Neurosymbolic Modular Refinement Type Inference

Georgios Sakkas
UC San Diego
gsakkas@ucsd.edu

Pratyush Sahu
UC San Diego
psahu@ucsd.edu

Kyeling Ong
UC San Diego
k8ong@ucsd.edu

Ranjit Jhala
UC San Diego
rjhala@ucsd.edu

***Abstract*—Refinement types, a type-based generalization of Floyd-Hoare logics, are an expressive and modular means of statically ensuring a wide variety of correctness, safety, and security properties of software. However, their expressiveness and modularity means that to use them, a developer must laboriously *annotate* all the functions in their code with potentially complex type specifications that specify the contract for that function. We present LHC, a neurosymbolic agent that uses LLMs to automatically generate refinement type annotations for all the functions in an entire package or module, using the refinement type checker LIQUIDHASKELL as an oracle to verify the correctness of the generated specifications. We curate a dataset of three Haskell packages where refinement types are used to enforce a variety of correctness properties from data structure invariants to low-level memory safety and use this dataset to evaluate LHC. Previously these packages required expert users several days to weeks to annotate with refinement types. Our evaluation shows that even when using relatively smaller models like the 3 billion parameter StarCoder LLM, by using fine-tuning and carefully chosen contexts, our neurosymbolic agent generates refinement types for up to 94% of the functions across entire libraries automatically in just a few hours, thereby showing that LLMs can drastically shrink the human effort needed to use formal verification.**

I. INTRODUCTION

Refinement types are a type-based generalization of Floyd-Hoare logics, where the programmer can specify correctness requirements by decorating classical types (*e.g.*`Int`) with *logical predicates* (*e.g.*`0 <= v`) that provide additional constraints on the values that can inhabit the type, thereby providing a *modular* and *expressive* means of statically enforcing a wide variety of correctness, safety, and security properties of software. Refinement types have been developed for various languages, from the ML family [1]–[4], to C [5]–[7], Ruby [8], Rust [9], [10], TypeScript [11], Scala [12], Solidity [13], Racket [14]. A recent paper presented a user study of 30 developers using refinement types for Java [15] that concluded that "LiquidJava helped users detect and fix more bugs, and that Liquid (Refinement) Types are easy to interpret and learn with few resources."

Sadly, as with other expressive and modular program verification tools like ESCJava [16] or Dafny [17], the wider usage of refinement types is hindered by the fact that to effectively use refinement types across their codebase, developers must laboriously *annotate* all the functions in their code with potentially complex type specifications that specify the behavior of that function to the rest of the code. The expressiveness of refinement contracts means that (unlike in classical type systems where often a type can be uniquely determined from the code) there is an infinite space of possible specifications for each function, which makes it tricky for the developer to determine the right one. The problem is exacerbated by modularity which means that the refinement type or contract specified for a function `f` may be "correct" for `f` in isolation, but may not suffice to verify `f`'s *clients*, and so the developer has to go back and forth changing the annotations of functions to get the entire codebase to verify.

In this paper, we present LHC[1], a neurosymbolic agent that uses *large language models* (LLMs) to automatically generate refinement type annotations for the functions in an entire codebase, using the refinement type checker LIQUIDHASKELL as an oracle to verify the correctness of the generated specifications. We develop our approach via three contributions.

1. Agent Our main contribution is an agent that systematically traverses the codebase's call-graph to generate each function's refinement type annotation. If we think of the refinement type annotation as the analog of a procedure *summary*, then we can think of our agent as a neurosymbolic program analysis, that combines a "bottom-up" analysis which uses neural LLMs to generate refinement type annotations (summaries) for functions, with a "top-down" analysis that kicks in when the LLM fails to generate correct types, that instead uses a symbolic predicate abstraction technique to generate refinement types from predicate *templates* obtained from the failed LLM predictions. Thus, even where the LLM fails to generate the correct type, its predictions can be used to generate an abstract domain that allows the symbolic analysis to succeed.

2. Dataset Our second contribution is a dataset comprising three Haskell packages: a suite of programs which are part of a tutorial on refinement types, a Haskell implementation of the Salsa20 cipher, and a widely used library that implements Byte-Strings with low-level pointer operations. The dataset includes a diverse set of functions, totalling about 5KLoC annotated with refinement types that enforce a variety of correctness properties ranging from data structure invariants to low-level memory safety. This dataset was curated to deliberately *exclude* code present in the popular open-source code LLM training dataset *The Stack* [18], [19], to ensure that successful type generation is not simply due to memorization.

3. Evaluation Our final contribution is an evaluation of LHC on our dataset, using a variety of pre-trained LLMs, including StarCoder and CodeLlama which were *not* trained on the code

[1]Stands for *Liquid Haskell Copilot* or LHCOPILOT

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00090

in our dataset. We demonstrate that by fine-tuning these LLMs on a small set of about 9,000 LIQUIDHASKELL programs, we can greatly improve the agent. We show how by combining the bottom-up generation of the neural models with top-down symbolic inference using qualifiers from the LLMs predictions, LHC can automatically generate refinement types for up to 94% of the functions across entire libraries. Furthermore, the entire generation process can be completed in just a few hours, a significant improvement over the several days or weeks of human effort that originally went into annotating the packages, thereby indicating that LLMs can drastically shrink the human effort needed to use formal verification.

II. BACKGROUND

We start with some preliminaries showing how refinement types can be used to specify and verify properties of programs § II-A, and how LLMs can be used to automatically generate the type annotations required for verification § II-B.

A. Refinement Type Checking with LIQUIDHASKELL

Specification Refinement type checkers like LIQUIDHASKELL let the programmer specify correctness requirements decorating classical types with *logical predicates* — typically drawn from an SMT-decidable theory — which provide additional constraints on the values that can inhabit the type. A refined *base* type of the form `{v:T|p(v)}` defines the set of values `v` of type `T` such that additionally, the constraint `p(v)` is true of the value `v`. For example, the type `{v:Int | 0 <= v}` specifies the set of *non-negative* integer values. A refined *function* type of the form `x:{In|pre(x)} -> {v:Out|post(v, x)}` can specify pre- and post-conditions for the underlying functions via constraints on the `Input` and `Output` types. For example, the type `x:{Int|0 <= x} -> {v:Int|v >= x}` specifies a function that requires non-negative inputs, and ensures that the returned value is at least as large as the input `x`.

Refinement type checkers also allow the programmer to specify properties of data using *measure* functions [3], [20], which are pure and total functions that map data types (such as lists, trees, *etc.*) to SMT-decidable values (such as integers, booleans, sets *etc.*). For example, the measure `notEmp` defines a boolean predicate on lists that is true if the list is non-empty:

```
measure notEmp :: [a] -> Bool
  notEmp []    = False
  notEmp (_:_) = True
```

and we can use it to specify that a particular function should only be called with non-empty lists

```
{-@ head :: {v:[a] | notEmp v} -> a @-}
head (x:_) = x
head []    = error "empty list" -- runtime crash
```

Verification Refinement type checkers like LIQUIDHASKELL verify the specifications by generating *verification conditions* (VCs) — logical formulas whose validity, determined by an SMT solver [21], ensures that the program is type-safe. For example, consider the code for the `head` function shown above, and assume that `error` — which aborts the program with a run-time panic — is a library function that is given the type

```
{-@ error :: {v:String | False} -> a @-}
```

That is, the precondition of `error` says it can only be called with `String` messages such that the predicate `False` holds. Since there are no such `String`s, the program will only verify if at compile-time, the refinement type checker can prove that `error` is never actually called. LIQUIDHASKELL verifies the code for `head` by generating the VC:

$$\forall v.\ notEmp(v) \ \Rightarrow\ \neg notEmp(v) \ \Rightarrow\ False$$

The first antecedent comes from the precondition that the input list is a non-empty list, the second antecedent comes from the fact that in the second case (where we call `error`) the input list is matched against `[]` whose measure is `False`, and the consequent *False* arises from the pre-condition of `error`. The SMT solver proves the above VC valid to verify that `head` will never crash on non-empty lists.

