Liquid Proof Macros

Henry Blanchette
blancheh@umd.edu
University of Maryland
College Park, USA

Niki Vazou
niki.vazou@imdea.org
IMDEA
Madrid, Spain

Leonidas Lampropoulos
leonidas@umd.edu
University of Maryland
College Park, USA

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.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Keywords: Liquid Haskell, proof macros, tactics

ACM Reference Format:
Henry Blanchette, Niki Vazou, and Leonidas Lampropoulos. 2022. Liquid Proof Macros. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium (Haskell '22), September 15–16, 2022, Ljubljana, Slovenia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3546189.3549921

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 [?]. Specifications for these applications are written in the form of *refinement types* [12], boolean predicates over program values.

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

```
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:

$$\forall a\ b\ c.\ min\ (min\ a\ b)\ c == min\ a\ (min\ b\ c)$$

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)}
@-}
assocMin :: N → N → N → Proof
assocMin = ...
```

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 [25], it leads to unpredictable verification, also known as the "butterfly effect" [14]. To that end, Liquid Haskell lifts

Haskell '22, September 15–16, 2022, Ljubljana, Slovenia

ACM ISBN 978-1-4503-9438-3/22/09…$15.00
https://doi.org/10.1145/3546189.3549921

```
{-@ 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 `min`

user-defined data types and functions into a representation that can be handled symbolically by SMT solvers [24]. 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 refinement—the only facts it knows are the three equations that were used in `min`'s definition: `min Z _ = Z`, `min _ Z = Z`, 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 `min`, the Liquid Haskell typechecker needs additional help, in the form of a recursive call to `assocMin 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 refinement 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

```
snd (assocMin a' b' c', ())
```

and Liquid Haskell would still gladly accept the definition. In fact, Liquid Haskell's conjunction operator (&&&) is defined exactly this way: it takes two `Proof`s and returns the second one—its only effect is making the refinement of both arguments visible to the SMT solver.

Even in this simple example of associativity of `min`, the full verbosity required is cumbersome and obscures the fact that the underlying argument is a straightforward induction. 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 [16], or Isabelle [17], 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, 26].

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 intutive proofs that resemble automated tactics[1] of more traditional proof assistants, including case analysis, induction, conditioning, and proof search. For example, the same proof of associativity of `min` using Liquid Proof macros can be seen in Figure 2.

[1] We refrain calling our DSL tactics in this paper, as that suggests a notion of interactivity that is impossible in the current version of Liquid Haskell. In truth, they lie somewhere between tactics (no interactivity) and macros (users have to dive into Template Haskell to extend the DSL).

```
{-@ 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 `min` using Liquid Proof Macros

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 reminiscent of shrinking in property-based testing [6], simplifying away any unnecessary components that result from proof search. Section 2.4 details how our system defines and handles pruning.

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 1.57× 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.

$$
\begin{array}{rcl}
\textit{decl-proto} & ::= & f :: \textit{typ} \\
 & & f \ = \ \overline{\textit{exp-proto}}; \\
\textit{exp-proto} & ::= & \lambda\ \textit{name} \rightarrow \textit{exp-proto} \\
 & | & \texttt{case}\ \textit{exp}\ \texttt{of}\ \overline{\textit{pat} \rightarrow \textit{exp-proto}}; \\
 & | & \texttt{if}\ \textit{exp}\ \texttt{then}\ \textit{exp-proto}\ \texttt{else}\ \textit{exp-proto} \\
 & | & \textit{exp}\ \&\&\&\ \textit{exp-proto} \\
 & | & \text{trivial} \\
 & | & \texttt{Auto}\ \{ \\
 & & \quad \texttt{init}\ =\ [\overline{\textit{exp},}], \\
 & & \quad \texttt{kept}\ =\ [\overline{\textit{exp},}], \\
 & & \quad \texttt{pruned}\ =\ [\overline{\textit{exp},}]\} \\
 & & \\
\textit{exp} & ::= & \textit{Haskell expression} \\
\textit{pat} & ::= & \textit{Haskell pattern} \\
\textit{typ} & ::= & \textit{Haskell type (monomorphic)} \\
f, x & ::= & \textit{Haskell name} \\
 & & n \in \mathbb{N}
\end{array}
$$

