Program Synthesis from Polymorphic Refinement Types

Nadia Polikarpova   Ivan Kuraj   Armando Solar-Lezama

MIT CSAIL, USA

{polikarn,ivanko,asolar}@csail.mit.edu

Abstract

We present a method for synthesizing recursive functions that provably satisfy a given specification in the form of a polymorphic refinement type. We observe that such specifications are particularly suitable for program synthesis for two reasons. First, they offer a unique combination of expressive power and decidability, which enables automatic verification—and hence synthesis—of nontrivial programs. Second, a type-based specification for a program can often be effectively decomposed into independent specifications for its components, causing the synthesizer to consider fewer component combinations and leading to a combinatorial reduction in the size of the search space. At the core of our synthesis procedure is a new algorithm for refinement type checking, which supports specification decomposition.

We have evaluated our prototype implementation on a large set of synthesis problems and found that it exceeds the state of the art in terms of both scalability and usability. The tool was able to synthesize more complex programs than those reported in prior work (several sorting algorithms and operations on balanced search trees), as well as most of the benchmarks tackled by existing synthesizers, often starting from a more concise and intuitive user input.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; I.2.2 [*Automatic Programming*]: Program Synthesis

General Terms Languages, Verification

Keywords Program Synthesis, Functional Programming, Refinement Types, Predicate Abstraction

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA
ACM. 978-1-4503-4261-2/16/06...$15.00
http://dx.doi.org/10.1145/2908080.2908093

1. Introduction

The key to scalable program synthesis is modular verification. Modularity enables the synthesizer to prune candidates for different subprograms independently, whereby combinatorially reducing the size of the search space it has to consider. This explains the recent success of *type-directed* approaches to synthesis of functional programs [12, 14, 15, 27]: not only do ill-typed programs vastly outnumber well-typed ones, but more importantly, a type error can be detected long before the whole program is put together.

Simple, coarse-grained types alone are, however, rarely sufficient to precisely describe a synthesis goal. Therefore, existing approaches supplement type information with other kinds of specifications, such as input-output examples [1, 12, 27], or pre- and post-conditions [20, 21]. Alas, the corresponding verification procedures rarely enjoy the same level of modularity as type checking, thus fundamentally limiting the scalability of these techniques.

In this work we present a novel system that pushes the idea of type-directed synthesis one step further by taking advantage of *refinement types* [13, 33]: types decorated with predicates from a decidable logic. For example, imagine that a user intends to synthesize the function `replicate`, which, given a natural number n and a value x, produces a list that contains n copies of x. In our system, the user can express this intent by providing the following signature:

$$\texttt{replicate} :: n{:}\texttt{Nat} \rightarrow x{:}\alpha \rightarrow \{\nu{:}\ \texttt{List}\ \alpha \mid \texttt{len}\ \nu = n\}$$

Here, the return type is refined with the predicate $\texttt{len}\ \nu = n$, which restricts the length of the output list to be equal to the argument n; `Nat` is a shortcut for $\{\nu{:}\ \texttt{Int} \mid \nu \geq 0\}$, the type of integers that are greater or equal to zero[1]. Given this signature, together with the definition of `List` and a standard set of integer components (which include zero, decrement function, and inequalities), our system produces a provably correct implementation of `replicate`, shown in Fig. 1, within fractions of a second.

We argue that refinement types offer the user a convenient interface to a program synthesizer: the signature above is only marginally more complex than a conventional ML or Haskell type. Contrast that with example-based synthesis, which would

[1] Hereafter the bound variable of the refinement is always called ν and the binding is omitted.

```
replicate :: n:Nat → x:α → {List α | len ν = n}
replicate = λn . λx . if n ≤ 0
  then Nil
  else Cons x (replicate (dec n) x)
```

Figure 1. Refinement type signature of `replicate` and the code synthesized from this signature.

require a conventional type together with multiple input-output pairs, and in return provide much weaker correctness guarantees.

The `replicate` example is a perfect illustration of the power of parametric polymorphism for specifying program behavior. Even though the signature in Fig. 1 never says explicitly that each element of the output list must equal x, it nevertheless captures the semantics of `replicate` completely: since the function knows nothing about the type parameter α, it has no way of constructing any values of this type other than x. This surprising expressiveness of polymorphic types had been long known [40], but combined with refinements, it enables full-fledged higher-order reasoning within the type system: a caller of `replicate` can instantiate α with any refinement type, obtaining the fact that whenever x has a certain property, every element of the output list shares that same property.

Perhaps surprisingly, prior work on *liquid types* [19, 33, 36, 37] has shown that this type of higher-order reasoning can be fully automated for a large class of programs and properties. The *liquid type inference* algorithm [33] uses a combination of Hindley-Milner unification and least-fixpoint Horn clause solver based on predicate abstraction to discover refined instantiations for polymorphic types and ultimately reduce verification to proving quantifier-free formulas over simple refinement predicates, efficiently decidable by SMT solvers. The unique combination of expressive power and decidability offered by polymorphic refinement types makes them ideal for program synthesis.

Technical Challenges. Unfortunately, liquid type inference cannot be applied out of the box to the context of synthesis. Designed for the setting where, given a program, the goal is to construct its type, the inference algorithm starts from the leaves of the program, whose types are known, and propagates type information bottom-up, constructing types of terms from the types of their subterms. In program synthesis, however, the setting is different: here the top-level type is given, and the goal is to construct the program. One way to do so is to exhaustively explore program candidates, performing liquid type inference on each one, and then checking if the inferred type matches the given specification. While this approach does rule out many ill-typed partial programs, it fails to take advantage of the specification for guiding the search. A more promising approach would propagate type information *top-down* from the specification, using it to filter out irrelevant partial solutions.

Some program terms naturally support decomposing a specification into independent requirements for their subterms. For example, given a goal type T and assuming that the top-level construct of the program is a conditional with a known guard, we can pass T on to the two branches of the conditional, together with the appropriate path conditions derived from the guard, and proceed to check (or synthesize) them completely independently. Unfortunately, for other program terms there might be infinitely many ways to precisely decompose a specification. Take a function application, $f\ x$: the specification $f\ x$ :: `Nat` can be satisfied by requiring that f subtract one and x be positive, or that f add one, and x be greater than negative one, and so on. The challenge in this case is to find an over-approximate decomposition: that is, construct requirements on f and x that are necessary but generally not sufficient for the correctness of $f x$, yet are strong enough to filter out many incorrect subterms.

To address this challenge, we propose a new type checking mechanism for refinement types, which we dub *local liquid type checking*. At the heart of the new mechanism is a type system inspired by bidirectional type checking [30]. Bidirectional systems interleave top-down and bottom-up propagation of type information depending on the syntactic structure of the program; in this work we extend the bottom-up phase of bidirectional checking with top-down propagation of over-approximate type information, resulting in a *round-trip type checking* mechanism, which promotes modular checking of function applications. Additionally, we equip the type system with a novel *liquid abduction* rule, which enables modular checking of branching terms.

Refinement type checking involves solving subtyping constraints over unknown refinement types. The modularity requirement precludes our system from using the two techniques employed to this end by liquid type inference—Hindley-Milner unification and the least-fixpoint Horn solver—since both techniques are designed to work on complete programs and propagate type information bottom-up. Instead, local liquid type checking incorporates an algorithm for solving subtyping constraints incrementally, as it analyzes different parts of the program. Most notably, top-down propagation requires finding the *greatest fixpoint* solution to unknown refinements instead of the least, which is known to be fundamentally more expensive; we propose a practical implementation for this fixpoint computation, which we call MUSFix, inspired by an existing algorithm for enumerating minimal unsatisfiable subsets (MUSes) [22].

Results. We have combined local liquid type checking and exhaustive enumeration of program terms in a prototype program synthesizer called Synquid (for "SYNthesis with liQUID types"), which we evaluated on 64 synthesis problems from a variety of sources. The implementation, the benchmarks, and a web interface for Synquid are available from the tool repository [31].

Our evaluation indicates that the techniques described above work well for synthesizing programs that manipulate lists and trees, as well as data structures with complex invariants and custom user-defined data structures. Synquid was able to synthesize programs that are more complex than those previously reported in the literature, including five different sorting algorithms, and manipulations of binary search trees, AVL trees, and Red-Black trees. The evaluation also shows that the modularity

features of local liquid type checking and the MUSFIX solver are crucial for the performance of the system.

We compare our system with the existing synthesizers based on input-output examples [1, 12, 14, 27], and Hoare-style verification [20, 21], and demonstrate that SYNQUID can handle the majority of its competitors' most challenging benchmarks, taking a similar or shorter amount of time. In addition, compared with the example-based tools, SYNQUID's specifications are usually more concise and the generated solutions are provably correct; compared with the tools based on Hoare-style reasoning, SYNQUID can verify (and thus synthesize) more complex programs thanks to automatic refinement inference.

2. Overview

SYNQUID operates within a core ML-like language featuring conditionals, algebraic datatypes, pattern matching, parametric polymorphism, and fixpoints. We equip the language with *general decidable* refinement types, closely following the liquid types framework [19, 33, 36]. The type system includes refined base types of the form $\{B \mid \psi\}$, where ψ is a *refinement predicate* over the program variables and a special *value variable* ν, which does not appear in the program. Base types can be combined into dependent function types of the form $x\colon T_x \to T_2$, where x may appear in the refinement predicates of T_2. Our framework is agnostic to the exact logic of refinement predicates as long as validity of their boolean combinations is decidable; our prototype implementation uses the quantifier-free logic of arrays[2], uninterpreted functions, and linear integer arithmetic, which is sufficient for all the examples and benchmarks in this paper.

A *synthesis problem* is defined by (1) a goal refinement type T (2) a *typing environment* Γ and (3) a set of *logical qualifiers* $\mathbb{Q}$. A solution to the synthesis problem is a program term t that has the type T in the environment Γ. The environment contains type signatures of components available to the synthesizer (which may include datatype constructors, "library" functions, and local variables) as well as any path conditions that can be assumed when synthesizing t. Qualifiers are predicates from the refinement logic used as building blocks for unknown refinements and branch guards. Our system extracts an initial set of such predicates automatically from the goal type and the types of components; for all our experiments, the automatically extracted qualifiers were sufficient to synthesize all the necessary refinements, but in general the user might have to provide additional predicates.

