Liquid Proof Macros

Anonymous Author(s)

Abstract

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

24

25

26

30

31

32

33

34

35

36

37

39

40

41

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.

22 CCS Concepts: • Software and its engineering \rightarrow For-23 mal software verification.

Keywords: Liquid Haskell, proof macros, tactics

ACM Reference Format:

Anonymous Author(s). 2018. Liquid Proof Macros. In Proceedings 27 of ACM Conference (Conference'17). ACM, New York, NY, USA, 28 12 pages. https://doi.org/XXXXXXXXXXXXXXXXX 29

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. 38

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

```
42
         data N = Z | S N
43
44
         min :: N \rightarrow N \rightarrow N
45
         min Z = Z
46
         min _Z = Z
47
         min (S m) (S n) = S (min m n)
48
```

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

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

52 Conference'17, July 2017, Washington, DC, USA

```
53
      2018. ACM ISBN 978-x-xxxx-x/YY/MM...$15.00
```

```
54
```

```
55
```

49

50

51

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 \rightarrow b:N \rightarrow c:N \rightarrow
      \{ : () \mid \min(\min a b) c == \min a (\min b c) \}
@-}
assocMin :: N \rightarrow N \rightarrow N \rightarrow 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 [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

108

109

110

70

71

72

73

74

75

76

56

57

58

59

60

111	$\{-@ assocMin :: a:N \rightarrow b:N \rightarrow c:N \rightarrow$
112	{min (min a b) c == min a (min b c)} @-}
113	assocMin :: N \rightarrow N \rightarrow N \rightarrow Proof
114	assocMin = $a b c \rightarrow$
115	case a of
116	$Z \rightarrow$
117	case b of
118	$Z \rightarrow$
119	case c of
120	$Z \rightarrow trivial$
121	S c' \rightarrow trivial
122	S b' \rightarrow
123	case c of
124	$Z \rightarrow trivial$
125	S c' \rightarrow trivial
126	S a' \rightarrow
127	case b of
128	$Z \rightarrow$
129	case c of
130	$Z \rightarrow trivial$
131	S c' \rightarrow trivial
132	S b' \rightarrow
133	case c of
134	$Z \rightarrow trivial$
135	S c' \rightarrow assocMin a' b' c'
136	
137	

Figure 1. Liquid Haskell proof term for associativity of min

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

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 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 \rightarrow N \rightarrow N \rightarrow Proof
assocMin a b c = induct a; induct b; induct c
[]

Figure 2. Associativity of min using Liquid Proof Macros

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 [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 intutive proofs that resemble automated tactics¹ 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.

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. 166 167

174 175 176

177

178

179

180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

219

¹We refrain calling our DSL tactics, as that suggests a notion of interactivity that is impossible in the current version of Liquid Haskell.

223

224

225

226

227

228

229

230

231

232

233

234

235

236

237

238

239

240

241

263

- 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.

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.

264 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

decl-proto f ·· typ	276
	277
f = exp-proto;	278
exp -proto $::= \lambda$ name $\rightarrow exp$ -proto	279
case exp of $\overline{pat \rightarrow exp-proto}$;	280
if <i>exp</i> then <i>exp-proto</i> else <i>exp-proto</i>	281
exp &&& exp-proto	283
trivial	284
L Auto (285
	286
$init = [\overline{exp},],$	287
$kept = [\overline{exp},],$	288
pruned = $[\overline{exp},]$	289
	290
avp Haskell expression	291
exp Huskell expression	292
pat ::= Haskell pattern	293
typ ::= Haskell type (monomorphic)	294
f, x ::= Haskell name	295
$n \in \mathbb{N}$	296
$n \in \mathbb{N}$	297
	298

Figure 3. Syntax of the proto-proof language

299

300

301

302

303

304

305

306

307

308

309

310

311

312

313

314

315

316

317

318

319

320

321

322

323

324

325

326

327

328

329

330

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

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 toplevel declaration to a (potentially empty) set of subterms that can be used to instantiate recursive calls without triggering