Modularity and Annotations Refinement type checking is *modular* in that when we check a client (*e.g.*`head`) that calls a function (*e.g.* `error`) the only information known about the callee is its *type signature*. This means that to analyze an entire package or module, the programmer must *annotate* all the functions of the module with (refinement) type signatures.

For example, consider the code in Figure 1 which shows a small Haskell module that implements a function that computes the `average` of a list of integers by computing the `sum` of the integers and then invoking `divide` with the `size` of the list. The `divide` function panics with `error` when the divisor is `0`, and otherwise calls the mathematical `div` operator. The `size` function recursively traverses the input list to count the number of elements in it.

To verify this module, the programmer must annotate each of the three functions with a type signature. First, for `divide` they must specify that the second argument is `NonZero` — so that LIQUIDHASKELL can verify that `error` will not be called at run-time. Second, for `size` they must specify that the function returns a strictly positive result *if* the input is non-empty. Finally, for `average` they must specify that the input list is itself non-empty, which lets LIQUIDHASKELL determine — using the annotation for `size` — that `total` is strictly positive, and hence that the call to `divide` is also safe.

Symbolic Type Inference with Qualifiers Refinement type checkers require type annotations in many places, *e.g.* for (recursive) functions, polymorphic type instantiation and so on. These can be viewed as type-based generalizations of the classic problem of having to specify pre- and post-conditions and loop- annotation invariants in Floyd-Hoare style verifiers like ESCJava or Dafny [16], [22]. As with loop invariants, refinement type inference is undecidable in general, but the type-based setting allows LIQUIDHASKELL to use a form of abstract interpretation called *predicate abstraction* [2], [23]. Here, the programmer provides a set of *qualifiers* — predicate fragments or templates — that LIQUIDHASKELL can then automatically conjoin to infer refinement types. In our running example in Figure 1, we could provide templates:

```haskell
type NonZero = {v:Int | v /= 0}

type NEList a = {v:[a] | notEmp v}

{-@ divide :: Int -> NonZero -> Int @-}
divide :: Int -> Int -> Int
divide _ 0 = error "divide-by-zero"
divide x n = x `div` n

{-@ size :: xs:_ -> {v:Nat | notEmp xs => v>0} @-}
size :: [a] -> Int
size []     = 0
size (_:xs) = 1 + size xs

{-@ average :: NEList Int -> Int @-}
average xs = divide total elems
  where
    total = sum xs
    elems  = size xs
```

Fig. 1: Haskell module with multiple dependent functions.

```haskell
qualif Qual0(v: a): v > 0
qualif Qual1(v: a, xs: b): notEmp xs => v > 0
```

and then simply annotate `average` and `size` with the *wildcard* types: `average :: _ -> _ -> _` and `size :: _ -> _` after which LIQUIDHASKELL will be able to *automatically* infer the refinement type annotations needed to verify the module [2]. Additionally, LIQUIDHASKELL can automatically *extract* qualifiers from annotated type specifications. For example if the programmer wrote a specification `x:Int -> {v:[a] | x < len v} -> a` then LIQUIDHASKELL would automatically extract a qualifier `Qual2(v: a, x: b): x < len v` and then use it for subsequent type inference.

B. Neural Type Inference with LLMs

Even with symbolic refinement type inference, there is a substantial burden on the programmer as they must be able to either write down the types for all functions *or* divine a set of suitable qualifiers from which types can be inferred. We aim to reduce this burden by using large language models (LLMs), specifically code-specific models, trained on large tracts of source code, to assist the programmer in generating the necessary type annotations needed to verify entire modules.

Constructing prompts LHC infers refinement types for entire modules by repeatedly crafting prompts that can guide the LLM to generate accurate and relevant refinement types for each function. In the case of refinement types, LHC uses LLMs that have been pre-trained on infilling *missing* (*masked*) parts of programs, and generate prompts that provide context about the Haskell code while indicating where the refinement type is missing. For Figure 1, LHC builds the following LLM prompt to generate a refinement type for `divide`

```haskell
-- Fill in the masked refinement type
{-@ type NonZero = {v:Int | v /= 0} @-}

{-@ measure notEmp :: [a] -> Bool
      notEmp []    = False
      notEmp (_:_) = True     @-}

{-@ type NEList a = {v:[a] | notEmp v} @-}

{-@ divide :: <mask> @-}
divide :: Int -> Int -> Int
divide _ 0 = error "divide-by-zero"
divide x n = x `div` n
```

Few-shot prompting with function dependencies *Few-shot prompting* is a technique used in the context of LLMs where the model is provided with *some examples* (typically between one and a few dozen) to illustrate the task it needs to perform. This approach helps the LLM understand the pattern and context of the task, improving its performance on similar tasks. In contrast, *zero-shot prompting* provides no specific examples to the LLM, relying entirely on the model's pre-trained knowledge to perform the task based on a descriptive prompt. Few-shot prompting generally yields better results than zero-shot prompting as it gives the LLM concrete examples to learn from, thereby reducing ambiguity and increasing accuracy.

In the context of refinement types, few-shot prompting can be particularly useful. For instance, when asking a Code LLM to generate refinement type annotations for Haskell code, a few-shot prompt would include several examples of functions annotated with appropriate refinement types. This helps the LLM learn the patterns and constraints associated with these types. By seeing specific examples, the LLM can more accurately predict and infill the missing refinement type annotations in new, un-annotated code.

Specifically, LHC adds all the functions and their types in the prompt, that the target function depends on. For our example in Figure 1, when we query a LLM to generate types for `average`, the functions `divide` and `size` with their type signatures will also be added to the prompt as extra examples to help the LLM generate the correct type for `average`.

III. OVERVIEW

Let's look at an overview of how LHC systematically infers the refinement type annotations needed to automatically verify a given Haskell codebase, by traversing its *call-graph*, prompting the LLM to *generate* new type predictions that can be locally *verified* by LIQUIDHASKELL, and then *back-jumping* to a dependency when the predictions fail.

A. Initialization

The input for LHC is an *un-annotated* program, *i.e.* a program where some functions are not yet annotated with a refinement type. Based on the running example the initial program is the code from Figure 1 where we removed the orange specifications for the *target* functions `divide`, `size` and `average`. Helper type aliases, such as `NEList` and `NonZero`, are standard in LIQUIDHASKELL and very commonly used by more complicated refinement types in order to simplify them. Therefore, here, they are left untouched for more context when prompting the LLM to get more accurate predictions.

LHC *State* During the entire type inference process, LHC maintains a global state S that captures the current state S_f of each function f which corresponds to a triple

(fuel, type, predicts). The fuel represents the maximum number of times that LHC will attempt to infer a type for f before giving up and asking the programmer to provide a type. The type represents the current type that the function f has been assigned and against which the implementation of f has been verified, or $\perp$ if no such type has been assigned. The predicts queue stores all the predicted types from a LLM that are yet *to be* tried. For our example, the initial global state S maps each of `divide`, `size` and `average` to a triple where fuel is 10, type is $\perp$ and predicts is empty.

Dependencies Next, we identify the potential *dependencies* between the different functions, because the order that we generate types and verify them matters for the correctness of our approach. We build the program's *call-graph* and get all function dependencies, where deps_f is the set of functions that f calls. For our example, `average` calls `divide` and `size`, each of which have no dependencies. Thus, the call-graph has a depth of 2: the dependencies of `divide` and `size` are empty, and of `average` is `[divide, size]`.

Worklist Finally, LHC maintains a worklist wkl with all the functions that are yet to be explored and verified by our approach. We initialize the wkl with all the *root functions*, *i.e.* functions that have *no dependencies*. These roots will be the starting points at which LHC will infer types. For our example we initialize wkl with `[divide, size]`.