Given a synthesis problem, SYNQUID constructs a candidate solution, by either by either picking a component in Γ or decomposing the problem into simpler subproblems and recursively obtaining a solution t_i to each one. Since the decomposition is generally incomplete, a candidate obtained by combining t_i's is not guaranteed to have the desired type T; to check if the candidate is indeed a solution, the system generates a *subtyping constraint*. If the constraint cannot be satisfied, the system backtracks to pick a different combination of solutions to subproblems (or a different decomposition altogether); the stronger the sub-goals produced during decomposition, the less SYNQUID has to backtrack. The rest of the section illustrates the details of this procedure and showcases various features of the specification language on a number of examples.

Example 1: Recursive Programs and Condition Abduction. We first revisit the `replicate` example from the introduction. We assume that the set of available components includes functions `0`, `inc` and `dec` on integers, as well as a list datatype whose constructors are refined with length information, expressed by means of an uninterpreted function (or *measure*) `len`.

```
0::{Int | ν = 0}
inc::x: Int → {Int | ν = x + 1}
dec::x: Int → {Int | ν = x - 1}

termination measure len :: List β → Nat
data List β where
  Nil :: {List β | len ν = 0}
  Cons :: β → xs: List β →
    {List β | len ν = len xs + 1}
```

Measure `len` also serves as the *termination metric* on lists (denoted with the **`termination`** keyword above): it maps lists to a type that has a predefined well-founded order in our language and thus enables termination checks for recursion on lists.

For the rest of the section, let us fix the set of logical qualifiers $\mathbb{Q}$ to $\{\star \leq \star, \star \neq \star\}$, where $\star$ is a placeholder that can be instantiated with any program variable or ν.

Given the specification

$$n\colon\texttt{Nat} \to x\colon\alpha \to \{\texttt{List}\ \alpha \mid \texttt{len}\ \nu = n\}$$

SYNQUID picks λ-abstraction as the top-level construct, and creates a synthesis subproblem for its body with a simpler goal type $\{\texttt{List}\ \alpha \mid \texttt{len}\ \nu = n\}$. The system need not consider other choices of the top-level construct, since every terminating program has an equivalent β-normal η-long form, where all functions are fully applied and the head of each application is a variable; moreover, the above decomposition is precise, since any solution to the subproblem will satisfy the top-level goal.

As part of the decomposition, the arguments $n : \texttt{Nat}$ and $x : \alpha$ are added to the environment, together with the function `replicate` itself, to account for the possibility that it may be recursive. In order to ensure termination of recursive calls, the system weakens the type of `replicate` in the environment to

$$n'\colon \{\texttt{Int} \mid 0 \leq \nu < n\} \to x'\colon \alpha \to \{\texttt{List}\ \alpha \mid \texttt{len}\ \nu = n'\}$$

demanding that the first argument be strictly decreasing. SYNQUID picks n as the termination metric in this case, since it is the only argument whose type has an associated well-founded order.

In the body of the function, the top-level construct might be a branching term. Rather than exploring this possibility explicitly, SYNQUID adds a fresh *predicate unknown* P_0 as a path condition to the environment, and then searches for a branch-free program term that satisfies the specification assuming P_0. Each candidate branch-free term t is validated by solving a subtyping constraint; as part of this process, SYNQUID discovers the weakest P_0 that

[2] Arrays are used to model sets.

makes t satisfy the specification. In case t is valid unconditionally, the weakest such P_0 is `True`, and no branch is generated.

Suppose the first branch-free term that SYNQUID considers is `Nil`; this choice results in a subtyping constraint $n\!:\!\texttt{Nat};x\!:\!\alpha;P_0 \vdash \{\texttt{List}\ \beta' \mid \texttt{len}\ \nu = 0\} <: \{\texttt{List}\ \alpha \mid \texttt{len}\ \nu = n\}$ where β' is a free type variable. The constraint imposes two requirements: (i) the *shapes* of the two types (i.e. their underlying unrefined types) must have a unifier [29] and (ii) the refinements of the subtype must subsume those of the supertype under the assumptions encoded in the environment. The constraint above gives rise to a unifier $[\beta' \mapsto \{\alpha \mid P_1\}]$ (where P_1 is a fresh predicate unknown) and two Horn constraints: $P_0 \wedge P_1 \Rightarrow \top$ and $0 \leq n \wedge P_0 \wedge \texttt{len}\ \nu = 0 \Rightarrow \texttt{len}\ \nu = n$. SYNQUID uses the MUSFIX Horn solver (Sec. 3.6) to find the weakest assignment of *liquid formulas* to P_0 and P_1 that satisfies both Horn constraints. A liquid formula is a conjunction of atomic formulas, obtained by replacing $\star$-placeholders in each qualifier in $\mathbb{Q}$ with appropriate variables. If for some P_i no valid assignment exists, or the weakest valid assignment is a contradiction, the candidate program is discarded.

In our example, MUSFIX discovers the weakest assignment $\mathcal{L} = [P_0 \mapsto n \leq 0, P_1 \mapsto \top]$, effectively *abducing* the necessary branching condition. Since the condition is not trivially true, the system proceeds to synthesize the remaining branch under the path condition $\neg(n \leq 0)$. A similar strategy for generating branching programs has been successfully employed in several existing synthesis tools [1, 4, 20, 21] and is commonly referred to as *condition abduction*. Each condition abduction technique faces the challenge of searching a large space of potential conditions efficiently; our approach, which we dub *liquid abduction*, addressed this challenge by restricting conditions to liquid formulas and using MUSFIX to explore the space of liquid formulas efficiently.

The remaining branch has to deal with the harder case of $n > 0$. When enumerating candidates for this branch, SYNQUID eventually decides to apply the `replicate` component (that is, make a recursive call), and searches for the parameters via recursive application of the synthesis procedure. At this point, the strong precondition on the argument m, $0 \leq \nu < n$, which arises from the termination requirement, enables filtering candidate arguments locally, before synthesizing the rest of the branch. In particular, the system will discard the candidates n and `inc` n right away, since they fail to produce a value strictly less than n.

Example 2: Complex Data Structures and Invariant Inference. Assuming comparison operators in our logic are generic, we can define the type of binary search trees as follows:

```
termination measure size :: BST α → Int
measure keys :: BST α → Set α
data BST α where
  Empty :: {BST α | keys ν = []}
  Node :: x:α → l:BST{α | ν < x} → r:BST{α | x < ν}
    → {BST α | keys ν = keys l + keys r + [x]}
```

According to this definition, one can obtain a `BST` either by taking an empty tree, or by composing a node with key x and two `BST`s, l and r, in which all keys are, respectively, strictly less and strictly greater than x. The type is additionally refined by the measure `keys`, which denotes the set of all keys in the tree, and a termination measure `size` (size-related refinements are omitted in the interest of space).

The following type specifies insertion into a `BST`:

```
insert :: x:α → t:BST α →
                  {BST α | keys ν = keys t + [x]}
```

From this specification, SYNQUID generates the following implementation within two seconds:

```
insert = λx . λt . match t with
  | Empty → Node x Empty Empty
  | Node y l r → if x ≤ y ∧ y ≤ x
      then t
      else if y ≤ x
        then Node y l (insert x r)
        else Node y (insert x l) r
```

Pattern matching in this example is synthesized using a special case of liquid abduction: type-checking the term `Node x Empty Empty` against the goal type $\{\texttt{BST}\ \alpha \mid \texttt{keys}\ \nu = \texttt{keys}\ t + [x]\}$, causes the system to abduce the condition $\texttt{keys}\ t = []$, which implies a match on t.

The challenging aspect of this example is reasoning about sortedness. For example, for the term $\texttt{Node}\ y\ l\ (\texttt{insert}\ x\ r)$ to be type-correct, the recursive call must return the type $\texttt{BST}\ \{\alpha \mid y < \nu\}$. This type does not appear explicitly in the user-provided signature for `insert`; in fact, verifying this program requires discovering a nontrivial inductive invariant of `insert` (that adding a key greater than some value z into a tree with keys greater than z again produces a tree with keys greater than z), which puts this and similar examples beyond reach of existing synthesizers based on Hoare-style reasoning [20, 21].

In our framework, this property is easily inferred by the Horn constraint solver in combination with polymorphic recursion. When `insert` is added to the environment, its type is generalized to $\forall \beta . x : \beta \to u : \{\texttt{BST}\ \beta \mid \texttt{size}\ u < \texttt{size}\ t\} \to \{\texttt{BST}\ \beta \mid \texttt{keys}\ \nu = \texttt{keys}\ t + \{x\}\}$. At the site of the recursive call, the precondition of $\texttt{Node}\ y\ l$ imposes a constraint that simplifies to $\texttt{BST}\ \beta <: \texttt{BST}\ \{\alpha \mid y < \nu\}$, which leads to instantiating $[\beta \mapsto \{\alpha \mid P_0\}]$ and $[P_0 \mapsto y < \nu]$.

Importantly, due to *round-trip type checking* (Sec. 3.2), this assignment is discovered before the two arguments to `insert` are synthesized, which has the effect of propagating the requirement imposed by `Node` *top-down* through the application of `insert` onto its arguments. In particular, using the goal type $\{\alpha \mid y < \nu\}$ for the first argument of `insert`, the system can immediately discard the candidate `y`, while trying `x` succeeds and leads to the abduction of the branch condition $y \leq x$. As our evaluation shows, disabling this type of early filtering increases the synthesis time for this example from less than two seconds to over two minutes.

Example 3: Abstract Refinements. Using refinement types as an interface to synthesis raises the question of their expressiveness. Restricting refinements to decidable logics fundamentally

limits the class of programs they can fully specify, and for other programs writing a refinement type might be possible but cumbersome compared to providing a set of input-output examples or a specification in a richer language. The previous examples suggest that refinement types are effective for specifying programs that manipulate data structures with nontrivial universal and inductive invariants. In this example we demonstrate how extending the type system with *abstract refinements* allows us to express a wider class of properties, for example, talk about the order of list elements.

Abstract refinements, proposed in [36], enable explicit quantification over refinements of datatypes and function types. For example, a list datatype can be parameterized by a binary relation r that must hold between any in-order pair of elements in the list:

```
data RList α <r::α → α → Bool> where
  Nil::RList α <r>
  Cons::x:α → RList {α | r x ν} <r> → RList α <r>
```

On the one hand this enables concise definitions of lists with various inductive properties as instantiations of `RList`:

```
IList α = RList α <λx λy . x ≤ y> -- Increasing list
UList α = RList α <λx λy . x ≠ y> -- Unique list
List α = RList α <λx λy . True> -- Unrestricted list
```

On the other hand, making list-manipulating functions polymorphic in this relation, provides an elegant way to specify order-related properties. Consider the following type for list reversal:

```
reverse::<r::α → α → Bool> . xs: RList α <r> →
  {RList α <λx λy . r y x> | len ν = len xs}
```

It says that whatever relation holds between every *in-order* pair of elements of the input list, also has to hold between every *out-of-order* pair of elements of the output list. This type does not restrict the applicability of `reverse`, since at the call site r can always be instantiated with `True`; the implementation of `reverse`, however, has to be correct to any value of r, which leaves the synthesizer no choice but reverse the order of list elements. Given the above specification and a component that appends an element to the end of the list (specified in a similar fashion), SYNQUID synthesizes the standard implementation of list reversal.