331	deal maara f tup
332	
333	$f \overline{y_i} = \overline{exp}-macro;$
334	
335	exp-macro ::= induct x
336	destruct exp
337	assert exp
338	dismiss exp
340	condition exp
341	auto $[\overline{x},]$ n
342	lise ern
343	
344	
345	
346	exp ::= Haskell expression
347	typ ::= Haskell type (monomorphic)
348	$f, x, y_i ::= Haskell name$
349	$n \in \mathbb{N}$
251	
352	Figure 4. The proof macro language
353	rigure 4. The proof macro language
354	
355	an infinite loop. We formalize this expansion as a 4-place
356	relation
357	$\Gamma; \mathbb{P} \vdash T \rightsquigarrow t$
358	which states that a proof macro T in the contexts Γ and
359	P expands to a proto-proof term t . We allow this term t to
360	contain holes that will be filled in by expansion of subsequent

³⁵⁹ P expands to a proto-proof term *t*. We allow this term *t* to ³⁶⁰ contain holes that will be filled in by expansion of subsequent ³⁶¹ macros, but in potentially updated contexts Γ' and P'. We ³⁶² annotate each hole with these contexts, writing $\Box_{\Gamma;P}$. This ³⁶³ expansion process is formalized in Figure 5, and we can ³⁶⁴ broadly identify two types of proof macros: *control-flow* and ³⁶⁵ *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 *e*:

Given a variable x that has type α in a context Γ , 372 induct x creates a pattern match on x, with a branch 373 for every constructor of α . The body of each branch is a 374 375 hole $\Box_{\Gamma_i;P_i}$, with the typing context updated to include the (fresh) pattern variables $\{y_{ij}\}$ and their bindings, 376 377 and with the recursion context updated to signify that any y_{ii} could be used to make terminating recursive 378 379 calls. Similarly, given a well typed Haskell expression *e* with type α in a context Γ , destruct *e* also creates a 380 pattern match where each branch body is a hole. The 381 382 only difference from **induct** is that it does not modify the recursion context, but gets to operate on arbitrary 383 expressions. 384

386

387

388

389

390

391

392

393 394

395

396

397

398

399

400

401

402

403

404

405

406

407

408

409

410

411

412

413

414

415

416

417

418

419

420

421

422

423

424

425

426

427

428

429

430

431

432

433

434

435

436

437

438

439

440

• condition *e*, assert *e*, 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 $\Box_{\Gamma;P}$ 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, 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 $\Box_{\Gamma_i;P_i}$ 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