B. Building the LLM prompt

LHC starts by popping `divide` from the working list wkl. Since we have no type predictions so far, we need to call the LLM to generate some. For that we make the following prompt (as described in § II-B):

```
measure notEmp :: [a] -> Bool
  notEmp []    = False
  notEmp (_:_) = True

type NonZero  = {v:Int | v /= 0}

type NEList a = {v:[a] | notEmp v}

{-@ divide :: <mask> @-}
divide :: Int -> Int -> Int
divide _ 0 = error "divide-by-zero"
divide x n = x `div` n
```

The prompt contains the program up to the function that we are generating refinement types for, where we add a "dummy" refinement type `<mask>` that the LLM needs to fill in.

DEPENDENCIES ***Optimization*** As described in § II-B, this is a few-shot promping technique where *all dependencies* that the target function calls, are added in the prompt. While `divide` and `size` don't have any dependencies and their prompts remain as described above, the prompt for `average` would include both of these functions when the DEPENDENCIES optimization is enabled, since `average` calls both of them.

C. Generating type predictions

Given the above prompt for `divide`, the LLM will generate the following types where the 3rd one is the correct one. [2]

```
divide :: NonZero -> Pos -> Pos -- rejected
divide :: NonZero -> Nat -> Pos -- rejected
divide :: Int -> NonZero -> Int -- correct
divide :: {v:Int | v /= 0} -> Nat -> Nat
divide :: NonZero -> Int -> {v:Int | v > 0}
```

D. Verifying types

The next step is to identify a type from the prediction queue above, that is *locally correct*, *i.e.* against which LIQUIDHASKELL will verify the *given* function (here, `divide`). We iteratively check `divide` against each of the candidate types, decreasing `divide`'s fuel each time, until we reach a locally correct type that is verified by LIQUIDHASKELL. After this step, the global state is updated so that for `divide`, we have two remaining type predictions in the predicts, the fuel has been decreased by 3 since we tested that many type predictions and the current type is updated to `Int -> NonZero -> Int`.

function	fuel	type/predicts
divide	7	**Int -> NonZero -> Int** /
		NonZero -> Nat -> Nat
		NonZero -> Int -> Nat
size	10	- /-
average	10	- /-

QUALIFIERS ***Optimization*** As described in § II-A, the programmer can provide a set of *qualifiers* and a *wildcard type* for the target function in order to enable LIQUIDHASKELL to automatically infer the appropriate refinement type [2]. When we enable the QUALIFIERS optimization, we add the wildcard type for the target function at the end of the list of predicted types, *i.e.* `divide :: _ -> _ -> _` for our example, in order for this type to be tested as a last resort if all other types fail verification. Additionally, we *extract automatically* all possible predicates from the predicted refinement types to add the corresponding qualifiers in the program. For example, for `divide` we would extract `Qual1(v: a): v /= 0` and `Qual2(v: a): v > 0` from the last two predictions from § III-C

E. Updating the working list

After we locally verify `divide` with one of the predicted types, we look up all functions f that call `divide`, *i.e.* the functions f such that deps_f contains `divide`, and we add them to the working list wkl if *all their dependencies* are resolved, *i.e.* all functions in deps_f have also been locally verified against their current type. In this case `average` calls `divide` but `average` also depends on `size`, which we have yet to explore. Therefore, *no new functions* are added to the working list wkl, which now, only contains `size`.

As in § III-B and § III-C, we perform the same steps for `size` to generate type predictions:

[2]We consider here the top 5 predictions but in reality can generate up to 50 types in total

```
size :: xs:[a] -> {v:Int | v > 0}
size :: xs:[a] -> {v:Int | v >= 0}
size :: xs:[a] -> {v:Int | v = size xs}
size :: xs:[a] -> {v:Nat | notEmp xs => v>0}
size :: xs:[a] -> {v:Nat | v = len xs}
```

The first predicted type `xs:[a] -> {v:Int | v > 0}` is rejected by LIQUIDHASKELL as it is not locally correct as `size` can return `0` on an empty list. The second type `xs:[a] -> {v:Int | v >= 0}`, however, is *locally* correct – the output of `size` is always non-negative. (Note, however, it is not the type that is needed to verify the whole module, in particular, that is needed to verify `average` which we still have not explored. We show next how this issue is resolved in our approach.) At this point, the global state is updated to

function	fuel	type/predicts
divide	7	**Int -> NonZero -> Int** / NonZero -> Nat -> Nat NonZero -> Int -> Nat
size	8	**xs:[a] -> v:Int | v >= 0** / xs:[a] -> v:Int | v = size xs xs:[a] -> v:Nat | notEmp xs => v > 0 xs:[a] -> v:Nat | v = len xs
average	10	- /-

which corresponds to the *partially annotated* program:

```
{-@ divide :: Int -> NonZero -> Int @-}
divide :: Int -> Int -> Int
divide _ 0 = error "divide-by-zero"
divide x n = x `div` n

{-@ size :: xs:[a] -> {v:Int | v >= 0} @-}
size :: [a] -> Int
size []     = 0
size (_:xs) = 1 + size xs

average xs = divide total elems
  where
    total  = sum xs
    elems  = size xs
```

At this stage, since all of `average`'s dependencies are also locally verified, we can add it to the working list wkl, so that we can generate and try types for it as well.

F. Back-jumping to a dependency when predictions fail

Now, LHC repeats the same generate-and-check procedure for `average` as it did for `divide` and `size`. However, this time, the 5 LLM-predicted types are invalid, in that *none* of them can be locally verified against the implementation of `average`. This could mean one of two things:

(1) One of `average`'s function dependencies has a type that is either too *weak* (*i.e.* does not specify what the client requires in its post-condition), or too *strong* (*i.e.* has a pre-condition that rejects the actual inputs provided by the client).
(2) All type predictions for `average` were wrong.

In this case, condition (1) holds as `size` had indeed a problematic type as we hinted earlier, which made `average` impossible to verify. However, at this stage, LHC cannot distinguish between (1) or (2) — *i.e.* we do not *know* which condition actually holds and therefore we need to make a decision based on the global state.

Since all of the predictions in the predicts queue for `average` are exhausted, and we have available fuel for it, we choose to *back-jump* to one of `average`'s function dependencies, in particular, the one that has the *highest remaining fuel*, *i.e.* the one that has been the least tested and thus has the most candidate type predictions left. The dependencies that we can potentially back-jump to are not just the immediate parent functions in the call graph, but any possible *ancestor*, which in this case includes `divide` and `size`.

In this case, we would indeed jump back to the problematic `size` that has the highest fuel of 8. In this process we clear *all current types* for functions that directly or transitively call `size` and we end up with the following state where `average`'s fuel has now fallen to 5, as we made 5 unsuccessful local verification attempts for it and additionally `average` and `size` have no assigned type.

function	fuel	type/predicts
divide	7	**Int -> NonZero -> Int** / NonZero -> Nat -> Nat NonZero -> Int -> Nat
size	8	- / xs:[a] -> v:Int | v = size xs xs:[a] -> v:Nat | notEmp xs => v > 0 xs:[a] -> v:Nat | v = len xs
average	5	- /-

We now repeat the process of trying type predictions from the predicts for `size` until we get another locally correct type. Of the three remaining predictions predicts the first of these is rejected by LIQUIDHASKELL's local verification, but the second is accepted yielding the state:

function	fuel	type/predicts
divide	7	**Int -> NonZero -> Int** / NonZero -> Nat -> Nat NonZero -> Int -> Nat
size	6	**xs:[a] -> v:Nat | notEmp xs => v > 0** / xs:[a] -> v:Nat | v = len xs
average	5	- /-

At this point, we again add `average` to the working list wkl, and proceed to generate fresh candidates, and to locally verify them. In this second time, the LLM generates the candidates:

```
average :: xs:[Int] -> {v:Int | size xs > 0}
average :: NEList Int -> Int
average :: NEList a -> Int
average :: xs:NEList Int -> Int
average :: {v:[Int] | notEmp v} -> Int
```

and LIQUIDHASKELL rejects the first type to locally verify the second candidate `NEList Int -> Int`, thereby completing the verification of the whole program.