Example 4: Higher-Order Combinators and Auxiliary Function Discovery. Complex programs might require recursive auxiliary functions. Discovering specifications for such functions automatically is a difficult task, akin to *lemma discovery* in theorem proving [7, 16, 25], which largely remains an open problem. SYNQUID expects users to provide the high-level insight about a complex algorithm in the form of auxiliary function signatures. For example, if the goal is to synthesize a list sorting function with the following signature

```
sort::xs: List α → {IList α | elems ν = elems xs}
```

(where `elems` denotes the set of list elements), the user can express the insight behind different sorting algorithms by providing different auxiliary functions: insertion into a sorted list for *insertion sort*, splitting and merging for *merge sort*, or partitioning and concatenation for *quick sort*. Naturally, the implementation of the auxiliary functions can in turn be synthesized, but coming up with their specification is the creative step that generally requires user interaction, and can be considered a major hurdle on the path to fully automatic synthesis.

It turns out, however, that replacing general recursion with higher-order combinators such as `map` and `fold`—a style widely used and highly encouraged in functional programming—makes it possible to infer requirements on the auxiliary function from the specification of the main program. This is one of the main insights behind the synthesizer λ^2 [12], which relies on hard-coded rules for propagating input-output examples top-down through common combinators. SYNQUID supports this top-down propagation of specifications out of the box thanks to the combination of refinement types and polymorphism.

Consider the following type for function `foldr`, which folds a binary operation `f` over a list `ys` from right to left:

```
foldr::<p::List β → γ → Bool> .
  f: (t: List β → h:β → acc: {γ | p t ν} →
       {γ | p (Cons h t) ν}) →
  seed: {γ | p Nil ν} →
  ys: List β → {γ | p ys ν}
```

The shape of this type is slightly different from the usual signature of `foldr`: the operation `f` takes an extra ghost argument `t`, which denotes the part of the list that has already been folded[3]. The type of is parametrized by a binary relation `p` that `foldr` establishes between the input list `ys` and the output; it requires that the relationship hold between the empty list and the `seed`, and that applying `f` to a head element `h` and the result of folding a tail list `t` yield a result that satisfies the relationship with `Cons h t` (in other words, `p` plays the role of a loop invariant). Note that folding a list left-to-right (`foldl`) requires a more complex specification that cannot currently be expressed within the SYNQUID type system.

What happens if we ask SYNQUID to synthesize `sort`, while providing `foldr` as the only component? When trying out an application of `foldr`, round-trip type checking handles its higher-order argument, `f`, in a special way, since in our type system, as in [33], `f` cannot appear in the result type of `foldr`. Consequently, the exact value of `f` is not required to determine the type of the application, which gives SYNQUID the freedom to synthesize it independently from the rest of the program.

The tool quickly figures out that `foldr ?? Nil xs` has the required type `{IList α | elems ν = elems xs}`, given the following assignment to `foldr`'s type and predicate variables: $[\beta \mapsto \alpha, \gamma \mapsto \texttt{IList}\ \alpha, \texttt{p} \mapsto \lambda as.\lambda bs.\texttt{elems}\ bs = \texttt{elems}\ as]$. Now that SYNQUID comes back to the task of filling in the first argument of `foldr`, its required type has been determined entirely as

```
t: List α → h:α →
  acc: {IList α | elems ν = elems t} →
  {IList α | elems ν = elems (Cons h t)}
```

[3] Extending SYNQUID with *bounded refinement types* [39] would enable a more natural specification without the ghost argument.

(where `elems (Cons h t)` is expanded into `[h] + elems t` using the definition of the `elems` measure in the type of `Cons`); in other words, the auxiliary function must insert `h` into a sorted list `acc`. Treating this inferred signature as an independent synthesis goal, SYNQUID easily synthesizes a recursive program for insertion into a sorted list, and thus completes the following implementation of insertion sort without requiring any hints from the user, apart from a general recursion scheme:

```
sort = λxs . foldr f Nil xs
  where f = λt . λh . λacc .
    match acc with
      Nil → Cons h Nil
      Cons z zs → if h ≤ z
        then Cons h (Cons z zs)
        else Cons z (f zs h zs)
```

The next section gives a formal account of the SYNQUID language and type system, and details its modular type checking mechanism, which enables scalable synthesis.

3. The SYNQUID Language

The central goal of this section is to develop a type checking algorithm for a core programming language with refinement types that is geared towards candidate validation in the context of synthesis. This context imposes two important requirements on the type checking mechanism which are necessary for the synthesis procedure to be automatic and scalable. The first one has to do with the amount of type inference: the mechanism can expect top-level type annotations—this is how users specify synthesis goals—but cannot rely on any annotations beyond that; in particular, the types of all polymorphic instantiations and arguments of anonymous functions must be inferred. The second requirement is to detect type errors *locally*: intuitively, if a subterm of a program causes a type error independently of its context, the algorithm should be able to report that error without analyzing the context.

We build our type checking mechanism as an extension to the the liquid types framework [19, 33], which uses a combination of Hidley-Milner unification and a Horn solver to infer refinement types. The original liquid type inference algorithm is not designed for synthesis, and thus makes different trade-offs: in particular it does not satisfy the locality requirement. Our type checking mechanism achieves locality based on three key ideas. First, we apply bidirectional type checking [30] to refinement types and reinforce it with additional top-down propagation of type information, arriving at *round-trip type checking* (Sec. 3.2); we then further improve locality of rules for for function applications and branching statements (Sec. 3.4). Second, we develop a new algorithm for converting subtyping constraints into horn clauses, which is able to do so incrementally as the constraints are issued before analyzing the whole program (Sec. 3.5). Finally, we propose a new, efficient implementation for a greatest-fixpoint Horn solver (Sec. 3.6). In the interest of space we omit abstract refinements (see Sec. 2) from the formalization; [36] has

ψ	$::=$		*Refinement term:*
		$\mid \top \mid \bot \mid 0 \mid + \mid$ ... *(varies)*	interpreted symbol
		$\mid x$	uninterpreted symbol
		$\mid \psi\ \psi$	application
Δ	$::=$		*Sort:*
		$\mid \mathbb{B} \mid \mathbb{Z} \mid$ ... *(varies)*	interpreted
		$\mid \delta$	uninterpreted
t	$::= e \mid b \mid f$		*Program term*
e	$::=$		*E-term:*
		$\mid x$	variable
		$\mid e\ e \mid e\ f$	application
b	$::=$		*Branching term:*
		$\mid \mathbf{if}\ e\ \mathbf{then}\ t\ \mathbf{else}\ t$	conditional
		$\mid \mathbf{match}\ e\ \mathbf{with} \mid_i \mathsf{C}_i\langle x_i^j \rangle \mapsto t_i$	match
f	$::=$		*Function term:*
		$\mid \lambda x.t$	abstraction
		$\mid \mathbf{fix}\ x.t$	fixpoint
B	$::=$		*Base type:*
		$\mid \mathsf{Bool} \mid \mathsf{Int}$	primitive
		$\mid D\ T_i$	datatype
		$\mid \alpha$	type variable
T	$::=$		*Type:*
		$\mid \{B \mid \psi\}$	scalar
		$\mid x\colon T \to T$	function
S	$::= \forall \alpha_i.T$		*Type schema*
C	$::= \cdot \mid x\colon T; C$		*Context*
$\hat{T}$	$::= \mathbf{let}\ C\ \mathbf{in}\ T$		*Contextual Type*

Figure 2. Terms and types.

shown that integrating this mechanism into the type system that already supports parametric polymorphism is straightforward.

In Sec. 3.7 we derive *synthesis rules* from the modular type checking rules; in doing so we follow previous work on type-directed synthesis [17, 27], which has shown how to turn type checking rules for a language into synthesis rules for the same language.

3.1 Syntax and Types

Fig. 2 shows the syntax of the SYNQUID language.

Terms. Unlike previous work, we differentiate between the languages of refinements and programs. The former consists of refinement terms ψ, which have sorts Δ; the exact set of interpreted symbols and sorts depends on the chosen refinement logic. We refer to refinement terms of the Boolean sort $\mathbb{B}$ as formulas.

The language of programs consists of program terms t, which we split, following [27] into *elimination* and *introduction* terms (E-terms and I-terms for short). Intuitively, E-terms—variables and applications—propagate type information bottom-up, *composing* a complex property from properties of their components; I-terms propagate type information top-down, *decomposing* a complex requirement into simpler requirements for their components. Note that conditional guards, match scrutinees, and left-hand sides of applications are restricted to E-terms. We

Well-Formed Types $\boxed{\Gamma \vdash \hat{T}}$

$$\text{WF-SC}\ \frac{\Gamma;\nu : B \vdash \psi}{\Gamma \vdash \{B \mid \psi\}} \qquad \text{WF-CTX}\ \frac{\Gamma;C \vdash T}{\Gamma \vdash \texttt{let}\ C\ \texttt{in}\ T}$$

$$\text{WF-FO}\ \frac{\Gamma \vdash \{B \mid \psi\} \quad \Gamma;x:\{B \mid \psi\} \vdash T}{\Gamma \vdash x\colon \{B \mid \psi\} \to T}$$

$$\text{WF-HO}\ \frac{T_x \text{ non-scalar} \quad \Gamma \vdash T_x \quad \Gamma \vdash T}{\Gamma \vdash T_x \to T}$$

Subtyping $\boxed{\Gamma \vdash T <: T'}$

$$\text{<:-SC}\ \frac{\Gamma \vdash B <: B' \quad \mathsf{Valid}([\![\Gamma]\!]_{\psi \Rightarrow \psi'} \wedge \psi \Rightarrow \psi')}{\Gamma \vdash \{B \mid \psi\} <: \{B' \mid \psi'\}}$$

$$\text{<:-FUN}\ \frac{\Gamma \vdash T_y <: T_x \quad \Gamma;y:T_y \vdash [y/x]T <: T'}{\Gamma \vdash x\colon T_x \to T <: y\colon T_y \to T'}$$

$$\text{<:-DT}\ \frac{\Gamma \vdash T_i <: T_i'}{\Gamma \vdash D\,T_i <: D\,T_i'} \qquad \text{<:-REFL}\ \frac{}{\Gamma \vdash B <: B}$$

Figure 3. Well-formedness and subtyping.

further separate I-terms into branching terms—conditionals and matches—and function terms—abstractions and fixpoints—and disallow branching terms on the right-hand side of application. This normal form is required to enable precise and efficient local type checking, as explained below. It does not fundamentally restrict the expressiveness of the language: every terminating program in lambda calculus can be translated to SYNQUID by first applying a standard β-normal η-long form [15, 27] and then pushing branching terms outside of applications, guards, and scrutinees.