385

366

367

368

369

370

Liquid Proof Macros

441	$\Gamma \vdash x : \alpha \qquad x \mapsto \emptyset \in \mathbb{P} \qquad \Gamma \vdash c_i : \beta_{ij} \to \alpha$					
442	$\Gamma_i := (\Gamma \cup \{\overline{y_{ii} : \beta_{ii}}\}) \qquad P_i := P[x \mapsto \{\overline{y_{ii} : \beta_{ii}}\}]$					
443						
444	$\Gamma; P \vdash \text{induct } x \rightsquigarrow \text{ case } x \text{ of } c_i \overline{y_{ij}} \rightarrow \Box_{\Gamma_i; P_i};$					
445						
446	$\Gamma \vdash e : \alpha \qquad \Gamma \vdash$	$+ c_i : \beta_{ij} \rightarrow \alpha$	501			
447	$\Gamma_i := \Gamma \cup \{\overline{y_{ij} : \beta_{ij}}\}$					
448			503			
449	1;PF destruct e vi ca	ise e of $c_i y_{ij} \rightarrow \Box_{\Gamma_i;\mathbb{P}};$	504			
450	$\Gamma \vdash \rho$	Bool	505			
451	$\Gamma \cdot P \vdash condition e consistent$	f e then then else then	506			
452			507			
453	$\Gamma \vdash e : \texttt{Bool}$	$\Gamma \vdash e : \texttt{Bool}$	508			
454	$\Gamma; P \vdash assert e \rightsquigarrow if e then \Box_{\Gamma:P} else trivial$	$\Gamma; P \vdash \text{dismiss } e \rightsquigarrow \text{if } e \text{ then trivial else } \Box_{\Gamma:P}$	509			
455			510			
456		$\Gamma \vdash e : Proof$	511			
457	Γ;P⊢ trivial ↔ trivial	$\Gamma; \mathbf{P} \vdash use \{ e \} \rightsquigarrow e \&\&\& \Box_{\Gamma;\mathbf{P}}$	512			
458			513			
459	$\Gamma \vdash x_i : \alpha_i \qquad n \in \mathbb{N} \qquad is = ge$	$enerate(\Gamma \cup \{\overline{x_i}\}, P, Proof, n)$	514			
460	$\Gamma; P \vdash auto [\overline{x_i}] n \rightsquigarrow Auto{init = is}$	$\overline{s, kept = [], pruned = []} \&\&\& \Box_{\Gamma:P}$	515			
461			516			
462	generate($\Gamma, P, \alpha, 0$) = {trivial}		517			
463	generate $(\Gamma, P, \beta, 1+n) = \{f a_1 \cdots a_n \mid \Gamma \vdash f\}$	$f: \alpha_1 \to \cdots \to a_n \to \beta, \ a_i \in \text{generate}(\Gamma, \mathbb{P}, \alpha_i, n) \}$ $: \alpha_1 \to \cdots \to a_n \to \beta, \ a_i \in \mathbb{P}_i, \ x_i \mapsto \mathbb{P}_i \in \mathbb{P} \}$				
464	$\bigcup \{ra_1 \cdots a_n \mid \Gamma \vdash r :$					
465						
466	Figure 5. Proof macro semantics					
467	0		522			
468			523			
469	Extended Example. For concreteness, consider the fol-		524			
470	lowing predicate which states that if a number x is an el-	$\llbracket \cdot \rrbracket(\Box_{\Gamma;P}) = trivial$	525			
471	ement of a list xs, then x is also in the list xs++ys for an	$\Gamma \cdot P \vdash T \rightsquigarrow t$	526			
472	arbitrary list ys, along with a corresponding Liquid Haskell	$\Box_{\mathbf{D},\mathbf{D}} \in t [T_{\mathbf{S}}](\Box_{\mathbf{D},\mathbf{D}}) = t$	527			
473	theorem that is proved with a short Liquid Proof macro. ²	$\frac{\Box_{I_i;P_i} \subset \iota \qquad [I] \Box_{I_i;P_i} - \iota_i}{\Box_{I_i;P_i} = \Box_{I_i}}$	528			
474	$concatElemP :: N \rightarrow [N] \rightarrow [N] \rightarrow Boo]$	$\llbracket T; Ts \rrbracket (\Box_{\Gamma; P}) = t \llbracket t_i / \Box_{\Gamma_i; P_i} \rrbracket$	529			
475	concatElemP x xs vs					
476	elem x xs = elem x (xs ++ vs)	Figure 6. Proof Macro Sequence Expansion	531			
477	$\int dterwise = T$					
478			533			
479		these macros back into the proto-proof language as shown	534			

{-@ concatElem :: $x:N \rightarrow xs:[N] \rightarrow ys:[N] \rightarrow$ {concatElemP x xs ys} @-} [tactic| $\texttt{concatElem} \ \colon \mathsf{N} \ \rightarrow \ [\mathsf{N}] \ \rightarrow \ [\mathsf{N}] \ \rightarrow \ \texttt{Proof}$ concatElem x xs ys = assert {elem x xs}; induct xs 1]

At a high level, what this proof macro does is condition 488 on the expression elem x xs, pattern match on xs, and 489 search for ways to complete the proof, potentially using the 490 tail of xs for a recursive call. This is achieved by expanding 491 492

480

481

482

483

484

485

486

487

in Figure 7.