Algorithm 1 LHC's algorithm

Input: Code Repository R
Output: Verified Code Repository R'

```
1:  procedure VerifyCodeRepo(R)
2:      S ← [f ↦ {fuel = N, type = ⊥, predicts = ∅}]
3:      deps ← BuildCallGraph(R)
4:      wkl ← {f ∈ R | deps_f = ∅}
5:      while wkl ≠ ∅ do
6:          f ← PopTop(wkl)
7:          ps ← GetPredictions(S, f)
8:          t ← TryPredictions(S, f, ps)
9:          if t = ⊥ then
10:             wkl ← wkl ∪ BackJump(S, f)
11:         else
12:             S_f.type ← t
13:             wkl ← wkl ∪ SolvedCallers(S, f, deps)
14:     return R(S)
```

Algorithm 2 GetPredictions's algorithm

Input: State S, Function f
Output: LLM type predictions ps

```
1:  procedure GetPredictions(S, f)
2:      if S_f.predicts = ∅ then
3:          R ← Program(S)
4:          prompt ← MakePrompt(R, f)
5:          S_f.predicts ← LLMGuess(prompt)
6:      ps ← S_f.predicts
7:      return ps
```

G. Asking the user for a type

Suppose that instead, the LLM generated a slate of incorrect types for `average` such that while trying out the generated candidates, the fuel for `average` falls to 0. In this case, we are potentially in condition (2), where all the LLM predicted types are wrong (*i.e.* fail to locally verify). In this scenario, LHC falls back to ask the user to provide a type for `average`.

If the verification fails even with the user-provided type, then we back-jump again to one of the dependencies and repeat the whole process until the function is verified *with* the user-provided type. That is, we presume that the user-provided type is correct, and we get the LLM to generate types for the *other* functions, so that the whole program verifies. Of course, our goal is to *minimize* the number of times we have to resort to asking the user for a type signature. In our example, assuming the user provides the correct type `NEList Int -> Int` the verification of the whole program succeeds and we return the fully annotated program (shown in Figure 1) to the user.

IV. Algorithm

We describe here the full algorithm of the LHC agent: an interactive approach to verifying a *code repository* R, comprising a set of functions, by automatically annotating all the functions in R with refinement types Algorithm 1 presents LHC's high-level iterative approach.

Global State We define as S the *global state*, that maps each function f to its current state S_f which is a triple of the form (fuel, type, predicts). The first element, fuel, is an integer representing the upper bound on the number of remaining verification attempts for f. The second element, type, is the current type that has been assigned to and locally verified for f or $\bot$ denoting that no type has been assigned. The third element, predicts, is a *priority queue* of the *predicted types* for f against which f has not yet been locally verified. The *global state* S is initialized such that for each f in the repository, S_f for each function f has a maximum the S_f.fuel is some maximum N, the S_f.type is $\bot$, and S_f.predicts is empty.

Call Graph BuildCallGraph generates the repository's *call-graph* returning returns all function dependencies deps_f which is the set of functions that f calls, which we also call the *immediate dependencies* of f. For example, the program in Figure 1 has the following dependencies $\mathsf{deps}_{\texttt{divide}} = \emptyset$, $\mathsf{deps}_{\texttt{size}} = \emptyset$, $\mathsf{deps}_{\texttt{average}} = [\texttt{divide}, \texttt{size}]$.

Worklist Next, we initialize a *worklist* wkl of functions that our procedure is going to operate on. The worklist wkl is initialized with all functions $f \in R$, such that the function f has no dependencies. These are the *root functions* of the repository from which LHC starts generating and checking types.

Main Loop The main body of the algorithm is on lines 5 to 13, which iterates till all functions are assigned types and wkl is empty. In each iteration we pop the top function f from the wkl stack. Then, we get new or existing *type predictions* ps (§ IV-A) for the function f. We try to *locally verify* f with a type from the predictions ps (§ IV-B) until we successfully find a type against which f locally verifies in the module with the current state S, *i.e.* with the currently assigned types for the transitive dependencies of f. If none of the predictions ps verified the function f, thus $\mathsf{t} = \bot$, we *back-jump* to f's least tested dependency (§ IV-C) to continue the iterations from there. If instead, a type t that locally verified the function f is found, then we update the state S_f.type with the new type t, and add in wkl the functions in SolvedCallers(S, f, deps). These are all the functions $g \in R$ such that (a) g calls f (*i.e.* $f \in \mathsf{deps}_g$), and (b) all the dependencies of g have an assigned type, (*i.e.* $\forall h \in \mathsf{deps}_g.\ \mathsf{S}_h.\mathsf{type} \neq \bot$). We then continue to the next iteration, ensuring all functions in wkl have been locally verified.

A. Generating type predictions

Algorithm 2 describes the procedure of generating *new type predictions* for a function f given a global state S. Initially, we check if we have any remaining type predictions in the function's queue S_f.predicts and if there are, no new types are generated and the remaining queue is just returned. Otherwise, we retrieve the relevant functions from the codebase, given the current state S, replacing any types that have already been locally verified (*i.e.* already assigned to the type field in S) and make a prompt for function f as discussed in § III-B. This prompt is sent to the LLM to generate new type predictions, which are then added to f's prediction queue S_f.predicts.

B. Trying type predictions

Algorithm 3 presents the process of trying new type predictions ps for a function f given a global state S, to find the first

Algorithm 3 TRYPREDICTIONS's algorithm

Input: State S, Function f, Type Predictions ps
Output: Successful type t or Failure $\bot$

```
1: procedure TRYPREDICTIONS(S, f, ps)
2:     while ps ≠ ∅ and S_f.fuel > 0 do
3:         (t, ps) ← GETNEXT(ps)
4:         S_f.fuel ← S_f.fuel − 1
5:         if LHVerify(S, f, t) then
6:             return t
7:     if S_f.fuel = 0 then
8:         return UserHint(f)
9:     return ⊥
```

Algorithm 4 BACKJUMP's algorithm

Input: State S, Function f
Output: Transitive dependency f' with max remaining fuel

```
1: procedure BACKJUMP(S, f)
2:     allDeps ← ∅, wkl ← {f}
3:     while wkl ≠ ∅ do
4:         f' ← POPTOP(wkl)
5:         allDeps ← allDeps ∪ {f'}
6:         wkl ← wkl ∪ {g | g ∈ deps_f', g ∉ allDeps}
7:     f' ← ⊥, maxfuel ← 0
8:     for all g ∈ allDeps − {f} do
9:         if maxfuel < S_g.fuel then
10:            f' ← g, maxfuel ← S_g.fuel
11:    S ← RESETDEPENDENCIES(S, f')
12:    return f'
```

prediction against which f locally verifies. In each iteration, we first check the *remaining* fuel for the current function f. If it has reached 0 then we ask the user to provide a type, as we discussed in (§ III-G). If there is more fuel left, then we decrement f's fuel S_f.fuel by one. Then, we get the next prediction t from the queue ps, and assign the type to f and query LIQUIDHASKELL to try to locally verify the program using the other types already assigned in S. If the verification process is successful, we return the locally verified type t, and otherwise we continue to the next iteration. If we have tried all the predictions in the queue ps and none of them locally verified the function f, then we return $\bot$ signalling that none of the candidate predictions were successful.

C. Back-jumping to the least tested dependency

Algorithm 4 outlines the BACKJUMP procedure, which identifies and returns the transitive dependency of a function f that has the maximum remaining fuel. The algorithm begins by finding all transitive dependencies of f. This is achieved using a worklist wkl, initialized with f. The algorithm iteratively processes each function in the worklist by popping the top function f', adding it to the set of all dependencies allDeps, and then including all its immediate dependencies $\mathsf{deps}_{f'}$ that are not already in allDeps back into the worklist wkl. This loop continues until the worklist is empty, ensuring that all transitive dependencies of f are collected.

Once all transitive dependencies are identified, the next step is to determine the dependency with the maximum remaining fuel. The algorithm initializes f' to $\bot$ and maxfuel to 0. It then iterates over each function g in allDeps excluding f. If the remaining fuel S_g.fuel for a function g is greater than maxfuel, it updates f' to g and maxfuel to S_g.fuel. This ensures that the function with the maximum remaining fuel among the transitive dependencies of f is selected.