Types and Schemas. A SYNQUID type is either a scalar—base type refined with a formula—or a dependent function type. Base types include primitives, type variables, and user-defined datatypes with zero or more type parameters. Datatype constructors are represented simply as functions that must have the type $\forall \alpha_1 \dots \alpha_m . T_1 \to \dots \to T_k \to D\ \alpha_1 \dots \alpha_m$. A *contextual type* is a pair of a sequence of variable bindings and a type that can mention those variables; contextual types are useful for precise type checking of applications, as explained in Sec. 3.2.

SYNQUID features ML-style polymorphism, where type variables are universally quantified at the outermost level to yield type schemas. Unlike ML, we restrict type variables to range only over scalars, which gives us the ability to determine whether a type is a scalar, even if it contains free type variables. We found this restriction not to be too limiting in practice.

Environments, Well-Formedness, and Subtyping. A *typing environment* Γ is a sequence of variable bindings $x:T$ and path conditions ψ; we denote conjunction of all path conditions in an environment as $\mathsf{P}(\Gamma)$. A formula ψ is *well-formed* in the environment Γ, written $\Gamma \vdash \psi$, if it is of a Boolean sort and each of its free variables is bound in Γ to a type that is consistent with its sort in ψ. Well-formedness extends to types as shown in Fig. 3. Note the two different rules for first-order and higher-order function types: in a function type $x\colon T_1 \to T_2$, T_2 may reference the formal argument x only if T_1 is a scalar type (that is, only first-order function types are dependent).

The *subtyping* relation $\Gamma \vdash T <: T'$ is relatively standard (Fig. 3). For simplicity, we consider all datatypes covariant in their type parameters (rule <:-DT); if need be, variance can be selected per type parameter depending on whether it appears positively or negatively in the constructors. The crucial part is the rule <:-SC, which reduces subtyping between scalar types to implication between their refinements, under the assumptions extracted from the environment. Since the refinements are drawn from a decidable logic, this implication is decidable. The function that extracts assumptions from the environment is parametrized by a formula ψ and returns a conjunction of all path conditions and refinements of all variables mentioned in ψ or the path conditions:

$$[\![\Gamma]\!]_\psi = \mathsf{P}(\Gamma) \wedge \mathsf{B}_{\mathsf{FV}(\mathsf{P}(\Gamma)) \cup \mathsf{FV}(\psi)}(\Gamma)$$

where

$$\mathsf{B}_v(\Gamma;x:\{B \mid \psi\}) = \begin{cases} [x/\nu]\psi \wedge \mathsf{B}_{v \setminus \{x\} \cup \mathsf{FV}(\psi)}(\Gamma) & (x \in v) \\ \mathsf{B}_v(\Gamma) & \text{(otherwise)} \end{cases}$$

$$\mathsf{B}_v(\Gamma;x:T) = \mathsf{B}_v(\Gamma) \quad (T \text{ non-scalar})$$

$$\mathsf{B}_v(\cdot) = \top$$

This definition limits the effect of an environment variable with an inconsistent refinement to only those subtyping judgments that (transitively) mention that variable.

3.2 Round-Trip Type Checking

This section describes the core of SYNQUID's type system. It is inspired by bidirectional type checking [30], which interleaves top-down and bottom-up propagation of type information depending on the syntactic structure of the program, with the goal of making type checks more local. Bidirectional typing rules use two kinds of typing judgments: an *inference* judgment, written $\Gamma \vdash e \uparrow T$, states that the term t generates type T in the environment Γ; a *checking* judgment, $\Gamma \vdash t \downarrow T$, states that the term t checks against a known type T in the environment Γ. Accordingly, all typing rules can be split into inference and checking rules, depending on the judgment they derive. Bidirectional type checking rules for SYNQUID can be found in the technical report [32].

In a bidirectional system, analyzing a program starts with propagating its top-level type annotation top-down using checking rules, until the system encounters a term t to which no checking rule applies. At this point the system switches to bottom-up mode, infers the type T' of t, and checks if T' is a subtype of the goal type; if the check fails, t is rejected. Bidirectional type propagation is "all-or-nothing": once a checking problem for a term cannot be decomposed perfectly into checking problems for its subterms, the system abandons all information about the goal type and switches to purely bottom-up inference. Our insight is that some information from the goal type can be retained in the bottom-up phase, leading to more local error detection. To this end, we modify the bidirectional inference judgment into a *strengthening* judgment $\Gamma \vdash t \downarrow T \uparrow T'$, which reads as follows: in the environment Γ, term t checks against a known type T *and*

Type Strengthening $\boxed{\Gamma \vdash_{\mathbb{Q}} e \downarrow T \uparrow \hat{T}'}$

$$\textsc{VarSc}\ \frac{\Gamma(x) = \{B \mid \psi\} \quad \Gamma \vdash \{B \mid \psi\} <: T}{\Gamma \vdash_{\mathbb{Q}} x \downarrow T \uparrow \{B \mid \nu = x\}}$$

$$\textsc{Var}\forall\ \frac{\Gamma(x) = \forall \alpha_i . T' \quad \Gamma \vdash_{\mathbb{Q}} T_i \quad \Gamma \vdash [T_i/\alpha] T' <: T}{\Gamma \vdash_{\mathbb{Q}} x \downarrow T \uparrow [T_i/\alpha] T'}$$

$$\textsc{AppFO}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathbb{Q}} e_1 \downarrow \{B \mid \bot\} \to T \uparrow \texttt{let}\ C_1\ \texttt{in}\ (x\colon \{B \mid \psi\} \to T') \\ \Gamma; C_1 \vdash_{\mathbb{Q}} e_2 \downarrow \{B \mid \psi\} \uparrow \texttt{let}\ C_2\ \texttt{in}\ T_x \\ \Gamma; C_1; C_2; x\colon T_x \vdash T' <: T\end{array}}{\Gamma \vdash_{\mathbb{Q}} e_1\ e_2 \downarrow T \uparrow \texttt{let}\ C_1; C_2; x\colon T_x\ \texttt{in}\ T'}$$

$$\textsc{AppHO}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathbb{Q}} e \downarrow \mathsf{bot} \to T \uparrow \texttt{let}\ C\ \texttt{in}\ (T'_x \to T') \\ \Gamma; C \vdash_{\mathbb{Q}} f \downarrow T'_x\end{array}}{\Gamma \vdash_{\mathbb{Q}} e\ f \downarrow T \uparrow \texttt{let}\ C\ \texttt{in}\ T'}$$

Type Checking $\boxed{\Gamma \vdash_{\mathbb{Q}} t \downarrow S}$

$$\textsc{IE}\ \frac{\Gamma \vdash_{\mathbb{Q}} e \downarrow T \uparrow \hat{T}'}{\Gamma \vdash_{\mathbb{Q}} e \downarrow T}$$

$$\textsc{Abs}\ \frac{\Gamma; y\colon T_x \vdash_{\mathbb{Q}} t \downarrow [y/x] T}{\Gamma \vdash_{\mathbb{Q}} \lambda y.t \downarrow (x\colon T_x \to T)}$$

$$\textsc{If}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathbb{Q}} e \downarrow \mathsf{Bool} \uparrow \texttt{let}\ C\ \texttt{in}\ \{\mathsf{Bool} \mid \psi\} \\ \Gamma; C; [\top/\nu]\psi \vdash_{\mathbb{Q}} t_1 \downarrow T \quad \Gamma; C; [\bot/\nu]\psi \vdash_{\mathbb{Q}} t_2 \downarrow T\end{array}}{\Gamma \vdash_{\mathbb{Q}} \texttt{if}\ e\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \downarrow T}$$

$$\textsc{Match}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathbb{Q}} e \downarrow \mathsf{top} \uparrow \texttt{let}\ C\ \texttt{in}\ \{D\ T_k \mid \psi\} \\ \mathsf{C}_i = T_i^j \to \{D\ T_k \mid \psi'_i\} \quad \Gamma_i = \{x_i^j\colon T_i^j\}; [x'/\nu]\psi'_i \\ \Gamma; C; [x'/\nu]\psi; \Gamma_i \vdash_{\mathbb{Q}} t_i \downarrow T\end{array}}{\Gamma \vdash_{\mathbb{Q}} \texttt{match}\ e\ \texttt{with}\ |_i\ \mathsf{C}_i \langle x_i^j \rangle \mapsto t_i \downarrow T}$$

$$\textsc{TAbs}\ \frac{\Gamma \vdash_{\mathbb{Q}} t \downarrow T \quad \alpha_i \text{ not free in } \Gamma}{\Gamma \vdash_{\mathbb{Q}} t \downarrow \forall \alpha_i . T}$$

$$\textsc{Fix}\ \frac{\Gamma; x\colon S^{\prec} \vdash_{\mathbb{Q}} t \downarrow S}{\Gamma \vdash_{\mathbb{Q}} \texttt{fix}\ x.t \downarrow S}$$

Figure 4. Rules of round-trip type checking.

generates a stronger type T'. We call the resulting type system *round-trip*, since it propagates types top-down and then back up.

Derivation rules for round-trip type checking are presented in Fig. 4. All judgments are parametrized by the set of qualifiers $\mathbb{Q}$, used to construct unknown refinements as explained below. Checking rules encode the way a checking judgment for an I-term t is decomposed into simpler checking judgments for its components. Strengthening rules encode the way a goal type for an E-term e is decomposed into *over-approximate* goal types for its subterms, which are necessary but in general not sufficient for correctness, while the precise type of e is constructed from the inferred types of its subterms. A round-trip type checker starts with a top-down phase, just as a bidirectional one would; when it encounters an E-term, it applies the corresponding strengthening rule and discards the inferred type (see rule IE). Thus, instead of detecting type errors at the boundary between I- and E-terms, the round-trip system performs local checks for each variable and function application.