535

536

537

538

539

540

541

542

543

544

545

546

547

548

549

550

Based on the declaration of concatElem inside the quasiquoter, 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 $\Box_{\Gamma;P}$, which will be filled by expanding the rest of the sequence, beginning with induct xs.

⁴⁹³ ²For the sake of readability, we used the usual list notations such as (++) 494 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.

551		$\Gamma = \{x \in \mathbb{N} \mid x \in [\mathbb{N}] \mid x \in [\mathbb{N}]\}$
552		$I = \{X : N, XS : [N], YS : [N]\}$
553		$\mathbf{P} = \{ x \mapsto \emptyset, \ xs \mapsto \emptyset, \ ys \mapsto \emptyset \}$
554		$\begin{bmatrix} 1 \\ 1 \\ 2 \\ 2$
555	_	<pre>[assert {elem x xs}; induct xs; auto [] 3](1;P) if alam x vs then</pre>
556	=	$\begin{bmatrix} \text{induct versus of } D \\ 2 \end{bmatrix} (\Gamma, D)$
557		
558		else trivial
559	=	If elem x xs then
560		
561		$[] \rightarrow [[auto]] 3](1; P)$
562		$\operatorname{Cons} x' xs' \rightarrow [[\operatorname{auto} [] 3]](1 \cup \{x' : N, xs' : [N]\}; P[xs \mapsto \{x' : N, xs' : [N]\}])$
563		
564	=	If elem x xs then
565		
566		$[] \rightarrow \text{generate}(1, P, Proof, 3) \&\&\& \text{ trivial}$
567		$Cons x^* xs^* \rightarrow generate(1 \cup \{x^* : N, xs^* : [N]\}, P[xs \mapsto \{x^* : N, xs^* : [N]\}], Proof, 3) \&a\& trivial$
568		
569	=	If elem x xs then
570		case xs of
571		$[] \rightarrow \text{trivial} \&\&\& \text{trivial}$
572		$Cons x' xs' \rightarrow$
573		Auto
574		$\{1n1t = 1, \dots, n, n, n, n\}$
575		[CONCATELEM X' XS' XS'
576		, concatElem x' xs' ys
577		, concatElem x xs' xs'
578		, concatElem x xs' ys]
579		, kept = []
580		, pruned = [] } &&&
581		
582		else trivial
583		
584		Figure 7. Step-by-step Expansion of a Proof Macro

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'
601
        • concatElem x' xs' ys
602
        • concatElem x xs' xs'
603
        • concatElem x xs' ys
604
605
```

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.

Variables and Conditional Expansion 2.3

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:

662

663

664

679

680

681

682

683

684

691

692

693

694

695

696

697

698

699

700

701

702

703

704

705

710

711

induct e as $[\overline{x_i}/\overline{y_i}/\cdots]$
destruct <i>e</i> as $[\overline{x_i}/\overline{y_i}/\cdots]$

665 That is, we allow users to specify names for the variables 666 introduced by different branches using the same syntax as 667 Coq's Ltac [10]. However, given the strictly sequential nature 668 of our proof macro language, this introduces a new problem: 669 what happens when a macro refers to a variable that is only 670 introduced in some branches? Potential but unsatisfactory 671 solutions would include failing to expand (which is overly 672 restrictive) or silently expanding to trivial (which would 673 lead to accepting proof macros with typos). Instead, we opted 674 to see this problem as an opportunity to explore a new point 675 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 1 :: [N] and then destruct on the head of the list they can write the following:

induc	ct 1	as	Ε	/	х	xs]
[x:]	des	tru	ct	х		

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
```

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.