After identifying the function f' with the maximum remaining fuel, the global state S is reset for this function and its dependencies using the RESETDEPENDENCIES procedure, thereby restarting the verification process for f' from scratch. For each function f that calls f', RESETDEPENDENCIES will set S_f.type to $\bot$ and recursively will be applied to each caller f to reset all functions that indirectly depend on f'. This allows the type verification process to strategically back-jump to f' and restart with updated predictions and fuel. The selected function f' is then finally returned.

V. EVALUATION

Next, we describe our implementation of LHC (LHCOPILOT), and an evaluation that addresses three research questions:

- ***RQ1:*** How *accurate* are LLMs at generating *single* refinement types? (§ V-A)
- ***RQ2:*** How *precisely* can LHC verify *whole* codebases? (§ V-B)
- ***RQ3:*** How *efficiently* can LHC verify a given codebase? (§ V-C)

Large Language Models For our evaluation, we focus on *Code LLMs*, *i.e.* LLMs that have been pre-trained specifically for generating code. Such models don't usually require special system or instruction prompts to effectively generate the target code, unlike CHATGPT or GPT-4O. Our emphasis is also on *smaller and public* LLMs for two important reasons. First, they can be more *robust* in generating types for the hundreds of times it is necessary to verify the given codebase (and likely just as effective [24]). Second, and more importantly, to guard against the possibility of data-leakage (memorization), we wish to ensure that our benchmarks are *not* in the training set for the models. Therefore, we use the following LLMs; STARCODER-3B [18], CODELLAMA-7B [25] and STARCODER2-15B [19] with 3, 7 and 15 billion trainable parameters respectively. All the models are open-source and have been pre-trained on public datasets of code [18], [19]. In each case, the training data does not include the benchmarks considered here. We run all experiments with STARCODER-3B and CODELLAMA-7B on an NVIDIA GeForce RTX 3080 Ti with 12 GB VRAM and an NVIDIA A100 PCIe with 80 GB VRAM for STARCODER2-15B.

Fine-tuning datasets We fine-tune our models using *The Stack v2* [19], a public dataset of open-souce code with permissive licenses, that includes a vast number of programming languages. While the dataset consists of less than 0.2% of Haskell programs, those programs amount to over 1 million. Of those programs, only 3350 used LIQUIDHASKELL and refinement types. From these programs, we extract a dataset of 8903 *unique refinement types*, and we use them to fine-tune the

LLMs. Specifically, we use QLoRA [26], an efficient fine-tuning approach that reduces memory usage enough to fine-tune much larger LLMs on smaller GPUs while preserving full 16-bit fine-tuning task performance. QLoRA backpropagates gradients through a frozen, 4-bit quantized pretrained language model into Low Rank Adapters (LoRA [27]). We fine-tune StarCoder-3B and CodeLlama-7B for 20 epochs, while the larger StarCoder2-15B for 10 epochs due to limited time and resources and its excessive cost.

Benchmarks We explore different benchmarks in order to answer our research questions. First, for single refinement type prediction (RQ1), we evaluate the different LLMs on a new benchmark based on the publicly available online LiquidHaskell tutorial [28]. We extract *68 refinement types* from the LHTutorial[3], corresponding to exercises with hidden solutions, into separate programs along with their relevant context, such as our running example in Figure 1. For each refinement type in this benchmark, we build a LLM prompt that includes the implemented Haskell function with its type signature and any surrounding context from the corresponding exercise. For example, any relevant functions, comments or possible input-output test cases that can help verify the correctness of the target function. Additionally, functions called by the target function that are already annotated with refinement types were also provided when available.

For the rest of our evaluation (RQ2 and RQ3), we use two public Haskell libraries, HSalsa20 and ByteString. HSalsa20 is a Haskell implementation of the Salsa20 cipher with refinement types used to track the sizes of various ciphers and keys. ByteString is a Haskell library for representing and efficiently operating on byte-strings via pointer operations that is widely used across the Haskell ecosystem; we use refinement types to statically track pointer arithmetic and ensure low-level memory safety. HSalsa20 was annotated with refinement types by its developer, including *96* such annotated functions, while we annotated 45 ByteString functions with refinement types, in order to establish a ground truth type for these benchmarks. (While there are several other LiquidHaskell repositories available online, they are in the training data for the LLMs we consider, and hence, are excluded from our evaluation.)

Emulating User Interaction We selected benchmarks that are already fully annotated with refinement types for each function, so that we could emulate the user interaction — UserHint in § IV-B — in our experiments using these ground truth types. That is, when all the predicted types fail for a given target function, we use the ground truth type used to annotate the function in the original benchmark *as* the type that was provided by the user.

Baselines We acknowledge the importance of evaluating against a baseline. However, as we discuss in section VI, symbolic program synthesis methods have historically faced significant challenges in this domain. Recent advances in machine learning and LLMs have enabled substantial improvements, making this problem more tractable. A purely random generation approach would be of limited utility given the immense size of the search space; the refinement type space is much larger and more complex than the space of standard Haskell types. Consequently, there aren't any meaningful baselines in the existing literature for generating refinement types and verifying entire codebases effectively until now.

[3]Our benchmarks are available along with code for LHC at https://github.com/gsakkas/ref-type-pred

A. RQ1: Single type prediction accuracy

Table I presents the cumulative results on the LHTutorial. We have manually categorized the 68 programs with single target functions that need to be annotated with a refinement type as *Easy*, *Medium* and *Hard*, based on the complexity of the ground truth refinement type. An example of an "Easy" refinement type is `average :: NEList Int -> Int` from our running example in Figure 1, while a "Hard" type would be for the following function:

```
{-@ zipOrNull :: as:[a]
    -> bs:{[b] | (notEmp as && notEmp bs)
                  => (len bs = len as)}
    -> {v:[(a, b)] | if (notEmp as && notEmp bs)
                     then (len v = len as)
                     else (len v = 0) }
@-}
zipOrNull :: [a] -> [b] -> [(a, b)]
zipOrNull [] _ = []
zipOrNull _ [] = []
zipOrNull xs ys = zipWith (,) xs ys
```

This refinement type verifies the correctness of `zipOrNull`, which zips two lists only if neither of them is empty. Other "Hard" examples in our benchmark can include more complicated predicates or functions with more dependencies. As with any code generation tasks, longer programs (or refinement types) with more dependencies tend to be more complicated and difficult to generate with LLMs.

For each target function we generate 50 refinement types using the pre-trained LLMs and their fine-tuned versions. We also show the number of functions that the LLMs successfully verified at Figure 2. We observe that all models showcase great performance for Easy types, with slight improvements when fine-tuned. However, there is great improvement for all LLMs in the Medium and Hard categories. Specifically, StarCoder-3B shows a 12-program improvement in the Medium category, while CodeLlama-7B and StarCoder2-15B show a 7- and 9-program improvement respectively. We see a similar improvement for the Hard programs with an increase of 10, 12 and 11 programs respectively.

We also present the $pass@k$ results in Table I. [29] introduced the $pass@k$ metric, and the Codex paper [30] popularized it recently, specifically for evaluating code generation LLMs. To calculate $pass@k$, k code samples are generated per problem and the problem is considered solved if any sample is correct, where $pass@k$ is the total fraction of problems solved. However, as it has been observed in [30], computing $pass@k$ this way can have high variance. Instead, to evaluate $pass@k$, $n \geq k$ samples per task are generated (in this paper, we use

LLM	*Total* (68)	*Easy* (17)	*Medium* (30)	*Hard* (21)	*pass@1*	*pass@5*	*pass@10*	*pass@20*	*pass@50*
StarCoder-3B	37	14	16	7	12.91%	30.19%	38.30%	45.82%	54.41%
+ Fine-tuned	62	17	28	17	65.88%	82.25%	85.22%	87.63%	91.18%
CodeLlama-7B	41	15	20	6	11.68%	32.01%	42.01%	50.85%	60.29%
+ Fine-tuned	60	15	27	18	47.97%	71.76%	78.67%	83.62%	88.24%
StarCoder-15B	42	16	19	7	18.79%	41.67%	50.21%	56.78%	61.76%
+ Fine-tuned	62	16	28	18	57.65%	78.79%	84.26%	88.19%	91.18%

TABLE I: LHTutorial results: 68 total single type benchmark, where we divided the user-intended type into 3 difficulty categories. We also present the $pass@k$ metrics for the full benchmark.

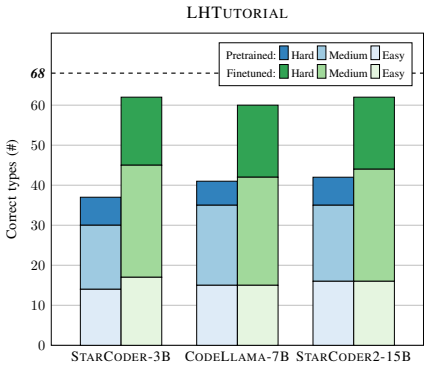

Fig. 2: LLM accuracy in generating single refinement types.

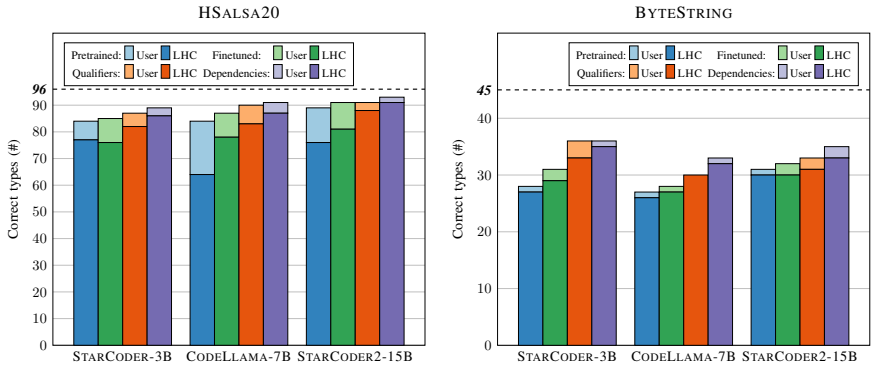

Fig. 3: LHC accuracy in generating and verifying refinement types for our Haskell benchmarks.

$n = 50$ and $k \leq 50$), and the number of correct samples $c \leq n$ is counted in order to calculate the unbiased estimator:

$$pass@k = \mathop{\mathbb{E}}_{problems}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

$pass@50$ here represents the total *accuracy* of each LLM for the LHTutorial, *i.e.* the percentage of the target functions that the LLMs generated at least one correct refinement type.

By fine-tuning, we observe great improvement for all $pass@k$ metrics, reaching a $pass@10$ of 85% and a $pass@50$ of 91% in some cases. We also see that there isn't a big improvement for $k > 10$ samples, especially for the fine-tuned models, indicating that sampling even 10 refinement types per function can yield very accurate results with a fine-tuned LLM.

> Fine-tuned LLMs — even with a small number of trainable parameters — learn to encode LiquidHaskell programs and can generate correct refinement types for *91% of the target functions*, an improvement of up to *35%* from the out-of-the-box models.

B. RQ2: Whole codebase precision

Next, we evaluate LHC on the two Haskell codebases HSalsa20 and ByteString. We evaluate here *four* different approaches with each LLM, presented in Table II for HSalsa20 and ByteString. First, we use the *original* pretrained LLMs as a backend for LHC in order to generate the various refinement types. Second, we *fine-tune* the models as described before. Next, we enable the Qualifiers verification optimization, an optimization which automatically extracts *all qualifiers* from the LLM predicted types and adds corresponding pragmas in the context of the program, as described in § III-D. This optimization also adds the relevant wildcard type as a last candidate type, that is tested when all LLM-predicted types are exhausted, which tells LiquidHaskell to perform symbolic type inference using the qualifiers. Finally, we enable the Dependencies prompt optimization for additional context. This optimization uses few-shot prompting to add the verified dependency functions to the LLM prompts (§ III-B).

Automatically Verified This metric (first column of Table II) indicates the number of functions that were automatically annotated and verified by LHC without human intervention. For example, StarCoder-3B in its fine-tuned version correctly inferred types for 76 functions in HSalsa20 and verified up to 86 functions when the Qualifiers and Dependencies optimizations are enabled. CodeLlama-7B shows similar improvement. However, the larger StarCoder2-15B reaches up to 91 functions that are correctly annotated by LHC.

Correct Unverified Type Predictions This metric shows the number of times that the LLMs generated a *correct* refinement type for a function, but LHC *ran out of fuel* (as discussed in § IV-B) and the function was not automatically (and locally) verified by LHC — thus *not included* in the previous metric. In a real-world setting, however, before asking the user to manually write a refinement for these unsuccessful functions, as a preliminary step we could provide the list of LLM-predicted types and have the *user select or approve* one in order to mitigate the manual effort. While a higher number for this metric indicates less dependency on manual input, it still represents wasted cycles for LHC, where it couldn't arrive to

Benchmark	LLM	*Automatically Verified*	*Correct Unverified*	LHC *Iterations*	*Total* LHC *Time (mins)*	*Type Pred. Time (mins)*	*Verification Time (mins)*
HSALSA20	STARCODER-3B	77	+7	562	257	81	176
	+ FINE-TUNED	76	+9	583	227	76	151
	+ QUALIFIERS	82	+5	400	295	61	234
	+ DEPENDENCIES	**86**	**+3**	**337**	**238**	71	167
	CODELLAMA-7B	64	+20	760	246	113	133
	+ FINE-TUNED	78	+9	485	252	88	164
	+ QUALIFIERS	83	+7	365	239	66	173
	+ DEPENDENCIES	**87**	**+4**	**334**	**201**	64	137
	STARCODER2-15B	76	+13	496	351	154	197
	+ FINE-TUNED	81	+10	472	336	151	185
	+ QUALIFIERS	88	+3	314	358	123	235
	+ DEPENDENCIES	**91**	**+2**	**258**	**254**	101	153
BYTESTRING	STARCODER-3B	27	**+1**	95	73	24	49
	+ FINE-TUNED	29	+2	104	52	23	29
	+ QUALIFIERS	33	+3	99	49	22	27
	+ DEPENDENCIES	**35**	**+1**	**93**	**42**	25	17
	CODELLAMA-7B	26	+1	114	87	34	53
	+ FINE-TUNED	27	+1	**110**	84	34	50
	+ QUALIFIERS	30	**+0**	137	90	35	55
	+ DEPENDENCIES	**32**	+1	114	**80**	32	48
	STARCODER2-15B	30	**+1**	95	115	56	59
	+ FINE-TUNED	30	+2	**88**	**110**	51	59
	+ QUALIFIERS	31	+2	101	121	55	66
	+ DEPENDENCIES	**33**	+2	100	111	53	58

TABLE II: HSALSA20 (*96 functions*) and BYTESTRING verification results (*45 functions*)

the correct combination of refinement types in order to locally verify the faulty ones. We observe that all LLMs have relatively high numbers when QUALIFIERS and DEPENDENCIES were not used. When we use both optimizations, the numbers go as low as only 2 for STARCODER2-15B, since LHC was able to automatically verify the vast majority of the functions when using this LLM, as we showed earlier.

Cumulative results Figure 2 also shows the cumulative results of the previous two metrics, *i.e.* the total number of functions LHC was able to *generate a correct type* for and were either automatically or manually verified. We observe that for HSALSA20 even without the last two optimizations, LHC generates a significant fraction — nearly 80% — of correct types, but the optimizations nevertheless improve the accuracy of types automatically generated by LHC to more than 90%. However, for the more complicated module BYTESTRING though, we observe that without the QUALIFIERS and DEPENDENCIES optimizations, we can only generate 60% of the types, and the addition of symbolic qualifier inference and context provides a significant improvement, allowing LHC to generate correct types for up to 77% of the functions.

> LHC can automatically generate formally verified types for up to *94%* of a codebase's functions.