In order to support goals types with an underspecified shape (as required for match scrutinees and higher-order applications), we augment SYNQUID with top and bot types, which are, respectively, a supertype and a subtype of every type. Note that these types are ignored when computing the logical representation of the environment $[\![\Gamma]\!]_\psi$, since they are not considered scalar. Also note that the precision of round-trip type checking crucially relies on the fact that only E-terms appear in strengthening judgments; this is why SYNQUID bans branching terms from function arguments, conditional guards, and match scrutinees.

Polymorphic instantiations. The rule VAR$\forall$, which handles polymorphic instantiations, replaces type variables α_i with types T_i chosen nondeterministically to satisfy all subtyping checks[4]. In order to tame this nondeterminism, following [33], we restrict T_is to *liquid types*. A formula ψ is *liquid* in Γ with qualifiers $\mathbb{Q}$, written $\Gamma \vdash_{\mathbb{Q}} \psi$, if it is a conjunction of well-formed formulas, each of which is obtained from a qualifier in $\mathbb{Q}$ by substituting $\star$-placeholders with variables. This notion extends to types, $\Gamma \vdash_{\mathbb{Q}} T$, in a way analogous to well-formedness (Fig. 3). Note that the set of all liquid formulas in a given environment is finite, and so is the set of all liquid types with a fixed shape. Sec. 3.5 and Sec. 3.6 present a deterministic algorithm for finding the types T_i.

Applications. The application rules APPFO and AppHO are the core of the round-trip type system: they are responsible for propagating partial type information down to the left-hand side of an application. The type system distinguishes between first-order and higher-order applications, since in a function type $x\colon T_1 \to T_2$, T_2 cannot mention x if T_1 is a function type (see Fig. 3). As a result, a higher-order application always yields the type T_2 independently of the argument. If instead T_1 is a scalar type, we have to replace x inside T_2 with the actual argument of the application. Unfortunately, we cannot assign the application $e_1\ e_2$ the type $[e_2/x]T_2$, since e_2 is a program term, which does not necessarily have a corresponding refinement precisely capturing its semantics. We address this problem by assigning $e_1\ e_2$ a *contextual type* $\texttt{let}\ C\ \texttt{in}\ T_2$, where the context C binds the variable x to the precise type of e_2.

Example. We demonstrate the local error detection enabled by rule APPFO on the following type-checking problem:

$$\Gamma \vdash_{\mathbb{Q}} \mathtt{append}\ xs\ xs \downarrow \{\mathtt{List\ Pos} \mid \mathtt{len}\ \nu = 5\}$$

where `Pos` is an abbreviation for $\{\mathtt{Int} \mid \nu > 0\}$ and Γ contains the following bindings:

$xs\colon \{\mathtt{List\ Nat} \mid \mathtt{len}\ \nu = 2\}$;

$$\mathtt{append}\colon \forall \alpha . l\colon \{\mathtt{List}\ \alpha \mid \mathtt{len}\ \nu \geq 0\} \to r\colon \{\mathtt{List}\ \alpha \mid \mathtt{len}\ \nu \geq 0\} \to \{\mathtt{List}\ \alpha \mid \mathtt{len}\ \nu = \mathtt{len}\ l + \mathtt{len}\ r\}$$

Intuitively, the constraint on the length of the output list is hard to verify without analyzing the whole expression, while the mismatch in the type of the list elements can be easily found without considering the second argument of `append`. Refinement types provide precise means to distinguish those cases: the length-related refinement of `append` is dependent on the

[4] The same rule handles monomorphic non-scalar variables, assuming zero type variables.

arguments l and r, whereas the type of the list elements cannot possibly mention l or r, since it has to be well-formed in a scope where these variables are not defined.

Applying the APPFO rule twice to the judgment above yields $\Gamma \vdash_{\mathbb{Q}} \texttt{append} \downarrow \{B_0 \mid \bot\} \to \{B_1 \mid \bot\} \to \{\texttt{List Pos} \mid \texttt{len}\ \nu = 2\}$, where the base types B_0 and B_1 are yet to be inferred. Applying VAR$\forall$, and decomposing the resulting subtyping check with $<:$-FUN, we get

$$\Gamma; l: \{B_0 \mid \bot\}; r: \{B_1 \mid \bot\} \vdash$$
$$\{\texttt{List}\ T_0 \mid \texttt{len}\ \nu = \texttt{len}\ l + \texttt{len}\ r\} <: \{\texttt{List Pos} \mid \texttt{len}\ \nu = 2\}$$

Using $<:$-SC, this judgment can be decomposed into an implication on refinements—vacuous thanks to the types of l and r—and subtyping on base types, $\texttt{List}\ T_0 <: \texttt{List Pos}$, which is not vacuous since here l and r are out of scope. The first argument of `append` is checked against the type $\texttt{List}\ T_0$ (in the second premise of APPFO), which imposes a subtyping check $\Gamma \vdash \texttt{List Nat} <: \texttt{List}\ T_0$. Since no type T_0 satisfies both subtyping relations, the type checker rejects the term $\texttt{append}\ xs$.

Recursion. Another rule in Fig. 4 that deserves some discussion is FIX, which comes with a termination check. In the context of synthesis, termination concerns are impossible to ignore, since non-terminating recursive programs are always simpler than terminating ones, and thus would be synthesized first if considered correct. The FIX rule gives the "recursive call" a termination-weakened type $S^{\prec}$, which intuitively denotes "S with strictly smaller arguments". The exact definition of termination-weakening is a parameter to our system. Our implementation provides a predefined well-founded order on primitive base types, and allows the user to define one on datatypes by mapping them to primitive types using termination metrics; then $S^{\prec}$ is defined as a lexicographic order on the tuple of all arguments of S that have an associated well-founded order.

3.3 Soundness and Completeness

We show soundness and completeness of round-trip type checking *relative* to purely bottom-up liquid type inference [33]. Detailed proofs are available in the technical report [32].

Round-trip type checking is *sound* in the sense that whenever a SYNQUID term t type-checks against a schema S, $\Gamma \vdash_{\mathbb{Q}} t \downarrow S$, there exist a set of qualifiers $\mathbb{Q}'$ and a schema S', such that the bottom-up system infers S' for t, $\Gamma \vdash_{\mathbb{Q}'} t :: S$, and $\Gamma \vdash S' <: S$. Note that bottom-up inference might require strictly more qualifiers than type checking: in the bottom-up system, generating types for branching statements and abstractions imposes the requirement that these types be liquid; the round-trip system obtains the types of those terms by decomposing the goal type, thus the liquid restriction does not apply. In practice the difference is irrelevant, since the type inference algorithm can extract the missing qualifiers from the top-level goal type and the preconditions of component functions. Thus, if $\mathbb{Q}$ contains a sufficient set of qualifiers such that the goals schema is liquid ($\Gamma \vdash_{\mathbb{Q}} S$) and the preconditions of component function are liquid, which we denote as $\vdash_{\mathbb{Q}} \Gamma$, then we can take $\mathbb{Q}' = \mathbb{Q}$.

Consistency $\boxed{\Gamma \vdash T \wedge: T'}$

$$\wedge:\text{-SC}\ \frac{\Gamma \vdash B \wedge: B' \qquad \mathsf{Sat}(\llbracket \Gamma \rrbracket_{\psi \wedge \psi'} \wedge \psi \wedge \psi')}{\Gamma \vdash \{B \mid \psi\} \wedge: \{B' \mid \psi'\}}$$

$$\wedge:\text{-FUN}\ \frac{\Gamma; x: T_x \vdash T \wedge: [x/y]T'}{\Gamma \vdash x: T_x \to T \wedge: y: T_y \to T'}$$

$$\wedge:\text{-DT}\ \frac{\Gamma \vdash T_i \wedge: T_i'}{\Gamma \vdash D\ T_i \wedge: D\ T_i'} \qquad \wedge:\text{-REFL}\ \frac{}{\Gamma \vdash B \wedge: B}$$

Figure 5. Type consistency.

Theorem 1 (Soundness of round-trip type checking)**.** If $\vdash_{\mathbb{Q}} \Gamma$, $\Gamma \vdash_{\mathbb{Q}} S$, and $\Gamma \vdash_{\mathbb{Q}} t \downarrow S$, then $\Gamma \vdash_{\mathbb{Q}} t :: S'$ and $\Gamma \vdash_{\mathbb{Q}} S' <: S$.

Unlike liquid type inference, the round-trip system requires a proof of termination for all fixpoints; thus if $\Gamma \vdash_{\mathbb{Q}} t :: S$, but t's termination cannot be shown using the chosen definition of termination weakening, the round-trip type system will reject t. Thus we show completeness for a *weakened* round-trip system, obtained from Fig. 4 by replacing $S^{\prec}$ in the premise of the FIX rule by S. We denote the checking judgment of the modified system as $\Gamma \vdash^*_{\mathbb{Q}} t \downarrow S$.

Theorem 2 (Completeness of round-trip type checking)**.** If $\Gamma \vdash_{\mathbb{Q}} t :: S$, then $\Gamma \vdash^*_{\mathbb{Q}} t \downarrow S$.

3.4 Type System Extensions

In this section we further improve the locality of type checking for function applications and branching terms.

Type Consistency. Recall the type checking problem

$$\Gamma \vdash_{\mathbb{Q}} \texttt{append}\ xs\ xs \downarrow \{\texttt{List Pos} \mid \texttt{len}\ \nu = 5\}$$

from Sec. 3.2, and let us change the type of xs to $\{\texttt{List Pos} \mid \texttt{len}\ \nu = 6\}$. In this case, xs has the right element type, `Pos`, but intuitively the partial application $\texttt{append}\ xs$ can still be safely rejected, since no second argument with a non-negative length can fulfill the goal type.

To formalize this intuition we introduce the notion of *type consistency*, defined in Fig. 5. Two scalar types are consistent if they have a common inhabitant for some valid valuation of environment variables. For function types, the relation is not symmetric: a type $x: T_x \to T$ is consistent with a goal type if their return types are consistent for some value of x of type T_x.

We add a premise $\Gamma \vdash T \wedge: T'$ to every rule in Fig. 4 that already has the premise of the form $\Gamma \vdash T <: T'$. The additional premise has no effect on full applications, since for scalar types consistency is subsumed by subtyping. The consistency check can, however, reject a *partial* application e allowed by subtyping, due to goals generated by the rule APPFO, which have a vacuous argument type $\{B \mid \bot\}$. It is easy to show that in the absence of consistency checks, any application of such e would always be rejected by the subtyping check in APPFO; thus introducing consistency checks does not affect completeness of type checking. With consistency checks in place, the term $\texttt{append}\ xs$ in the example above is rejected since the formula $\texttt{len}\ xs = 6 \wedge \texttt{len}\ r \geq 0 \wedge \texttt{len}\ \nu = \texttt{len}\ xs + \texttt{len}\ r \wedge \texttt{len}\ \nu = 5$ is unsatisfiable.