728

729

730

731

732

733

734

735

736

737

738

739

740

741

742

743

744

745

746

747

748

749

750

751

752

753

754

755

756

757

758

759

760

761

762

763

764

765

766

767

768

769

770

For each Auto structure in a proto-proof term, each *exp* 716 in its init field is attempted to be pruned one at a time. 717 This is done by moving the *exp* from the init field to the 718 pruned field, embedding and splicing the new proto-proof 719 term into the original Haskell file in place of the original 720 proof macro, and then running Liquid Haskell to check if this 721 prune was safe. If it was, pruning continues with the rest of 722 the *exp*s in the init fields of the Auto structures; otherwise 723 this prune is undone, and the *exp* that was attempted to be 724 pruned is instead moved to the kept field before continuing 725 pruning. This process is very similar to shrinking in the style 726 of QuickCheck [6] from the property-based testing literature. 727

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 :: $N \rightarrow [N] \rightarrow [N] \rightarrow Proof$
concatElem = $x xs ys \rightarrow$
if elem x ys then
case xs of
Nil \rightarrow trivial
Cons x' xs' \rightarrow 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







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)

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× 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,

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

Liquid Proof Macros

∑ zsh + ∨ ⊟ 前 < ×</p>

S n_0 -> case c of Z -> trivial S n_1 -> trivia

case b of Z -> case c of Z -> trivial S n_1 -> trivia

____ > case c of Z -> trivial S n_2 -> ((asso

S n_1

[console] wrote inlined splices in file: /Users/he nry/Documents/Research/LiquidHaskell/liquidHaskell -metapogramming/h-tatics-test/src/TIP/ProJ31. [console] pruning autos in module */Users/henry/Do cuments/Research/LiquidHaskell/tiquidHaskell-metag

rogramming/lh-tactics-test/src/TIP/Prop31.hs" in d efinition "assocMin"



Prop31.hs U ×

assocMin

-> \ b

assocMin :: N -> N -> N -> Proof

b -> \ c -> case a of 7 -> ca

Z -> case b of

S n_0

S n_0

-> case b of

S n_1

Z -> case c of

-> case c of

Z -> case c of

-> case c of

Z -> trivial

S n_0 -> trivial

-> trivial

S n_1 -> trivial

> trivial

S n_1 -> trivial

Z -> trivial

(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.

TERMINAL

cMin n_0) n_1) n_2 -- %tactic:end:assocMin

(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.

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

994

995

1030

1031

1032

1033

1034

1035

1036

1037

1038

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], 1001 the tactics and metaprogramming framework for the F* lan-1002 guage [20]. In this work, Martínez et al. face the same issues 1003 that Liquid Haskell users face: how to reconcile the auto-1004 matic but black-box nature of SMT-aided program verifiers 1005 with the expressive tactic-based facilities of interactive the-1006 orem provers. Their approach is similar in nature to ours, 1007 but enjoys the benefit of developing the metaprogramming 1008 framework with the particular use case of verification in 1009 mind. In contrast, we showed how one can work within the 1010 already established constraints and limitations of the Haskell 1011 ecosystem, using Template Haskell to provide a more stream-1012 lined user experience. 1013

1014 Interactive Tactics. In the land of interactive proof assis-1015 tants like Coq [21], Lean [15], or Isabelle [16], tactics are the 1016 primary way by which users interactively manipulate the 1017 systems proof state. Tactics are usually written in a meta-1018 language that is built with the explicit purpose of developing 1019 proofs, often evolving along the proof assistant. For exam-1020 ple, Coq's tactic language Ltac [10] has been the target of 1021 multiple enhancement attempts, such as Mtac [27] that en-1022 forced a typing discipline, or Ltac2 [19] which provides more 1023 advanced metaprogramming capabilities [17]. Such tactics 1024 operate on the underlying representation of the proof state 1025 in a proof assistant, and are therefore inherently more ex-1026 pressive in the capabilities they provide. In this work, we 1027 drew inspiration from the kind of reasoning that tactics al-1028 low for, to give Liquid Haskell users the ability to write more 1029 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

1046

1047

1048

1049

1050

1051

1052

1053

1054

1055

1056

1057

1058

1059

1060

1061

1062

1063

1064

1065

1066

1067

1068

1069

1070

1071

1072

1073

1074

1075

1076

1077

1078

1079

1080

1081

1082

1083

1084

1085

1086

1087

1088

1089

1090

1091

1092

1093

1094

1095

1096

1097

1098

1099

1100

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 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 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 $f :: N \rightarrow N \rightarrow N, m, n :: N$, 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.