C. RQ3: Efficiency

The last four columns of Table II summarize how *efficiently* LHC can verify codebases.

LHC *Iterations* This metric counts the number of iterations that LHC needs to verify the full module. Fewer iterations suggest a more efficient verification process. We observe that, for HSALSA20 that has more functions to verify and deeper dependencies, there is a significant improvement in the time we spent verifying them when we use QUALIFIERS and DEPENDENCIES. On the other hand, for BYTESTRING, we observe a slight overhead, for unclear reasons, but perhaps due increasing the size of the prompts and getting diminishing results from the extra context in terms of efficiency.

Total* LHC *Time This metric provides an overview of the total time in minutes that LHC spent in fully verifying the modules. At a high-level, it is remarkable that LHC is able to annotate and verify entire codebases in 1-4 hours, as typically this work takes days or weeks for a human to do. (Of course, this comes with the caveat that with these benchmarks we know that suitable types exist, and we emulated human assistance when the LLM got stuck). The results also indicate that the verification time varies significantly depending on the LLM used and the specific optimizations applied. For instance, we see that LHC with STARCODER-3B required 257 minutes to verify HSALSA20, which was reduced to 227 minutes with fine-tuning, then adjusted to 295 minutes with QUALIFIERS, but drops to 238 minutes with DEPENDENCIES. Similarly, CODELLAMA-7B and STARCODER2-15B models show notable reductions in verification time when dependencies are included. In the BYTESTRING results, the trend is consistent for STARCODER-3B and CODELLAMA-7B. However, STARCODER2-15B shows no significant improvement.

Type Prediction & Verification Time These metrics evaluate the time spent by LHC on generating type predictions using LLMs compared to the time required for codebase verification with LIQUIDHASKELL. Our analysis reveals that smaller LLMs are notably more efficient, generating refinement types more quickly and consistently than their larger counterparts. Specifically, type generation by LLMs accounted for approximately 37% of the total processing time when using STARCODER-3B, whereas it increased to 44% when employing STARCODER2-15B. Verification times with LIQ-

UIDHASKELL remain very consistent across different models. Finally, introducing the QUALIFIERS and DEPENDENCIES optimizations led to slight reductions in verification time, primarily due to the enhanced accuracy of our approach.

> LHC can, with modest emulated human assistance, annotate and verify entire codebases in a few hours. The fine-tuning, QUALIFIERS and DEPENDENCIES optimizations greatly enhance verification efficiency by reducing verification time by an average *18%* (and up to *42%*) across real-world codebases.

D. Threats to Validity

We note three threats to the validity of our results. First, we have only considered three codebases: the LHTUTORIAL, HSALSA20, and BYTESTRING. It is entirely possible that larger or more complex codebases may require annotations that cannot be generated so effectively by LLMs. Second, our approach currently doesn't support *mutually recursive functions*. A potential solution to this is to *break the cycles* created by these mutually recursive functions by either requiring the programmer to specify a type for one of the functions of the cycle, or speculatively breaking the cycle and letting our backtracking mechanism synthesize the types. However, we have not tried either of these approached as mutually recursive functions don't occur in our benchmarks. Third, we have emulated human assistance in our evaluation, meaning on our benchmarks we know *a priori* that suitable refinement type annotations exist. In a more realistic scenario using LHC on a new codebase, such types may not exist because bugs in the code may require it to be modified, or because of limitations in the verifier (LIQUIDHASKELL) itself. We defer the evaluation of LHC on new codebases with actual users to future work (but note that this is challenging as realistically, annotation requires days or weeks of human effort).

VI. RELATED WORK

Finally, we discuss some related lines of work on using machine learning and LLMs to automate program verification.

Generating Proof Annotations LHC is most closely related to several other neurosymbolic approaches to generating the annotations needed for formal verification. CODE2INV [31], [32], and Pei et al. [33] present techniques to synthesize loop invariants [34] using neural networks and fine-tuned LLMs, respectively. Kamath et al. [35] and [36] integrates LLMs natively with automated reasoners — ESCJava [16] and ESBMC [37] — to additionally check whether the generated invariants are actually inductive and, optionally, further to repair the invariants by querying the LLMs and reducing the proposed invariants into an inductive set using the HOUDINI algorithm [38]. Several studies have also explored the generation of annotations and formal postconditions using advanced techniques. Similarly, NL2POSTCOND [39] investigates the transformation of natural language intent into formal method postconditions using LLMs, proposing metrics for assessing the quality of these transformations. Finally, LAUREL [40] automatically generates helper assertions for proofs written in DAFNY by leveraging LLMs as well. All the above focus on a single goal: generating loop invariants, or contracts for single functions in isolation. In contrast, LHC is an *interprocedural* method that aims to generate interdependent refinement type annotations (which generalize invariants, pre- and post-conditions) across the whole codebase.

Generating Code from Specifications A different line of work looks at using LLMs to synthesizing *code* from formal specifications in proof-oriented languages. Misu et al. [41] and CLOVER [42] use LLMs to synthesize verified DAFNY code corresponding to natural language and formal specifications. Similarly, Chakraborty et al. [24] investigate using LLMs to synthesize F* programs (and proofs) from dependent type specifications. In contrast to the above, LHC's goal is not to generate code, but only the refinement contract annotations needed for modular verification of existing code(bases).

LLMs for Proof Generation Several groups have looked into using machine learning techniques to automate proofs written in tactic-based interactive theorem provers. Sanchez et al. [43] uses machine learning techniques to generate COQ proofs. BALDUR [44] explores the use of LLMs to generate entire Isabelle/HOL proofs for program verification, a departure from traditional proof assistants that generate one proof statement at a time. Complementing this, Yang et al. [45] introduces LEANDOJO, a tool that combines LLMs with retrieval-augmented mechanisms to enhance theorem proving in the LEAN environment, demonstrating improved proof generation capabilities. Wu et al. [46] evaluates the performance of LLMs in autoformalization in Isabelle/HOL, introducing a neural theorem prover trained on autoformalized statements. Unlike the above, LHC is designed to work in the setting of SMT-based "auto-active" verification, where the only programmer input is the code and the type specification; the rest is handled by the SMT solver.

In-Context Prompting In-context prompting techniques have been explored to enhance the few-shot learning capabilities of LLMs. Liu et al. [47] investigates the relationship between in-context example selection and GPT-3's few-shot performance, introducing a retrieval model for better example selection. Su et al. [48] proposes a framework for selective annotation to improve accuracy in in-context learning scenarios. Lu et al. [49] demonstrates the importance of prompt order, using model-generated sequences to find optimal prompts, while Sorensen et al. [50] introduces an information-theoretic approach to prompt engineering, maximizing mutual information to select the best prompts without relying on model weights or ground truth labels. LAUREL [40] introduced a lemma similarity metric to import potentially related lemmas to the current proof. These lines of work inspired our DEPENDENCIES optimization, where we augment our prompts with additional context. Unlike the above, LHC does not rely on similarity heuristics but instead, it includes the function dependencies of the target function we are trying to locally verify.

References

[1] H. Xi and F. Pfenning, "Eliminating array bound checking through dependent types," in *PLDI*, 1998.

[2] P. M. Rondon, M. Kawaguci, and R. Jhala, "Liquid types," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 159–169. [Online]. Available: https://doi.org/10.1145/1375581.1375602

[3] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement types for haskell," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 269–282. [Online]. Available: https://doi.org/10.1145/2628136.2628161

[4] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J. Zinzindohoue, and S. Béguelin, "Dependent types and multi-monadic effects in f," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, 2016, pp. 256–270, 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016 ; Conference date: 20-01-2016 Through 22-01-2016. [Online]. Available: https://popl16.sigplan.org/home

[5] P. Rondon, M. Kawaguchi, and R. Jhala, "Low-level liquid types," in *POPL*, 2010.