Liquid Abduction. Consider the IF rule in Fig. 4: the type checker can analyze the two branches of the conditional independently of each other, but can only proceed with either branch once the precise type of the guard has been inferred. In the context of synthesis this amount to blindly enumerating type-correct boolean expressions as guards and then checking if any of them enables synthesis of a correct branch. The goal of this section is to improve the locality of the IF rule in order to avoid such blind enumeration.

The idea comes from *condition abduction* [4, 20, 21]: instead of starting with the guard, for which no information can be extracted from the goal type, start by analyzing one of the branches and use logical abduction to infer the weakest assumption under which the branch fulfills the goal type. If such a condition does not exist or is a contradiction, the branch candidate is deemed ill-typed; otherwise the abduced condition can be used as a specification for the guard.

This strategy relies on the availability of a sufficiently fast mechanism to perform logical abduction, which is generally challenging. In SYNQUID, we treat unknown path conditions the same way as unknown refinements in polymorphic instantiations: we restrict their valuations to liquid formulas over environment variables, and use the greatest-fixpoint Horn solver (described in Sec. 3.6) to discover the weakest such valuation. We refer to the modified rule for conditionals as the *liquid abduction* rule:

$$\textsc{IfAb}\ \frac{\Gamma \vdash_{\mathbb{Q}} \psi \quad \mathsf{Sat}(\llbracket\Gamma\rrbracket_\psi \wedge \psi) \quad \Gamma \vdash_{\mathbb{Q}} e \downarrow \{\texttt{Bool} \mid \nu = \psi\} \quad \Gamma;\psi \vdash_{\mathbb{Q}} t_1 \downarrow T \quad \Gamma;\neg\psi \vdash_{\mathbb{Q}} t_2 \downarrow T}{\Gamma \vdash_{\mathbb{Q}} \texttt{if}\ e\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \downarrow T}$$

This rule limits completeness of round-trip type checking by restricting valid guard types to the form above. Most notably, it excludes guards that contain function composition, and thus users have to provide wrapper components to encapsulate complex guard predicates; in all our experiments, the set of required guard components was quite intuitive, thus we conclude that the trade-off between expressiveness and efficiency offered by liquid abduction is reasonable in the context synthesis.

Match Abduction. A similar technique can be used to propose pattern matching, assuming the types of potential scrutinees are restricted to liquid types. In this case, however, the liquid restriction imposes more substantial limitations on the structure of the program: abduction only works if the scrutinee is a variable and its datatype has at least one scalar constructor (such as `Nil` in `List`). Thus, SYNQUID employs a combined approach: it first tries an abduction-based rule, but if that fails, the system reverts to the original MATCH rule of Fig. 4. As a result, type checking (and synthesis) enjoys the efficiency benefits of abduction without compromising completeness.

3.5 The Local Liquid Type Checking Algorithm

Starting from the round-trip typing rules presented above, this section develops *local liquid type checking*: a deterministic algorithm that takes as input a SYNQUID program t, an environment Γ, a goal schema S, and a set of qualifiers $\mathbb{Q}$, and either produces a derivation of $\Gamma \vdash_{\mathbb{Q}} t \downarrow S$ or rejects the program. The main challenge is to find suitable instantiations for polymorphic components, as required by the rule VAR$\forall$; to this end, the algorithm replaces the type variables α_i in the component schema with fresh free type variables α_i'[5], extracts *subtyping constraints* on α_i' from the subtyping premises of the derivation, and then solves the subtyping constraints to either discover a valid type assignment mapping free type variables to liquid types, or conclude that such an assignment does not exist.

For the purpose of constraint solving, we extend the syntax of refinement terms with *predicate unknowns* P_i. Local liquid type checking maintains a set of subtyping constraints $\mathcal{C} = \{\Gamma_i \vdash T_i <: T_i'\}$, a set of Horn constraints $\mathcal{H} = \{\psi_i\}$, a *type assignment* $\mathcal{T} = [\alpha_i' \mapsto T_i]$, and a *liquid assignment* $\mathcal{L} = [P_i \to \{\psi\}_i]$. We denote with $\llbracket\psi\rrbracket_{\mathcal{L}}$ the formula ψ with all predicate unknowns substituted with conjunctions of their valuations in $\mathcal{L}$. The type checking process alternates between the following two steps: it either extends the type derivation by applying one of the rules of Fig. 4, adding any of its subtyping premises to $\mathcal{C}$, or it picks a constraint c from $\mathcal{C}$ and solves it; constraint solving is formalized in the procedure Solve in Fig. 6.

Solve does one of the following, depending on the operands of a subtyping constraint: it either substitutes a type variable for which an assignment already exists (Eq. 1, Eq. 2), unifies a type variable with a type (Eq. 4, Eq. 5), decomposes subtyping over compound types (Eq. 6, Eq. 7), or translates subtyping over scalar types into a Horn constraint and uses the procedure Horn, described in the next section, to find an $\mathcal{L}$ that satisfies all Horn constraints (Eq. 8). Local liquid type checking terminates when the entire type derivation has been built, and all constraints in $\mathcal{C}$ are between free type variables (only Eq. 3 applies).

During unification of α' and T, procedure Fresh inserts fresh predicate unknowns in place of all refinements in T; note that due to the incremental nature of our algorithm, T might itself contain free type variables, which are simply replaced with fresh free type variables to be unified later as more subtyping constraints arise. This novel feature of local liquid type checking, which we call *incremental unification*, is crucial for early error detection. Existing refinement type checkers [13, 33] cannot interleave shape and refinement discovery, since they rely on the global Hindley-Milner inference algorithm to fully reconstruct the shapes of all types in the program before discovering their refinements.

Example. Starting from empty $\mathcal{T}$ and $\mathcal{L}$, Solve($\vdash \alpha' <:$ `List` $\beta' \mid$ `len` $\nu > 0$) instantiates α' by Eq. 4 leading to $\mathcal{T} = [\alpha' \mapsto \{\texttt{List}\ \gamma' \mid P_0\}]$, $\mathcal{L} = [P_0 \mapsto \emptyset]$ and recycles the subtyping constraint; next by Eq. 1 and Eq. 7, the constraint is decomposed into $\vdash \{\texttt{List} \mid P_0\} <: \{\texttt{List} \mid \texttt{len}\ \nu > 0\}$ and $\vdash \gamma' <: \beta'$. The former produces a Horn constraint $P_0 \Rightarrow \texttt{len}\ \nu > 0$, which leads to strengthening $\mathcal{L}[P_0]$, while the latter is retained in $\mathcal{C}$. If further type checking produces a subtyping constraint on β', say `Nat` $<: \beta'$, $\mathcal{T}$ will be extended with an assignment $[\beta' \to \{\texttt{Int} \mid P_1\}]$, which in turn will lead to transforming the constraint on γ' into

[5] We prime the names of free type variables to differentiate them from the bound type variables of the top-level goal schema.

$$
\begin{aligned}
&\mathsf{Solve}(\Gamma \vdash c) = \textbf{match}\ c\ \textbf{with}\\
&\quad \mid \{\alpha' \mid \psi\} <: T, \alpha' \in \mathsf{dom}(\mathcal{T}) \longrightarrow\\
&\qquad \mathcal{C} \leftarrow \mathcal{C} \cup \{\Gamma \vdash \mathsf{Refine}(\mathcal{T}(\alpha'), \psi) <: T\} && (1)\\
&\quad \mid T <: \{\alpha' \mid \psi\}, \alpha' \in \mathsf{dom}(\mathcal{T}) \longrightarrow \text{(symmetrical)} && (2)\\
&\quad \mid \{\alpha' \mid \psi_1\} <: \{\beta' \mid \psi_2\} \longrightarrow\\
&\qquad \mathcal{C} \leftarrow \mathcal{C} \cup \{\alpha' \mid \psi_1\} <: \{\beta' \mid \psi_2\} && (3)\\
&\quad \mid \{\alpha' \mid \psi\} <: T, \alpha' \notin T \longrightarrow\\
&\qquad \mathcal{T} \leftarrow \mathcal{T}[\alpha' \mapsto \mathsf{Fresh}(T)];\\
&\qquad \mathcal{C} := \mathcal{C} \cup \{\Gamma \vdash \{\alpha' \mid \psi\} <: T\} && (4)\\
&\quad \mid T <: \{\alpha' \mid \psi\} \longrightarrow \text{(symmetrical)} && (5)\\
&\quad \mid (x : T_x \to T_1) <: (y : T_y \to T_2) \longrightarrow\\
&\qquad \mathcal{C} \leftarrow \mathcal{C} \cup \{\Gamma \vdash T_y <: T_x, \Gamma; y : T_y \vdash [y/x]T_1 <: T_2\} && (6)\\
&\quad \mid \{D\, T_1^i \mid \psi_1\} <: \{D\, T_2^i \mid \psi_2\} \longrightarrow\\
&\qquad \mathcal{C} \leftarrow \mathcal{C} \cup \{\Gamma \vdash \{D \mid \psi_1\} <: \{D \mid \psi_2\}, \Gamma \vdash T_1^i <: T_2^i\} && (7)\\
&\quad \mid \{B \mid \psi_1\} <: \{B \mid \psi_2\} \longrightarrow\\
&\qquad \mathcal{H} \leftarrow \mathcal{H} \cup \{[\![\Gamma]\!]_{\psi_1 \Rightarrow \psi_2} \wedge \psi_1 \Rightarrow \psi_2\}; && (8)\\
&\qquad \mathcal{L} \leftarrow \mathsf{Horn}(\mathcal{L}, \mathcal{H})\\
&\quad \mid \text{otherwise} \longrightarrow \text{fail} && (9)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{Refine}(\{B \mid \psi\}, \psi') = \{B \mid \psi \wedge \psi'\}\\
&\mathsf{Fresh}(T) = \textbf{match}\ T\ \textbf{with}\\
&\qquad \mid \{\alpha' \mid \psi\} \longrightarrow \beta'\\
&\qquad \mid \{D\, T^i \mid \psi\} \longrightarrow \{D\, \mathsf{Fresh}(T^i) \mid P\}, \quad \mathcal{L} \leftarrow \mathcal{L}[P \mapsto \emptyset]\\
&\qquad \mid \{B \mid \psi\} \longrightarrow \{B \mid P\}, \quad \mathcal{L} \leftarrow \mathcal{L}[P \mapsto \emptyset]\\
&\mathsf{Horn}(\mathcal{L}, \mathcal{H}) = \textbf{if}\ \forall h \in \mathcal{H}.\mathsf{Valid}([\![h]\!]_{\mathcal{L}})\ \textbf{then}\ \mathcal{L}\ \textbf{else}\\
&\qquad\qquad \textbf{let}\ h \leftarrow \{\mathcal{H} \mid \neg \mathsf{Valid}([\![h]\!]_{\mathcal{L}})\}\ \textbf{in}\\
&\qquad\qquad \textbf{let}\ \mathcal{L}' \leftarrow \mathsf{Strengthen}(\mathcal{L}, h)\ \textbf{in}\ \mathsf{Horn}(\mathcal{L}', \mathcal{H})
\end{aligned}
$$

Figure 6. Solving subtyping constraints.