1104 1105

Engineering Challenges. In addition to the simple ex-1106 tensions described above, our framework could currently be 1107 improved with some investment in a non-trivial but straight-1108 forward engineering effort. In particular, the user interface to 1109 1110 Liquid Haskell has been developed into a plugin that works in tandem with the Haskell stack build system. Currently, 1111 the proof macro system requires the user to run an external 1112 tool on proof macros for pruning purposes. User experience 1113 would be greatly improved if the proof macro system was 1114 integrated into the existing Liquid Haskell plugin, and run 1115 automatically when the project is built. 1116

Similarly, Template Haskell splices code implicitly during 1117 compilation, in such a way that the splices are never actually 1118 displayed inline with the user's original code. Currently, 1119 1120 Template Haskell is not well-supported by Liquid Haskell, 1121 and our external tool explicitly splices the pruned code in for efficiency purposes. It would be interesting to further explore 1122 this interaction between Template and Liquid Haskell to see 1123 if can can get the best of both worlds: the conciseness of 1124 1125 proof macros with the efficient compilation of the pruned 1126 proof terms.

Moreover, the current auto macro cannot handle polymor-1127 phism, because as Template Haskell only provides support 1128 for syntactic equality when checking if an value's type is 1129 compatible with the type expected for an argument in a neu-1130 1131 tral form being generated. Supporting polymorphism would 1132 require writing a simple unification function at the Template Haskell level, which would fit nicely with the rest of our 1133 framework. 1134 1135

1136

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 *exps* 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 *exps* 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.

1156

1157

1158

1159

1160

1161

1162

1163

1164

1165

1166

1167

1168

1169

1170

1171

1172

1173

1174

1175

1176

1177

1178

1179

1180

1181

1182

1183

1184

1185

1186

1187

1188

1189

1190

1191

1192

1193

1194

1195

1196

1197

1198

1199

1200

1201

1202

1203

1204

1205

1206

1207

1208

1209

1210

References

- 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, IJ-CAR 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] T. Freeman and F. Pfenning. 1991. Refinement Types for ML. In PLDI.
- [12] Mustafa Hafidi. 2020. From testing to formal verification in Haskell. https://oa.upm.es/63616/
- [13] 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
- [14] 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
- [15] 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

1211 Publishing, Cham, 625–635.

- [16] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. Is *abelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media.
- [17] Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. 2022.
 Computing Correctly with Inductive Relations. In *Proceedings of the* ACM SIGPLAN Symposium on Programming Language Design and
- 1217 Implementation (PLDI).[18] Lawrence C. Paulson and Jasmin Christian Blanchette. 2010. Three
- 1218 [18] Lawrence C. Paulson and Jasmin Christian Bianchette. 2010. Three years of experience with Sledgehammer, a Practical Link Between
- 1219
 Automatic and Interactive Theorem Provers. In The 8th International

 1220
 Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, In

 1221
 donesia, October 9, 2011 (EPiC Series in Computing, Vol. 2), Geoff Sut

 1222
 cliffe, Stephan Schulz, and Eugenia Ternovska (Eds.). EasyChair, 1–11.
- 1222 cliffe, Stephan Schulz, and Eugenia Ternovska (Eds.). EasyChair, 1–11.
 1223 https://doi.org/10.29007/36dt
 [19] Pierre-Marie Pédrot. 2019. Ltac2: Tactical Warfare. The Fifth In-
- 1224 [17] Herre-Marie Feuror. 2019. Ltacz: factical warrare. 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, Leonidas Lampropoulos, and Jeff Polakow. 2017. A Tale of Two Provers: Verifying Monoidal String Matching in Liquid Haskell and Coq. In Proceedings of the ACM SIGPLAN International Symposium on Haskell.
- [25] 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
- [26] 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
- [27] 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).