[6] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg, "Refinedc: automating the foundational verification of C code with refined ownership types," in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 158–174. [Online]. Available: https://doi.org/10.1145/3453483.3454036

[7] C. Pulte, D. C. Makwana, T. Sewell, K. Memarian, P. Sewell, and N. Krishnaswami, "CN: verifying systems C code with separation-logic refinement types," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, pp. 1–32, 2023. [Online]. Available: https://doi.org/10.1145/3571194

[8] M. Kazerounian, N. Vazou, A. Bourgerie, J. S. Foster, and E. Torlak, "Refinement types for ruby," *CoRR*, vol. abs/1711.09281, 2017. [Online]. Available: http://arxiv.org/abs/1711.09281

[9] N. Lehmann, A. T. Geller, N. Vazou, and R. Jhala, "Flux: Liquid types for rust," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, jun 2023. [Online]. Available: https://doi.org/10.1145/3591283

[10] L. Gäher, M. Sammler, R. Jung, R. Krebbers, and D. Dreyer, "Refinedrust: A type system for high-assurance verification of rust programs," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, pp. 1115–1139, 2024. [Online]. Available: https://doi.org/10.1145/3656422

[11] P. Vekris, B. Cosman, and R. Jhala, "Refinement types for typescript," in *PLDI*, 2016.

[12] J. Hamza, N. Voirol, and V. Kuncak, "System FR: formalized foundations for the stainless verifier," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 166:1–166:30, 2019. [Online]. Available: https://doi.org/10.1145/3360592

[13] B. Tan, B. Mariano, S. K. Lahiri, I. Dillig, and Y. Feng, "Soltype: refinement types for arithmetic overflow in solidity," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, jan 2022. [Online]. Available: https://doi.org/10.1145/3498665

[14] A. M. Kent, D. Kempe, and S. Tobin-Hochstadt, "Occurrence typing modulo theories," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 296–309. [Online]. Available: https://doi.org/10.1145/2908080.2908091

[15] C. Gamboa, P. Canelas, C. S. Timperley, and A. Fonseca, "Usability-oriented design of liquid types for java," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1520–1532. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00132

[16] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *PLDI*, 2002.

[17] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370.

[18] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you," 2023. [Online]. Available: https://arxiv.org/abs/2305.06161

[19] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder 2 and the stack v2: The next generation," 2024. [Online]. Available: https://arxiv.org/abs/2402.19173

[20] M. Kawaguchi, P. Rondon, and R. Jhala, "Type-based data structure verification," in *PLDI*, 2009.

[21] C. G. Nelson, "Techniques for program verification," Ph.D. dissertation, Stanford University, 1980.

[22] K. R. M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010. [Online]. Available: https://doi.org/10.1007/978-3-642-17511-4_20

[23] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, ser. Lecture Notes in Computer Science, O. Grumberg, Ed., vol. 1254. Springer, 1997, pp. 72–83. [Online]. Available: https://doi.org/10.1007/3-540-63166-6_10

[24] S. Chakraborty, G. Ebner, S. Bhat, S. Fakhoury, S. Fatima, S. Lahiri, and N. Swamy, "Towards neural synthesis for smt-assisted proof-oriented programming," 2024. [Online]. Available: https://arxiv.org/abs/2405.01787

[25] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2024. [Online]. Available: https://arxiv.org/abs/2308.12950

[26] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," 2023. [Online]. Available: https://arxiv.org/abs/2305.14314

[27] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021. [Online]. Available: https://arxiv.org/abs/2106.09685

[28] N. V. Ranjit Jhala, Eric Seidel, "Programming with refinement types," 2014. [Online]. Available: https://ucsd-progsys.github.io/liquidhaskell-tutorial/

[29] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. S. Liang, "Spoc: Search-based pseudocode to code," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/7298332f04ac004a0ca44cc69ecf6f6b-Paper.pdf

[30] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder,

B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[31] X. Si, A. Naik, H. Dai, M. Naik, and L. Song, "Code2inv: A deep learning framework for program verification," in *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, ser. Lecture Notes in Computer Science, S. K. Lahiri and C. Wang, Eds. Springer, 2020, vol. 12225, pp. 151–164.

[32] N. Kobayashi, T. Sekiyama, I. Sato, and H. Unno, "Toward neural-network-guided program synthesis and verification," 2021. [Online]. Available: https://arxiv.org/abs/2103.09414

[33] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, "Can large language models reason about program invariants?" in *Proceedings of the 40th International Conference on Machine Learning*, ser. ICML'23. JMLR.org, 2023.

[34] R. W. Floyd, "Assigning meanings to programs," *Mathematical aspects of computer science*, vol. 19, no. 19-32, p. 1, 1967.

[35] A. Kamath, A. Senthilnathan, S. Chakraborty, P. Deligiannis, S. K. Lahiri, A. Lal, A. Rastogi, S. Roy, and R. Sharma, "Finding inductive loop invariants using large language models," 2023. [Online]. Available: https://arxiv.org/abs/2311.07948

[36] H. Wu, C. Barrett, and N. Narodytska, "Lemur: Integrating large language models in automated program verification," 2024. [Online]. Available: https://openreview.net/forum?id=Q3YaCghZNt

[37] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "Esbmc 5.0: an industrial-strength c model checker," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 888–891. [Online]. Available: https://doi.org/10.1145/3238147.3240481

[38] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for esc/java," in *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, ser. FME '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 500–517.

[39] M. Endres, S. Fakhoury, S. Chakraborty, and S. K. Lahiri, "Can large language models transform natural language intent into formal method postconditions?" *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, 2024. [Online]. Available: https://doi.org/10.1145/3660791

[40] E. Mugnier, E. A. Gonzalez, R. Jhala, N. Polikarpova, and Y. Zhou, "Laurel: Generating dafny assertions using large language models," 2024. [Online]. Available: https://arxiv.org/abs/2405.16792

[41] M. R. H. Misu, C. V. Lopes, I. Ma, and J. Noble, "Towards ai-assisted synthesis of verified dafny methods," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, p. 812–835, Jul. 2024. [Online]. Available: http://dx.doi.org/10.1145/3643763

[42] C. Sun, Y. Sheng, O. Padon, and C. Barrett, "Clover: Closed-loop verifiable code generation," 2024. [Online]. Available: https://arxiv.org/abs/2310.17807

[43] A. Sanchez-Stern, Y. Alhessi, L. Saul, and S. Lerner, "Generating correctness proofs with neural networks," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–10. [Online]. Available: https://doi.org/10.1145/3394450.3397466

[44] E. First, M. N. Rabe, T. Ringer, and Y. Brun, "Baldur: Whole-proof generation and repair with large language models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1229–1241. [Online]. Available: https://doi.org/10.1145/3611643.3616243

[45] K. Yang, A. M. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. Prenger, and A. Anandkumar, "Leandojo: Theorem proving with retrieval-augmented language models," 2023. [Online]. Available: https://arxiv.org/abs/2306.15626

[46] Y. Wu, A. Q. Jiang, W. Li, M. N. Rabe, C. Staats, M. Jamnik, and C. Szegedy, "Autoformalization with large language models," 2022. [Online]. Available: https://arxiv.org/abs/2205.12615

[47] J. Liu, D. Shen, Y. Zhang, B. Dolan, L. Carin, and W. Chen, "What makes good in-context examples for gpt-3?" 2021. [Online]. Available: https://arxiv.org/abs/2101.06804

[48] H. Su, J. Kasai, C. H. Wu, W. Shi, T. Wang, J. Xin, R. Zhang, M. Ostendorf, L. Zettlemoyer, N. A. Smith, and T. Yu, "Selective annotation makes language models better few-shot learners," 2022. [Online]. Available: https://arxiv.org/abs/2209.01975

[49] Y. Lu, M. Bartolo, A. Moore, S. Riedel, and P. Stenetorp, "Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity," 2022. [Online]. Available: https://arxiv.org/abs/2104.08786

[50] T. Sorensen, J. Robinson, C. Rytting, A. Shaw, K. Rogers, A. Delorey, M. Khalil, N. Fulda, and D. Wingate, "An information-theoretic approach to prompt engineering without ground truth labels," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2022. [Online]. Available: http://dx.doi.org/10.18653/v1/2022.acl-long.60