$\vdash \gamma' <: \{\texttt{Int} \mid P_1\}$ and instantiating $[\gamma' \to \{\texttt{Int} \mid P_2\}]$, at which point all free type variables have been eliminated.

3.6 Solving Horn Clauses

The set of Horn constraints $\mathcal{H}$ produced by Solve in Fig. 6 consists of implications of the form $\psi \Rightarrow \psi'$, where each side is a conjunction of a known formula and zero or more predicate unknowns P. The goal of the procedure Horn is to find a liquid assignment to P that validates all constraints in $\mathcal{H}$ or determine that $\mathcal{H}$ is unsatisfiable. The space of possible valuations of each P is $2^{\mathbb{Q}_P}$, where $\mathbb{Q}_P$ is a set of atomic formulas obtained by instantiating qualifiers $\mathbb{Q}$ in the environment where P was created.

Local liquid type checking invokes Horn after every new Horn constraint is issued, and expects to detect an unsatisfiable set of constraints—and thus a type error—as early as possible. Round-trip typing rules—in particular, AppFO and If-Abd—produce constraints in a specific order, such that for each unknown P, implications where P appears negatively (on the left) are issued before the ones where it appears positively (on the right). To enable early error detection in this setting, procedure Horn looks for the *weakest* valuation of each P that validates all Horn constraints issued so far, and deems $\mathcal{H}$ unsatisfiable if for some P such a valuation does not exist *or is inconsistent* (an inconsistent valuation can be safely discarded since it is guaranteed to violate some future constraint where P appears positively).

As an optimization, Horn always starts form the current assignment $\mathcal{L}$ and possibly makes it stronger, since all weaker assignments are known to be too weak to satisfy the previously issued constraints (for a fresh P, $\mathcal{L}[P]$ is initialized with $\emptyset$). Horn uses an iterative greatest-fixpoint computation, outlined in Fig. 6; in every iteration, $\mathsf{Strengthen}(\mathcal{L}, \psi \Rightarrow \psi')$ produces the weakest consistent assignment $\mathcal{L}'$ strictly stronger than $\mathcal{L}$, such that $[\![\psi]\!]_{\mathcal{L}'} \Rightarrow [\![\psi']\!]_{\mathcal{L}}$ is valid (or fails if this is not possible). In general, $\mathcal{L}'$ is not unique; in this case our algorithm simply explores all alternatives independently, which happens rarely enough in the context of refinement type checking and synthesis.

Implementing Strengthen efficiently is challenging: for every unknown P in ψ, the algorithm has to find the smallest subset of atomic predicates from $\mathbb{Q}_P \setminus \mathcal{L}[P]$ that validates the implication. Existing greatest-fixpoint Horn solvers [34] use breadth-first search, which is exponential in the cumulative size of $\mathbb{Q}_P$ and does not scale sufficiently well to practical cases of condition abduction (see Sec. 4). Instead, we observe that this task is similar to the problem of finding minimal unsatisfiable subsets (MUSs) of a a set of formulas; based on this observation, we build a practical algorithm for Strengthen which we dub MUSFix.

The task of Strengthen amounts to finding all MUSs of the set $\bigcup_{\kappa \in \psi}(\mathbb{Q}_\kappa \setminus \mathcal{L}[\kappa]) \cup \{\neg [\![\psi']\!]_{\mathcal{L}}\}$ under the assumption $[\![\psi]\!]_{\mathcal{L}}$. MUSFix borrows the main insight of the Marco algorithm [22] for MUS enumeration, which relies on the ability of the SMT solver to produce unsatisfiable cores from proofs. We modify Marco to only produce MUSs that contain the negated right-hand side of the Horn constraint, $\neg [\![\psi']\!]_{\mathcal{L}}$, since Horn should only produce consistent solutions. For each resulting MUS (stripped of $\neg [\![\psi']\!]_{\mathcal{L}}$), MUSFix finds all possible partitions into valuations of individual predicate unknowns. Since MUSes are normally much smaller than the original set of formulas, a straightforward partitioning algorithm works well and rarely yield more than one valid partition. As an important optimization, when MUS enumeration returns multiple syntactically minimal subsets, MUSFix prunes out *semantically* redundant subsets, *i.e.* it removes a subset m_i if $\bigwedge m_i \Rightarrow \bigwedge m_j$ for some $j \neq i$.

3.7 Synthesis from Refinement Types

From the rules of round-trip type checking we can obtain synthesis rules, following the approach of [27] and reinterpreting the checking and strengthening judgments in such a way that the term t is considered unknown. This interpretation yields a synthesis procedure, which, given a goal schema S, picks a

rule where the goal schema in the conclusion matches S, and constructs the term t from subterms obtained from the rule's premises. More concretely, starting from the top-level goal schema S, the algorithm always starts by applying rule FIX (if $S^{\prec}$ is defined) followed by TABS (if the schema is polymorphic), and finally ABS (if the goal type is a function type). Given a scalar goal, the procedure performs exhaustive enumeration of well-typed E-terms up to a given bound on their depth, solving subtyping constraints at every node and simultaneously abducing a path condition as per the IF-ABD rule. If the resulting condition ψ is trivially true, the algorithm has found a solution; if ψ is inconsistent, the E-term is discarded; otherwise, the algorithm generates a conditional and proceeds to synthesize its remaining branch under the fixed assumption $\neg\psi$, as well a term of type $\{\texttt{Bool} \mid \nu = \psi\}$ to be used as the branch guard. Once all possible E-terms are exhausted, the algorithm attempts to synthesize a pattern match using an arbitrary E-term as a scrutinee, unless the maximal nesting depth of matches has been reached.

Soundness and Completeness. Soundness of synthesis follows straightforwardly from soundness of round-trip type checking, since each program candidate is constructed together with a typing derivation in the round-trip system. Completeness is less obvious: due to condition abduction, the synthesis procedure only explores programs where the left branch of each conditional is an E-term. We can show that every SYNQUID program can be rewritten to have this form (by flattening nested conditionals and pushing conditionals inside matches). Thus the synthesis procedure is complete in the following sense: for each schema S, if there exists a term t, such that the depth of applications and pattern matches in t are within the given bounds, the procedure is guarantees to find some term t' that also type-checks against S; if such a term t does not exist, the procedure will terminate with a failure. Note that the algorithm imposes no a-priori bound on the nesting depth of conditionals (which is crucial for completeness as stated above); this does not preclude termination, since in any given environment, liquid formulas partition the input space into finitely many parts, and every condition abduction is guaranteed to cover a nonempty subset of these parts.

4. Evaluation

We performed an extensive experimental evaluation of SYNQUID with the goal of assessing usability and scalability of the proposed synthesis technique compared to existing alternatives. This goal materializes into the following research questions:

(1) Are refinement types supported by SYNQUID *expressive* enough to specify interesting programs, including benchmarks proposed in prior work?

(2) How *concise* are SYNQUID's input specifications compared both to the synthesized solutions and to inputs required by existing techniques?

(3) Are SYNQUID's inputs *intuitive*: in particular, is the algorithm applicable to specifications not tailored for synthesis?

(4) How *scalable* is SYNQUID: can it handle benchmarks tackled by existing synthesizers? Can it scale to more complex programs than those previously reported in the literature?

(5) How is synthesis performance impacted by various features of SYNQUID and its type system?

4.1 Benchmarks

In order to answer the research questions stated above, we arranged a benchmark suite that consists of 64 synthesis challenges from various sources, representing a range of problem domains. In the interest of direct comparison with existing synthesis tools, our suite includes benchmarks that had been used in the evaluation of those tools [1, 4, 12, 14, 20, 21, 24, 27]. From each of these papers, we picked top three most complex challenges (judging by the reported synthesis times) that were expressible in SYNQUID's refinement logic, plus several easier problems that were common or particularly interesting.

Our second source of benchmarks are verification case studies from the LiquidHaskell tutorial [18]. The purpose of this second category is two-fold: first, these problems are larger and more complex than existing synthesis benchmarks, and thus can show whether SYNQUID goes beyond the state of the art in synthesis; second, the specifications for these problems have been written by independent researchers and for a different purpose, and thus can serve as evidence that input accepted by SYNQUID is sufficiently general and intuitive. Out of the total of 14 case studies, we picked 5 that came with sufficiently strong functional specifications (list sorting, binary-search trees, content-aware lists, unique lists, and AVL trees), erased all implementations, and made relatively straightforward syntactic changes in order to obtain valid SYNQUID input.

Tab. 1 lists the 64 benchmarks together with some metrics of our type-based specifications: the number of synthesis goals including auxiliary functions, the set of components provided, the number of measures used, and the cumulative size of refinements. Note that the reported specification size only includes refinements in the signatures of the synthesis goals; refinements in component functions are excluded since every such function (except trivial arithmetic operations and helper functions) serves as a synthesis goal in another benchmark; refinements in datatype definitions are also excluded, since those definitions are reusable between all benchmarks in the same problem domain. Full specifications are available from the SYNQUID repository [31].

The benchmarks are drawn from a variety of problem domains with the goal of exercising different features in SYNQUID. List and tree benchmarks demonstrate pattern matching, structural recursion, the ability to generate and use polymorphic and higher-order functions (such as `map` and `fold`), as well as reasoning about nontrivial properties of data structures, both universal (e.g. all elements are non-negative) and recursive (e.g. size and set of elements). Our most advanced benchmarks include sorting and operations over data structures with complex representation invariants, such as binary search trees, heaps, and balanced trees. These benchmarks showcase expressiveness

Group	Description	#G	Components	#M	Spec	Code	T-all	T-def	T-nrt	T-ncc	T-nmus
List	is empty	1	true, false	1	6	6	0.02	0.02	0.02	0.02	0.01
	is member	1	true, false, $=$, $\neq$	2	6	18	0.11	0.11	0.13	0.10	-
	duplicate each element	1		1	7	16	0.05	0.05	-	0.08	0.04
	replicate	1	0, inc, dec, $\leq$, $\neq$	1	4	21	0.05	0.05	9.63	0.05	-
	append two lists	1		1	8	15	0.15	0.09	-	0.13	0.10
	concatenate list of lists	1	append	3	5	12	0.05	0.05	0.22	0.04	0.04
	take first n elements	1	0, inc, dec, $\leq$, $\neq$	1	8	27	0.12	0.12	55.82	0.12	-
	drop first n elements	1	0, inc, dec, $\leq$, $\neq$	1	11	20	0.10	0.10	7.87	0.09	-
	delete value	1	$=$, $\neq$	2	8	26	0.10	0.10	0.17	0.12	-
	map	1		1	5	22	0.03	0.03	0.06	0.03	0.02
	zip	1		1	10	22	0.08	0.08	-	0.10	0.07
	zip with function	1		1	10	33	0.07	0.07	-	0.17	0.06
	cartesian product	1	append, map	3	8	26	0.30	0.29	5.83	0.25	0.23
	i-th element	1	0, inc, dec, $\leq$, $\neq$	1	12	20	0.05	0.05	0.38	0.05	-
	index of element	1	0, inc, dec, $=$, $\neq$	2	8	20	0.08	0.08	0.14	0.07	-
	insert at end	1		2	21	19	0.10	0.10	0.24	0.11	0.12
	reverse	1	insert at end	2	15	12	0.09	0.10	0.29	0.12	0.09
	foldr	1		2	14	32	0.10	0.10	-	0.10	0.44
	length using fold	1	0, inc, dec	2	4	17	0.03	0.07	0.03	0.03	0.02
	append using fold	1		2	8	20	0.04	2.19	0.05	0.04	0.03
Unique list	insert	1	$=$, $\neq$	2	8	26	0.27	0.22	0.85	0.20	-
	delete	1	$=$, $\neq$	2	8	22	0.18	0.19	1.07	0.26	-
	remove duplicates	2	is member	2	13	47	0.36	0.87	0.72	0.33	-
	remove adjacent dupl.	1	$=$, $\neq$	3	5	32	1.33	1.32	-	1.31	-
	integer range	1	0, inc, dec, $\leq$, $\neq$	2	13	23	2.36	2.33	22.27	2.33	-
Strictly sorted list	insert	1	$<$	2	8	41	0.18	0.17	0.43	0.16	-
	delete	1	$<$	2	8	29	0.10	0.09	0.21	0.10	-
	intersect	1	$<$	2	8	40	0.33	0.32	0.68	0.34	-
Sorting	insert (sorted)	1	$\leq$, $\neq$	2	8	34	0.25	0.24	0.68	0.23	-
	insertion sort	1	insert (sorted)	4	5	12	0.06	0.06	0.20	0.06	0.05
	sort by folding	1	foldr, $\leq$, $\neq$	2	11	47	2.14	-	-	2.21	-
	extract minimum	1	$\leq$, $\neq$	4	23	40	4.28	4.35	-	7.58	-
	selection sort	1	extract minimum	6	5	16	0.49	0.44	-	0.42	0.38
	balanced split	1		4	31	33	0.96	0.51	-	1.40	0.80
	merge	1	$\leq$, $\neq$	2	17	41	2.19	14.61	-	6.85	-
	merge sort	1	split, merge	6	11	25	2.10	2.10	-	2.52	1.69
	partition	1	$\leq$	4	27	40	2.84	7.89	-	3.42	-
	append with pivot	1		2	28	22	0.22	0.15	0.58	0.22	0.19
	quick sort	1	partition, append w/pivot	6	11	22	2.71	18.45	-	2.49	4.94
Tree	is member	1	false, not, or, $=$	2	6	28	0.29	0.29	7.90	0.28	-
	node count	1	0, 1, +	1	4	18	0.20	0.20	-	0.91	0.14
	preorder	1	append	2	5	18	0.21	0.20	-	0.91	0.15
	create balanced	1	0, inc, dec, $\leq$, $\neq$	2	7	29	0.14	0.15	-	0.21	-
BST	is member	1	true, false, $\leq$, $\neq$	2	6	37	0.09	0.08	0.10	0.08	-
	insert	1	$\leq$, $\neq$	2	8	55	0.91	0.88	-	0.82	-
	delete	1	$\leq$, $\neq$	2	8	68	5.68	5.62	-	10.74	-
	BST sort	5	$\leq$, $\neq$	6	51	115	1.38	1.35	-	1.25	-
Binary Heap	is member	1	false, not, or, $\leq$, $\neq$	2	6	43	0.38	0.38	9.63	0.35	-
	insert	1	$\leq$, $\neq$	2	8	55	0.51	0.50	8.83	0.48	-
	1-element constructor	1	$\leq$, $\neq$	2	5	8	0.02	0.02	0.02	0.02	0.02
	2-element constructor	1	$\leq$, $\neq$	2	6	55	0.08	0.08	0.25	0.07	-
	3-element constructor	1	$\leq$, $\neq$	2	7	246	2.10	2.12	-	1.98	-
AVL	rotate left	3	inc	3	104	91	11.08	12.43	-	17.06	10.08
	rotate right	3	inc	3	107	91	19.23	18.34	-	36.35	17.87
	balance	1	rotate, nodeHeight, isSkewed, isLHeavy, isRHeavy	4	31	119	1.56	-	-	1.76	-
	insert	1	balance, $<$	3	22	47	1.84	1.81	-	1.64	-
	extract minimum	1	$<$	5	11	25	1.92	1.87	-	1.72	-
	delete	2	extract minimum, balance, $<$	5	37	63	15.67	-	-	13.79	-
RBT	balance left	2		9	143	137	5.62	5.53	-	48.47	-
	balance right	2		9	144	137	7.63	7.72	-	45.32	-
	insert	3	balance left, right, $\leq$, $\neq$	9	49	112	8.95	8.53	-	7.93	-
User	desugar AST	1	0, 1, 2	4	5	46	1.17	1.10	-	1.23	0.78
	make address book	1	is private	3	5	35	0.62	3.67	-	0.94	0.55
	merge address books	1	append	3	8	19	0.35	5.85	-	0.31	0.24

Table 1. Benchmarks and SYNQUID results. For each benchmark, we report the number of synthesis goals *#G*; the set of provided *Components*; the number of defined measures *#M*; cumulative size of *Spec*ification and synthesized *Code* (in AST nodes) for all goals; as well as SYNQUID running times (in seconds) with minimal bounds (*T-all*), with default bounds (*T-def*), without round-trip checking (*T-nrt*), without type consistency checking (*T-ncc*), and without MUSFIX (*T-nmus*). "-" denotes timeout of 2 minutes or out of memory.

of refinement types, exercise SYNQUID's ability to perform nontrivial reasoning through refinement discovery, and represent a scalability challenge beyond the current state of the art in synthesis. Finally, we included several benchmarks operating on "custom" datatypes (including the "address book" case study from [20]) in order to demonstrate that SYNQUID's applicability is not limited to standard textbook examples.

4.2 Results

Evaluation results are summarized in Tab. 1. SYNQUID was able to synthesize (and fully verify) solutions for all 64 benchmarks; the table lists sizes of these solutions in AST nodes (*Code*) as well as synthesis times in seconds (*T-all*).

The results demonstrate that SYNQUID is efficient in synthesizing a variety of programs: all but 7 benchmarks are synthesized within 5 seconds; it also scales to programs of nontrivial size, including complex recursive (red-black tree insertion of size 69) and non-recursive functions (3-value binary heap constructor of size 246). Even though specification sizes for some benchmarks are comparable with the size of the synthesized code, for many complex problems the benefits of describing computations as refinement types are significant: for example, the type-based specifications of the three main operations on binary-search trees are over six times more concise than their implementations.

The synthesis times discussed above were obtained for optimal exploration bounds, which could differ across benchmarks. Tab. 1 also reports synthesis times *T-def* for a setting where all benchmarks in the same category share the same exploration bounds. Although this inevitably slows down synthesis, on most of the benchmarks the performance penalties were not drastic: only three benchmarks failed to terminate within the two-minute timeout.

In order to assess the impact on performance of various aspects of our algorithm and implementation, Tab. 1 reports synthesis times using three variants of SYNQUID, where certain features were disabled: the column *T-nrt* corresponds to replacing round-trip type checking with bidirectional type checking (that is, disabling subtyping checks for partial applications); *T-ncc* corresponds to disabling type consistency checks; *T-nmus* corresponds to replacing MUSFIX with naive breadth-first search. The results demonstrate that the most significant contribution comes from using MUSFIX: without this feature 37 out of 64 benchmarks time out, since breadth-first search cannot handle condition abduction even with a moderate number of logical qualifiers. The second most significant feature is round-trip type checking, with 33 benchmarks timing out when disabled, while consistency checks only bring significant speedups for the most complex examples.

4.3 Comparative Evaluation

We compared SYNQUID with state-of-the-art synthesis tools that target recursive functional programs and offer a comparable level of automation. The results are summarized in Tab. 2. For each tool, we list the three most complex benchmarks reported

	Benchmark	*Spec*	*SpecS*	*Time*	*TimeS*
LEON	strict sorted list delete	14	8	15.1	0.10
	strict sorted list insert	14	8	14.1	0.18
	merge sort	9	11	14.3	2.1
JEN	BST find	51	6	64.8	0.09
	bin. heap 1-element	80	5	61.6	0.02
	bin. heap find	76	6	51.9	0.38
MYTH	sorted list insert	12	8	0.12	0.25
	list rm adjacent dupl.	13	5	0.07	1.33
	BST insert	20	8	0.37	0.91
λ^2	list remove duplicates	7	13	231	0.36
	list drop	6	11	316.4	0.1
	tree find	12	6	4.7	0.29
ESC	list rm adjacent dupl.	n/a	5	1	1.33
	tree create balanced	n/a	7	0.24	0.14
	list duplicate each	n/a	7	0.16	0.05
MYTH2	BST insert	n/a	8	1.81	0.91
	sorted list insert	n/a	8	1.02	0.25
	tree count nodes	n/a	4	0.45	0.20

Table 2. