalize the expansion of a macro sequence (Figure 6). The empty proof macro sequence is expanded into a \texttt{trivial}—as that has no effect on SMT resolution. To expand a sequence \texttt{T; Ts} in some contexts \texttt{\Gamma} and \texttt{P}, we expand \texttt{T} to some proto-proof term \texttt{t}, and for every hole \texttt{\Gamma\_P}, in \texttt{t}, we recursively expand the sequence \texttt{Ts} in the updated context, and replace the hole with the result. That is, whenever proof macros introduce multiple branches, the rest of the sequence of proof macros is expanded into each such branch, unlike traditional proof assistants. We will return to this point when discussing conditional expansion of macros below (Section 2.3).

Finally, before expansion of a sequence we preprocess it: if it does not end in an evidence macro, then a default \texttt{auto} macro with no hints and height 3 will be implicitly included at the end of the sequence. That is, the proof macro of Figure 2 is equivalent to the following one that includes an explicit \texttt{auto \emptyset 3} macro:
\begin{align*}
\texttt{induct a; induct b; induct c; auto \emptyset 3}
\end{align*}
\[ \Gamma \vdash x : \alpha \quad x \not\in P \quad \Gamma \vdash c_i : \beta_{ij} \rightarrow \alpha \]
\[ \Gamma_i = (\Gamma \cup \{ y_{ij} : \beta_{ij} \}) \quad P_i = P[x \mapsto \{ y_{ij} : \beta_{ij} \}] \]
\[ \Gamma; P \vdash \text{induct } x \sim \text{ case } x \text{ of } c_i \ y_{ij} \rightarrow \Box_{\Gamma,P} ; \]
\[ \Gamma \vdash e : \alpha \quad \Gamma \vdash c_i : \beta_{ij} \rightarrow \alpha \]
\[ \Gamma_i := \Gamma \cup \{ y_{ij} : \beta_{ij} \} \]
\[ \Gamma; P \vdash \text{destruct } e \sim \text{ case } e \text{ of } c_i \ y_{ij} \rightarrow \Box_{\Gamma,P} ; \]
\[ \Gamma \vdash e : \text{Bool} \]
\[ \Gamma; P \vdash \text{condition } e \sim \text{ if } e \text{ then } \Box_{\Gamma,P} \text{ else trivial} \]
\[ \Gamma \vdash e : \text{Bool} \]
\[ \Gamma; P \vdash \text{assert } e \sim \text{ if } e \text{ then } \Box_{\Gamma,P} \text{ else trivial} \]
\[ \Gamma \vdash e : \text{Proof} \]
\[ \Gamma; P \vdash \text{use } \{ e \} \sim e \quad \text{&} \quad \text{&} \quad \text{&} \quad \Box_{\Gamma,P} \]

\[ \Gamma; P \vdash \text{trivial } \sim \text{ trivial} \]
\[ \Gamma \vdash x_i : \alpha_i \quad n \in \mathbb{N} \quad is = \text{generate}(\Gamma \cup \{ x_i \}, P, \text{Proof}, n) \]
\[ \Gamma; P \vdash \text{auto } [x_i] n \sim \text{Auto } [\text{init } = \text{is}, \text{kept } = [\text{ } ], \text{pruned } = [\text{ } ] ] \quad \text{&} \quad \text{&} \quad \text{&} \quad \Box_{\Gamma,P} \]

\begin{align*}
generate(\Gamma, P, \alpha, 0) & = \{ \text{trivial} \} \\
generate(\Gamma, P, \beta, 1 + n) & = \{ f \ a_1 \cdots a_n \mid \Gamma \vdash f : \alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow \beta, \ a_i \in \text{generate}(\Gamma, P, \alpha_i, n) \} \\
& \cup \{ r \ a_1 \cdots a_n \mid \Gamma ; r : \alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow \beta, \ a_i \in P, \ x_i \mapsto P_i \in P \} \\
\end{align*}

Figure 5. Proof macro semantics

\[ [\cdot](\Box_{\Gamma,P}) = \text{trivial} \]
\[ \Gamma; P \vdash T \sim t \]
\[ \Box_{\Gamma,P} \in t \quad \langle T_s, (\Box_{\Gamma,P}) \rangle = t_i \]
\[ \langle T; T_s, (\Box_{\Gamma,P}) \rangle = t[t_i/\Box_{\Gamma,P}] \]

Figure 6. Proof Macro Sequence Expansion

Extended Example. For concreteness, consider the following predicate which states that if a number \( x \) is an element of a list \( xs \), then \( x \) is also in the list \( xs ++ ys \) for an arbitrary list \( ys \), along with a corresponding Liquid Haskell theorem that is proved with a short Liquid Proof macro. ²

\[
\text{concatElemP} :: N \rightarrow [N] \rightarrow [N] \rightarrow \text{Bool}
\text{concatElemP} x xs ys
| \text{elem } x xs = \text{elem } x (xs ++ ys)
| \text{otherwise } = \top
\]

{\text{tactic}}
\[
\text{concatElem} :: N \rightarrow [N] \rightarrow [N] \rightarrow \text{Proof}
\text{concatElem} x xs ys =
\text{assert } \{(\text{elem } x xs) ;
\text{induct } xs \}
\]

At a high level, what this proof macro does is condition on the expression \( \text{elem } x xs \), pattern match on \( xs \), and search for ways to complete the proof, potentially using the tail of \( xs \) for a recursive call. This is achieved by expanding these macros back into the proto-proof language as shown in Figure 7.

Based on the declaration of \( \text{concatElem} \) inside the quasi-quoter, we can initialize the typing context with the types for \( x, xs \), and \( ys \).³ Similarly, the recursion context is initialized with the empty set for all of the function arguments. The proof macro sequence is then preprocessed and, since it doesn’t end with an explicit evidence macro, an \text{auto } \{ \} ³ is appended at its end.

The first proof macro in the sequence is \text{assert } \{(\text{elem } x ys) \}, which creates an if statement whose condition is \( \text{elem } x ys \) and whose else branch is simply trivial. The then branch is a hole \( \Box_{\Gamma,P} \), which will be filled by expanding the rest of the sequence, beginning with \text{induct } xs.

²For the sake of readability, we used the usual list notations such as \((++)\) for list append, rather than their refined list counterparts.

³Types of local bindings cannot be inferred with Template Haskell, so the type signature is in fact necessary.
\[ \Gamma = \{ x : N, \, xs : [N], \, ys : [N] \} \]
\[ P = \{ x \mapsto \emptyset, \, xs \mapsto \emptyset, \, ys \mapsto \emptyset \} \]

\[
\begin{align*}
\text{if elem x xs then} & \quad \text{induct xs; auto \;} 3\; (\Gamma; P) \\
\text{else trivial} & \quad \text{induct xs; auto \;} 3\; (\Gamma; P) \\
\text{case xs of} & \quad \text{induct xs; auto \;} 3\; (\Gamma; P) \\
\text{auto \;} & \quad \text{auto \;}\; 3\; (\Gamma; P) \\
\text{generate}(\Gamma, P, 3) & \quad \text{generate}(\Gamma, P) \\
\text{auto \;} & \quad \text{auto \;}\; 3\; (\Gamma; P) \\
\text{auto \;} & \quad \text{auto \;}\; 3\; (\Gamma; P) \\
\text{trivial} & \quad \text{trivial} \\
\text{init =} & \quad \text{init =} \\
\text{concatElem \;} x' \, xs' \, xs' & \quad \text{concatElem \;} x' \, xs' \, xs' \\
\text{concatElem \;} x' \, xs' \, ys & \quad \text{concatElem \;} x' \, xs' \, ys \\
\text{concatElem \;} x \, xs' \, xs' & \quad \text{concatElem \;} x \, xs' \, xs' \\
\text{concatElem \;} x \, xs' \, ys & \quad \text{concatElem \;} x \, xs' \, ys \\
\text{kept = } & \quad \text{kept = } \\
\text{pruned = } & \quad \text{pruned = } \\
\text{trivial} & \quad \text{trivial} \\
\text{else trivial} & \quad \text{else trivial} \\
\end{align*}
\]

*Figure 7. Step-by-step Expansion of a Proof Macro*

The `induct` xs macro expands into a pattern matching with two cases, one for the empty list and one for a nonempty list `x':xs'`, for fresh variables `x'` and `xs'`. Both branch bodies are holes to be filled by the expansion of an `auto` macro, but at different contexts. The empty list branch did not introduce any new variables, and as a result both contexts remain unchanged. In the nonempty branch, the variables `x'` and `xs'` are added to both the typing and to the the recursion context for the second argument position, since `xs` is in the second argument position.

Finally, the `auto` macros are expanded using the `generate` metafunction, which yields just `trivial` in the empty case, and a bunch of different potential recursive calls in the nonempty case, as there are several neutral forms available:

- `concatElem \ x' \ xs' \ xs'`
- `concatElem \ x' \ xs' \ ys`
- `concatElem \ x \ xs' \ xs'`
- `concatElem \ x \ xs' \ ys`

All of these recursive calls are potentially valid, as they all have at least one argument from the recursion context at that argument position: `xs'`. This list of neutral forms can be seen in `init` field of the `Auto` structure in the final step of Figure 7.

### 2.3 Variables and Conditional Expansion

The final piece of the puzzle involves the treatment of variables that are introduced during macro expansion. From the proof macros defined above, such variables can only be introduced by `induct` and `destruct`. These control flow macros do not, by default, give the user the ability to name the introduced variables — the generated names are fresh via Template Haskell. Extending our framework with this capability is just a matter of changing the `induct` and `destruct` constructors from Figure 4 to include optional name annotations with the following syntax:
With a sequence data structure that has both its first and last element exposed (as is the case in finger trees [5]), we could selectively destruct the first (or last!) element of it after induction. That is, given the following data type:

\[\text{data Seq a = Nil | Unit a | More a (Seq a) a}\]

the following proof macro would only be expanded in the Unit and More cases:

\[\text{induct s as } [ / x / x s' y];\]
\[\text{[x:] destruct x}\]

Anecdotally, when carrying out our evaluation, we found this binding-based conditional expansion to be particularly useful in organizing our macros and avoid redundancy in proofs.

### 2.4 Pruning

Of course, it turns out that not all of these terms are needed for a valid proof. The pruning process, uses the rest of the Auto record to safely prune such unnecessary terms.

For each Auto structure in a proto-proof term, each \(\text{exp}\) in its init field is attempted to be pruned one at a time. This is done by moving the \(\text{exp}\) from the init field to the pruned field, embedding and splicing the new proto-proof term into the original Haskell file in place of the original proof macro, and then running Liquid Haskell to check if this prune was safe. If it was, pruning continues with the rest of the \(\text{exp}\)s in the init fields of the Auto structures; otherwise this prune is undone, and the \(\text{exp}\) that was attempted to be pruned is instead moved to the kept field before continuing pruning. This process is very similar to shrinking in the style of QuickCheck [6] from the property-based testing literature.

Recall the proto-proof term that resulted from processing the proof macro used to prove \(\text{concatElem}\). There is an Auto structure that clearly has a few \(\text{exp}\)s that can be pruned since they are unnecessary for the proof. The subset of necessary \(\text{exp}\)s is found via the linear pruning procedure, trying to remove each \(\text{exp}\) one at a time to see which can be safely removed. After pruning, the final resulting proto-proof term can be embedded into Haskell a last time and presented to the user as a valid proof:

\[
\text{concatElem :: N -> [N] -> [N] -> Proof}\\
\text{concatElem = \text{\ } x \text{ x s y \ } \rightarrow}\\
\text{\text{\ } if \text{ elem x s y \ } \rightarrow}\\
\text{\text{\ } case x s of}\\
\text{Nil \rightarrow trivial}\\
\text{Cons x' x s' \rightarrow concatElem x x' s y}\\
\text{else}\\
\text{trivial}
\]

Note that this is the minimal proof a user would need to write in Liquid Haskell to convince its typechecker that \(\text{concatElem}\) is well typed.

### 3 Design, Evaluation, and Usage

In this section, we focus on the choices that influenced our design (Section 3.1), we evaluate these choices in an existing Liquid Haskell benchmark of programs (Section 3.2), and demonstrate the usage of our tool with a small example (Section 3.3).

#### 3.1 Design Choices

When designing Liquid Proof Macros, we wanted users to benefit along the following three axes:

**Conciseness.** Extrinsic proofs are mostly written in a particular subset of Haskell, the proto-proof language. Using metaprogramming to generate terms in this subset we would expect proofs to be shorter than when written out explicitly. Moreover, as most proof assistant users are familiar with, often many different branches of a proof can be handled by the same proof strategy. Rather than requiring users to replicate the same proof term in each branch, proof macros...
allow for writing such strategies once and applying them across all branches.

**Modularity.** Even though many proofs can be encapsulated by the same proof strategy (e.g. simple induction), vanilla Liquid Haskell requires that strategy to be written out in full verbosity in each instance (e.g. pattern matching on a list of a natural numbers, and then supplying the tail or predecessor to the recursive call in the second cases respectively). The proof macro system allows the user to modularly encode proof strategies in such a way that the same sequence of proof macros can be used to prove a wide variety of similar theorems that use the same proof strategy.

**Practicality.** The style of proof search used by our proof macro system is very inefficient, verily because it includes all generatable neutral forms (using a limited) context without using any sort of guided search. The secondary pruning process, which is performed on a passing proto-proof term after proof search is complete, aims to recover a minimal proof term for the sake of readability and efficient re-checking.

### 3.2 Evaluation

To evaluate our design, we turned to a prior benchmark suite of 80+ Liquid Haskell proofs [12], that consists of a collection of boolean predicates over natural numbers, lists, and binary trees, ranging from very simple facts to inductive properties that require auxiliary lemmas. All proofs take advantage of both proof macros, and *proof by logical evaluation* [25], a complementary technique for delegating some equational reasoning to the SMT solver.

Using Liquid Proof Macros we were able to concisely prove (< 16 LoC\(^4\) for all and < 10 LoC for all but two). Figure 8 shows the lines of code across all such benchmark using (1) Liquid Proof Macros (2) the expanded proof term (3) the pruned minimal proof term. We found that in all cases proof macros where smaller than the Liquid Haskell proof terms, with a (geometric) average of 57% reduction in LoC compared to the minimal pruned version. Moreover, to give an estimate of the cost of proof search, the unpruned expanded splice generated by Template Haskell was \(2.88\times\) larger on average.

### 3.3 Usage Example

Figure 9 shows the user’s workflow when proving a theorem such as `assocMin` from the introduction (Figure 2). It shows three screenshots of VSCode with proof macro processing in progress. First, the user must invoke our tool *lh-tactics*,

\(^4\)Measured from the signature of the top-level function to the last line of the macro sequence or proof term.
The user writes the proof macro and runs the \texttt{lh-tactics} command line tool on the input file, which exists inside of a \texttt{stack} project that is configured to use the LiquidHaskell as a plugin.

The user waits for the \texttt{lh-tactics} tool to complete. During this time, the tool will overwrite the input file on each pruning attempt.

Once pruning has completed, the final proof term is presented and the original proof macros that generated it are left in a comment immediately above.

\textbf{Figure 9.} Usage example of the proof macros tool.
which then parses the proof macro, expands it, and repeatedly prunes it until a minimal proof term is reached (see Figure 1).

4 Related Work

The formal verification and proof assistant literature is vast. Here, we discuss the most directly relevant related work, focusing on automation and metaprogramming in such frameworks.

Meta F*: Arguably the closest related work is Meta-F* [14], the tactics and metaprogramming framework for the F* language [20]. In this work, Martínez et al. face the same issues that Liquid Haskell users face: how to reconcile the automatic but black-box nature of SMT-aided program verifiers with the expressive tactic-based facilities of interactive theorem provers. Their approach is similar in nature to ours, but enjoys the benefit of developing the metaprogramming framework with the particular use case of verification in mind. In contrast, we showed how one can work within the already established constraints and limitations of the Haskell ecosystem, using Template Haskell to provide a more streamlined user experience.

Interactive Tactics. In the land of interactive proof assistants like Coq [21], Lean [15], or Isabelle [16], tactics are the primary way by which users interactively manipulate the systems proof state. Tactics are usually written in a meta-language that is built with the explicit purpose of developing proofs, often evolving along the proof assistant. For example, Coq’s tactic language Ltac [10] has been the target of multiple enhancement attempts, such as Mtap [27] that enforced a typing discipline, or Ltac2 [19] which provides more advanced metaprogramming capabilities [17]. Such tactics operate on the underlying representation of the proof state in a proof assistant, and are therefore inherently more expressive in the capabilities they provide. In this work, we drew inspiration from the kind of reasoning that tactics allow for, to give Liquid Haskell users the ability to write more concise and modular proofs.

Moreover, hammers have been developed for proof assistants such as Coqhammer [7, 8] for Coq or Sledgehammer [3, 18] for Isabelle, which aim to bring the benefits of automated verification to the interactive setting. Hammers give the users the ability to directly discharge their current goal, but suffer from the same drawback as program verifiers like Liquid Haskell: there is little the user can do if the hammer fails, without resorting back to tactic-based reasoning.

Liquid Haskell Automation. Naturally, we are not the first to attempt automating Liquid Haskell proof generation. First, Vazou et al. [25] introduced proof by logical evaluation, a proof search technique inspired by abstract interpretation to automate equational reasoning in Liquid Haskell, by increasing the burden on the SMT solver. This is largely orthogonal to Liquid Proof Macros, as it operates on function definitions while our macros are focused on structural reasoning and searching for hints. More recently, Hafidi [12] developed a quasiquote that allows Liquid Haskell to use different ‘techniques’, such as induction, during SMT solving. However, as we saw in the evaluation section Liquid Proof Macros completely subsume all of its functionality, while allowing for finer control over proof generation. Finally, Haskell users can gain access to proof-assistant-based reasoning by using HS-to-Coq [4], a tool that translates Haskell programs to Coq ones. In this work, we instead tried to bring some of the advantages of that style of reasoning within the Haskell ecosystem.

5 Conclusion

In this paper, we demonstrated how we can operate within the bounds of the existing Haskell ecosystem, and provide a light weight solution to proof automation in Liquid Haskell by leveraging Template Haskell functionality. While our framework can already handle a wide variety of properties of interest, there are still many reasonable extensions to consider, requiring varying degrees of implementation effort.

Simple Extensions. The proof macro system currently only supports simple pattern matching via the destruct and induct macros. However, tactic languages in proof assistants demonstrate how deeper pattern matching can be given a convenient interface and be very useful. Such pattern matching features can easily be implemented in the proof macro language by expanding them to a sequence of existing Liquid Proof Macros.

In the same spirit, there is currently no way to define an abstract macro that expands into a sequence of macros, resulting in needless redundancy where many proofs contain the same sequence of macros, differing only in the particular argument given. Such sequences are again straightforward to implement at the framework level, as proof macro language is easy to extend. However, providing such functionality at the user level is a more ambitious endeavor that we leave for future work.

Finally, our auto macro is implemented in the minimally complex way while still being useful: it simply generates every neutral form it can up to a certain syntactic height. However, more specific kinds of similar searches in the space of neutral terms can be allowed, such as a refined → auto macro that take as input a neutral form with holes in place of some of its (perhaps nested) arguments. Then, the macro system would generate all neutral forms that correspond to the original neutral form with its holes filled by neutral forms. For example,

\[
\text{refined-auto \{assocMin (f \_ n) \_\}} [] 3
\]

where \( f : : N \rightarrow N \rightarrow N, m, n :: N \) would generate all neutral forms of the form assocMin \( (f \_ n) (f \_ k) \).
where $k$ is a range over all neutral forms constructed from values in context (including valid recursions) up to a certain syntactic height.

**Engineering Challenges.** In addition to the simple extensions described above, our framework could currently be improved with some investment in a non-trivial but straightforward engineering effort. In particular, the user interface to Liquid Haskell has been developed into a plugin that works in tandem with the Liquid Haskell stack build system. Currently, the proof macro system requires the user to run an external tool on proof macros for pruning purposes. User experience would be greatly improved if the proof macro system was integrated into the existing Liquid Haskell plugin, and run automatically when the project is built.

Similarly, Template Haskell splices code implicitly during compilation, in such a way that the splices are never actually displayed inline with the user’s original code. Currently, Template Haskell is not well-supported by Liquid Haskell, and our external tool explicitly splices the pruned code in for efficiency purposes. It would be interesting to further explore this interaction between Template and Liquid Haskell to see if we can get the best of both worlds: the conciseness of proof macros with the efficient compilation of the pruned proof terms.

Moreover, the current `auto` macro cannot handle polymorphism, because as Template Haskell only provides support for syntactic equality when checking if a value’s type is compatible with the type expected for an argument in a neutral form being generated. Supporting polymorphism would require writing a simple unification function at the Template Haskell level, which would fit nicely with the rest of our framework.

**Research Challenges.** Outside of the aforementioned implementation drawbacks that can easily be overcome, there remain two significant research questions that limit the usability of our current approach.

First, the pruning algorithm used is guaranteed to find the subset of the `auto`-generated `exp`s that make the proof pass, if such a subset exists, but it is a slow process. As shown in Figure 8, sometimes the number of `exp`s generated is too large to be pruned in a reasonable amount of time. A smarter approach would need to be devised to scale the minimization to larger case studies.

Second, Liquid Proof Macros still suffer from the lack of interactivity, which limits their usefulness compared to their tactic counterparts in traditional proof assistants. To enable such interactivity, we need to fundamentally rethink the way Liquid Haskell communicates with the underlying SMT solver. Until then, Liquid Proof Macros are a very useful abstraction to reduce the burden of Liquid Haskell users.

References