Figure 3. Syntax of the proto-proof language

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 (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 = [a1, ..., aM]
     , kept = [b1, ..., bN]
     , pruned = _ }
```

is embedded into Haskell as:

```
a1 &&& ... &&& aM &&& b1 &&& ... &&& bN
```

$$
\begin{aligned}
\textit{decl-macro} &::= f :: \textit{typ} \\
&\qquad f\ \overline{y_i}\ =\ \overline{\textit{exp-macro};} \\[1em]
\textit{exp-macro} &::= \texttt{induct}\ x \\
&\quad \mid \texttt{destruct}\ \text{exp} \\
&\quad \mid \texttt{assert}\ \{\textit{exp}\} \\
&\quad \mid \texttt{dismiss}\ \{\textit{exp}\} \\
&\quad \mid \texttt{condition}\ \{\textit{exp}\} \\
&\quad \mid \texttt{auto}\ [\overline{x},]\ n \\
&\quad \mid \texttt{use}\ \{\textit{exp}\} \\
&\quad \mid \texttt{trivial} \\[1em]
\textit{exp} &::= \textit{Haskell expression} \\
\textit{typ} &::= \textit{Haskell type (monomorphic)} \\
f, x, y_i &::= \textit{Haskell name} \\
n &\in \mathbb{N}
\end{aligned}
$$

Figure 4. The proof macro language

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 `induct`s 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 an infinite loop. For example, in Figure 1, only `a'`, `b'`, and `c'` are in scope to use as arguments for the recursive call `assocMin a' b' c'`.

We formalize this expansion as a 4-place relation

$$\Gamma; \mathrm{P} \vdash T \rightsquigarrow t$$

which states that a proof macro T in the contexts Γ and P expands to a proto-proof term t. We allow this term t to contain holes that will be filled in by expansion of subsequent macros, but in potentially updated contexts Γ' and P'. We annotate each hole with these contexts, writing $\square_{\Gamma;\mathrm{P}}$. This expansion process is formalized in Figure 5, and we can broadly identify two types of proof macros: *control-flow* and *evidence* macros.

Control-Flow Macros. Control flow macros correspond to proof terms that alter the control flow of the program, such as pattern matching. The first five constructs from Figure 5 exhibit such functionality:

- `induct` x, `destruct` $\{exp\}$:
  Given a variable x that has type α in a context Γ, `induct` x creates a pattern match on x, with a branch for every constructor of α. The body of each branch is a hole $\square_{\Gamma_i;\mathrm{P}_i}$, with the typing context updated to include the (fresh) pattern variables $\{y_{ij}\}$ and their bindings, and with the recursion context updated to signify that any y_{ij} could be used to make terminating recursive calls. Similarly, given a well typed Haskell expression e with type α in a context Γ, `destruct` $\{e\}$ also creates a pattern match where each branch body is a hole. The only difference from `induct` is that it does not modify the recursion context, but gets to operate on arbitrary expressions. Figure 7 shows an extended step-by-step example of how the `induct` macro expands.
- `condition` $\{exp\}$, `assert` $\{exp\}$, `dismiss` $\{e\}$:
  These three macros all expand into `if` statements with the given boolean expression e as its condition. The difference lies in the holes produced in this expansion: `condition` creates a hole $\square_{\Gamma;\mathrm{P}}$ for both the `then` and the `else` branches; `assert` only creates such a hole for the `then` branch with the `else` branch being `trivial`; and `dismiss` is the dual of `assert`.

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.

- `trivial`:
  This macro expands into Liquid Haskell's `trivial` with type `Proof` in the resulting Haskell term. Since `trivial` = () and `Proof` = (), using this macro effectively means that the SMT solver can discharge any remaining obligations.
- `use` $\{exp\}$:
  This macro makes the refinement type of the expression available to the SMT solver using Liquid Haskell's conjunction &&&, similar to how, in Figure 1, a call to `assocMin a' b' c'` was needed to conclude the proof.
- `auto` $[\overline{x}]$ n:
  Finally, the `auto` macro is the core of our framework's automation. It takes two optional parameters, a sequence $\overline{x}$ of *hints*, and a natural number n, and it generates all well-typed neutral forms of type `Proof` up to height n that use variables from the current context or the hints. To ensure recursive calls are terminating,

$$\frac{\Gamma \vdash x : \alpha \qquad x \mapsto \varnothing \in \mathrm{P} \qquad \Gamma \vdash c_i : \overline{\beta_{ij}} \to \alpha \qquad \Gamma_i := (\Gamma \cup \{\overline{y_{ij} : \beta_{ij}}\}) \qquad \mathrm{P}_i := \mathrm{P}[x \mapsto \{\overline{y_{ij} : \beta_{ij}}\}]}{\Gamma; \mathrm{P} \vdash \texttt{induct}\ x \rightsquigarrow \texttt{case}\ x\ \texttt{of}\ \overline{c_i\ \overline{y_{ij}} \ \to\ \square_{\Gamma_i;\mathrm{P}_i};}}$$

$$\frac{\Gamma \vdash e : \alpha \qquad \Gamma \vdash c_i : \overline{\beta_{ij}} \to \alpha \qquad \Gamma_i := \Gamma \cup \{\overline{y_{ij} : \beta_{ij}}\}}{\Gamma; \mathrm{P} \vdash \texttt{destruct}\ \{e\} \rightsquigarrow \texttt{case}\ e\ \texttt{of}\ \overline{c_i\ \overline{y_{ij}} \ \to\ \square_{\Gamma_i;\mathrm{P}};}}$$

$$\frac{\Gamma \vdash e : \texttt{Bool}}{\Gamma; \mathrm{P} \vdash \texttt{condition}\ \{e\} \rightsquigarrow \texttt{if}\ e\ \texttt{then}\ \square_{\Gamma;\mathrm{P}}\ \texttt{else}\ \square_{\Gamma;\mathrm{P}}}$$

$$\frac{\Gamma \vdash e : \texttt{Bool}}{\Gamma; \mathrm{P} \vdash \texttt{assert}\ \{e\} \rightsquigarrow \texttt{if}\ e\ \texttt{then}\ \square_{\Gamma;\mathrm{P}}\ \texttt{else trivial}} \qquad \frac{\Gamma \vdash e : \texttt{Bool}}{\Gamma; \mathrm{P} \vdash \texttt{dismiss}\ \{e\} \rightsquigarrow \texttt{if}\ e\ \texttt{then trivial else}\ \square_{\Gamma;\mathrm{P}}}$$

$$\frac{}{\Gamma; \mathrm{P} \vdash \texttt{trivial} \rightsquigarrow \texttt{trivial}} \qquad \frac{\Gamma \vdash e : \texttt{Proof}}{\Gamma; \mathrm{P} \vdash \texttt{use}\ \{e\} \rightsquigarrow e\ \&\&\&\ \square_{\Gamma;\mathrm{P}}}$$

$$\frac{\Gamma \vdash x_i : \alpha_i \qquad n \in \mathbb{N} \qquad is = \mathrm{generate}(\Gamma \cup \{\overline{x_i}\}, \mathrm{P}, \texttt{Proof}, n)}{\Gamma; \mathrm{P} \vdash \texttt{auto}\ [\overline{x_i}]\ n \rightsquigarrow \texttt{Auto}\{init = is, kept = [], pruned = []\}\ \&\&\&\ \square_{\Gamma;\mathrm{P}}}$$

$$\begin{array}{lcl} \mathrm{generate}(\Gamma, \mathrm{P}, \alpha, 0) & = & \{\texttt{trivial}\} \\ \mathrm{generate}(\Gamma, \mathrm{P}, \beta, 1+n) & = & \{f\ a_1\ \cdots\ a_n \mid \Gamma \vdash f : \alpha_1 \to \cdots \to a_n \to \beta,\ a_i \in \mathrm{generate}(\Gamma, \mathrm{P}, \alpha_i, n)\} \\ & \cup & \{r\ a_1\ \cdots\ a_n \mid \Gamma \vdash r : \alpha_1 \to \cdots \to a_n \to \beta,\ a_i \in \mathrm{P}_i,\ x_i \mapsto \mathrm{P}_i \in \mathrm{P}\} \end{array}$$

Figure 5. Proof macro semantics

we keep track of a separate recursion context that is specially constructed in the rest of the proof.

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 `trivial`—as that has no effect on SMT resolution. To expand a sequence $T; Ts$ in some contexts Γ and P, we expand T to some proto-proof term t, and for every hole $\square_{\Gamma_i;\mathrm{P}_i}$ in t, we recursively expand the sequence 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 `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 `auto` [] 3 macro:

```
induct a; induct b; induct c; auto [] 3
```

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]

```
concatElemP :: N → [N] → [N] → Bool
concatElemP x xs ys
  | elem x xs = elem x (xs ++ ys)
  | otherwise = ⊤

{-@ concatElem :: x:N → xs:[N] → ys:[N] →
    {concatElemP x xs ys} @-}
[tactic|
concatElem :: N → [N] → [N] → Proof
concatElem x xs ys =
  assert {elem x xs};
  induct xs
|]
```

At a high level, what this proof macro does is condition on the expression `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

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

$$\frac{}{[\![\cdot]\!](\square_{\Gamma;\mathrm{P}}) = \texttt{trivial}}$$

$$\frac{\Gamma;\mathrm{P} \vdash T \rightsquigarrow t \qquad \square_{\Gamma_i;\mathrm{P}_i} \in t \qquad [\![Ts]\!](\square_{\Gamma_i;\mathrm{P}_i}) = t_i}{[\![T; Ts]\!](\square_{\Gamma;\mathrm{P}}) = t[t_i/\square_{\Gamma_i;\mathrm{P}_i}]}$$

Figure 6. Proof Macro Sequence Expansion

these macros back into the proto-proof language as shown in Figure 7.

Based on the declaration of `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 `auto [] 3` is appended at its end.

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

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:

$$\texttt{induct}\ e\ \texttt{as}\ [\overline{x_i}/\overline{y_i}/\cdots]$$
$$\texttt{destruct}\ e\ \texttt{as}\ [\overline{x_i}/\overline{y_i}/\cdots]$$

That is, we allow users to specify names for the variables introduced by different branches using the same syntax as Coq's Ltac [10]. However, given the strictly sequential nature of our proof macro language, this introduces a new problem: what happens when a macro refers to a variable that is only introduced in some branches? Potential but unsatisfactory solutions would include failing to expand (which is overly restrictive) or silently expanding to `trivial` (which would lead to accepting proof macros with typos). Instead, we opted to see this problem as an opportunity to explore a new point in the design space by introducing *conditional expansion*.

In particular, tactic languages in traditional proof assistants can follow a tree-like pattern. For example, in Coq, one can write:

```
t; [t1|t2|t3]
```

and that will execute the tactic `t`, followed by executing the tactics `t1`, `t2`, and `t3` in each of the three subgoals produced by `t`. Unfortunately, that can introduce a lot of repetition across similarly handled branches.

To counter this repetition, we allowed for optionally prepending variable name annotations to proof macros, which causes them to only be expanded when those variable names are in scope. So, for example, if a user wants to `induct` on a list `l :: [N]` and then `destruct` on the head of the list they can write the following:

```
induct l as [ / x xs];
[x:] destruct {x}
```

Moreover, by reusing names, more complicated expansion structures can be achieved. For instance, if we were dealing 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 `Seq` datatype:

```
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:

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

[3] Types of local bindings cannot be inferred with Template Haskell, so the type signature is in fact necessary.

$$\Gamma = \{\mathsf{x} : \mathsf{N},\ \mathsf{xs} : [\mathsf{N}],\ \mathsf{ys} : [\mathsf{N}]\}$$
$$\mathrm{P} = \{x \mapsto \emptyset,\ xs \mapsto \emptyset,\ ys \mapsto \emptyset\}$$

```
  ⟦assert {elem x xs}; induct xs; auto [] 3⟧(Γ; P)
= if elem x xs then
    ⟦induct xs; auto [] 3⟧(Γ; P)
  else trivial
= if elem x xs then
    case xs of
      [] → ⟦auto [] 3⟧(Γ; P)
      Cons x′ xs′ → ⟦auto [] 3⟧(Γ ∪ {x′ : N, xs′ : [N]}; P[xs ↦ {x′ : N, xs′ : [N]}])
  else trivial
= if elem x xs then
    case xs of
      [] → generate(Γ, P, Proof, 3) &&& trivial
      Cons x′ xs′ → generate(Γ ∪ {x′ : N, xs′ : [N]}, P[xs ↦ {x′ : N, xs′ : [N]}], Proof, 3) &&& trivial
  else trivial
= if elem x xs then
    case xs of
      [] → trivial &&& trivial
      Cons x′ xs′ →
        Auto
          { init =
            [ concatElem x′ xs′ xs′
            , concatElem x′ xs′ ys
            , concatElem x xs′ xs′
            , concatElem x xs′ ys ]
          , kept = []
          , pruned = [] } &&&
        trivial
  else trivial
```

Figure 7. Step-by-step Expansion of a Proof Macro. The holes $\square_{\Gamma;\mathrm{P}}$ that are generated and then immediately substituted by each step (see Figure 6) are omitted.

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 *exp* in its `init` field is attempted to be pruned one at a time. This is done by moving the *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 *exps* in the `init` fields of the `Auto` structures; otherwise this prune is undone, and the *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 `concatElem`. There is an `Auto` structure that clearly has a few *exps* that can be pruned since they are unnecessary for the proof. The subset of necessary *exps* is found via the linear pruning procedure, trying to remove each *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:

```
concatElem :: N → [N] → [N] → Proof
concatElem = \x xs ys →
  if elem x ys then
    case xs of
      Nil → trivial
```

```
      Cons x' xs' → concatElem x xs' ys
  else
    trivial
```

Note that this is the minimal proof a user would need to write in Liquid Haskell to convince its typechecker that `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 axis:

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 [13], 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* [24], 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× 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`, 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* [15], 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 [16], or Isabelle [17], tactics are the primary way by which users interactively manipulate the systems proof state. Tactics are usually written in a metalanguage 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 Mtac [26] that enforced a typing discipline, or Ltac2 [19] which provides more advanced metaprogramming capabilities [?]. Such tactics

[4] Measured from the signature of the top-level function to the last line of the macro sequence or proof term.

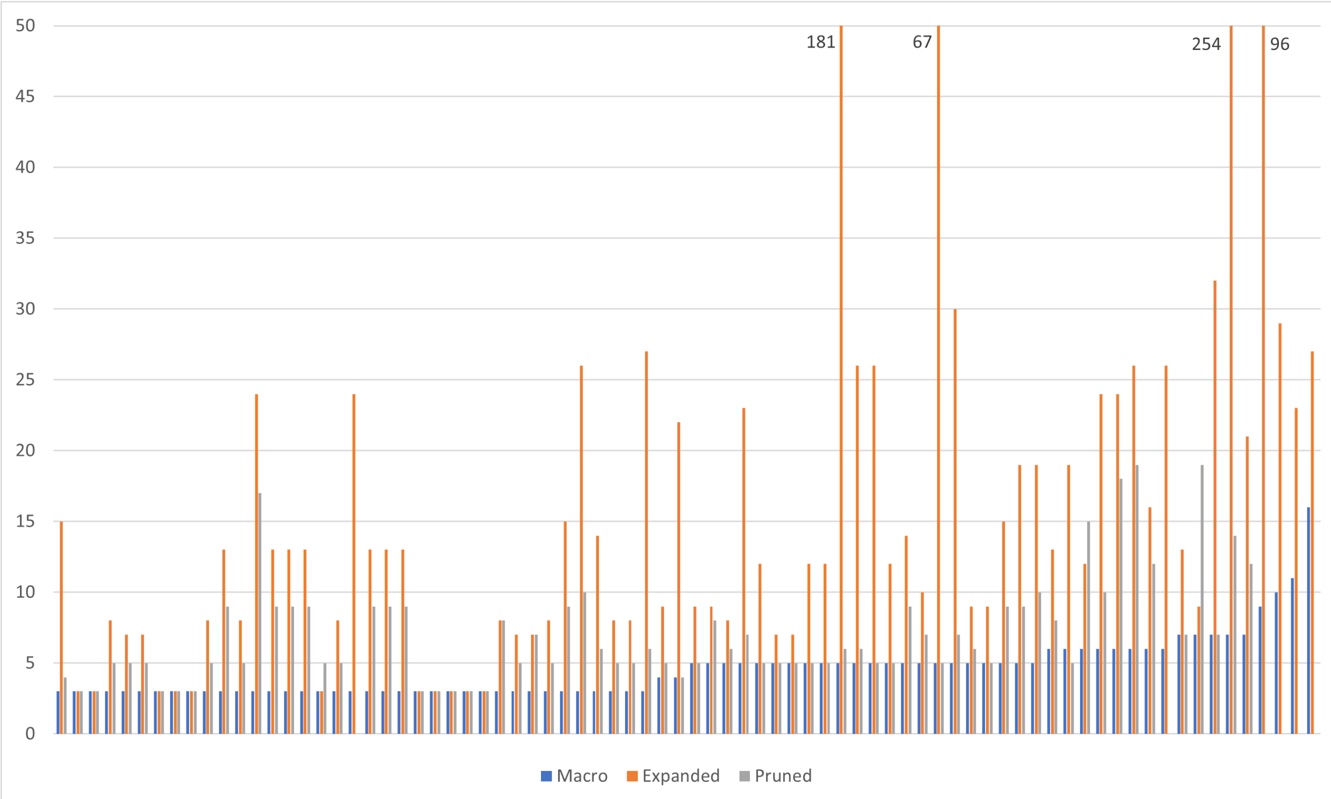


Figure 8. LoC across benchmarks of Liquid Proof Macros (in blue), the expanded splice (in orange), and the minimal Liquid Haskell proof term (in gray)

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.

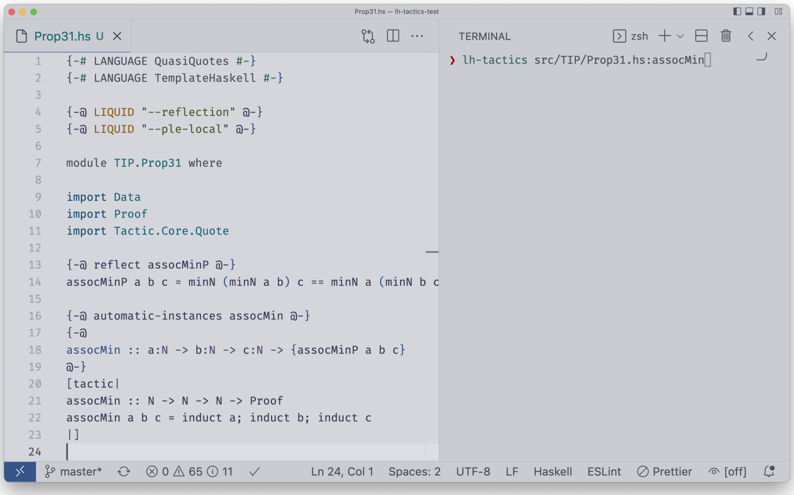
Moreover, hammers have been developed for proof assistants such as Coqhammer [7, 8] for Coq or Sledgehammer [3, 18] for Isabelle, which aim to bring the benefits of automated verification to the interactive setting. Hammers give the users the ability to directly discharge their current goal, but suffer 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.

Dependent Haskell. An alternative way of reasoning about Haskell programs is Dependent Haskell Eisenberg [11], which is currently under development (based on ghc-proposal#378). Yet, there is no current active proposal neither for proof automation nor for the development of a tactic language. Since the logic of both Liquid Haskell and Dependent Haskell is constructive, we conjecture that our Template Haskell-based design can be easily adjusted to generate Dependent Haskell proof terms.

Liquid Haskell Automation. Naturally, we are not the first to attempt automating Liquid Haskell proof generation. First, Vazou et al. [24] 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 [13] developed a quasiquoter 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.

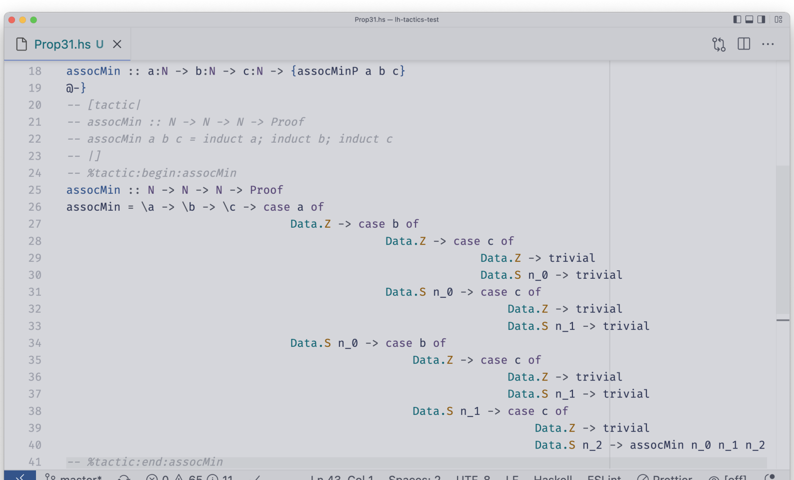
5 Conclusion

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



(a) The user writes the proof macro and runs the `lh-tactics` command line tool on the input file, which exists inside of a *stack* project that is configured to use the LiquidHaskell as a plugin.

(b) The user waits for the `lh-tactics` tool to complete. During this time, the tool will overwrite the input file on each pruning attempt.



(c) 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.

Figure 9. Usage example of the proof macros tool.

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,

```
refined-auto {assocMin (f m n) (f _ _)} [] 3
```

where `f :: N → N → N`, `m`, `n :: N`, would generate all neutral forms of the form `assocMin (f m n) (f l k)` where `l`, `k :: N` 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 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 can 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 an 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.

6 Data Availability Statement

This paper is accompanied by an artifact [**?**] that implements Liquid Proof Macros on top of Liquid Haskell, and allows for easy replication of the experiments in the evaluation section.

Acknowledgments

We thank Jacob Prinz, Michael Greenberg, and the anonymous reviewers for their helpful comments. This work is partially funded by the Horizon Europe ERC Starting Grant CRETE (GA: 101039196). This work was supported by NSF award #2107206, *Efficient and Trustworthy Proof Engineering* (any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF).

References

[1] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification*, Ganesh Gopalakrishnan and

Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 171–177.

[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. *The SMT-LIB Standard: Version 2.0.* Technical Report. Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[3] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. 2013. Extending Sledgehammer with SMT Solvers. *J. Autom. Reason.* 51, 1 (2013), 109–128. https://doi.org/10.1007/s10817-013-9278-5

[4] Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, Joshua Cohen, and Stephanie Weirich. 2021. Ready, Set, Verify! Applying hs-to-coq to real-world Haskell code. *J. Funct. Program.* 31 (2021), e5. https://doi.org/10.1017/S0956796820000283

[5] Koen Claessen. 2020. Finger Trees Explained Anew, and Slightly Simplified (Functional Pearl). In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell* (Virtual Event, USA) *(Haskell 2020)*. Association for Computing Machinery, New York, NY, USA, 31–38.

[6] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 268–279. https://doi.org/10.1145/351240.351266

[7] Lukasz Czajka. 2020. Practical Proof Search for Coq by Type Inhabitation. In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12167)*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer, 28–57. https://doi.org/10.1007/978-3-030-51054-1_3

[8] Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning* 61 (06 2018). https://doi.org/10.1007/s10817-018-9458-4

[9] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* 4963, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[10] David Delahaye. 2002. A Proof Dedicated Meta-Language. In *Logical Frameworks and Meta-Languages (LFM) (Electronic Notes in Theoretical Computer Science (ENTCS), Vol. 70(2))*, Frank Pfenning (Ed.). Elsevier, Copenhagen (Denmark), 96–109.

[11] Richard A. Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice.* Ph. D. Dissertation. University of Pennsylvania. https://arxiv.org/abs/1610.07978

[12] T. Freeman and F. Pfenning. 1991. Refinement Types for ML. In *PLDI*.

[13] Mustafa Hafidi. 2020. From testing to formal verification in Haskell. https://oa.upm.es/63616/

[14] K. Rustan M. Leino and Clément Pit-Claudel. 2016. Trigger Selection Strategies to Stabilize Program Verifiers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 361–381. https://doi.org/10.1007/978-3-319-41528-4_20

[15] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 2019. Meta-F ^\star : Proof Automation with SMT, Tactics, and Metaprograms. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science)*. Springer. https://doi.org/10.1007/978-3-030-17184-1_2

[16] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 625–635.

[17] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media.

[18] Lawrence C. Paulson and Jasmin Christian Blanchette. 2010. Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers. In *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011 (EPiC Series in Computing, Vol. 2)*, Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska (Eds.). EasyChair, 1–11. https://doi.org/10.29007/36dt

[19] Pierre-Marie Pédrot. 2019. Ltac2: Tactical Warfare. The Fifth International Workshop on Coq for Programming LanguagesCoqPL. https://www.pédrot.fr/articles/coqpl2019.pdfpdf

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

[21] The Coq Development Team. 2020. *The Coq Proof Assistant, version 8.11.0.* https://doi.org/10.5281/zenodo.3744225

[22] N. Vazou. 2016. *Liquid Haskell: Haskell as a theorem prover.* Ph. D. Dissertation. UCSD.

[23] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. 2018. Theorem Proving for All: Equational Reasoning in Liquid Haskell (Functional Pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. ACM, New York, NY, USA, 132–144. https://doi.org/10.1145/3242744.3242756

[24] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement reflection: complete verification with SMT. *PACMPL* 2, POPL (2018), 53:1–53:31. https://doi.org/10.1145/3158141

[25] Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen, and Dan Rosén. 2013. HALO: Haskell to Logic through Denotational Semantics. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 431–442. https://doi.org/10.1145/2429069.2429121

[26] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2015. Mtac: A monad for typed tactic programming in Coq. *Journal of Functional Programming* 25 (2015).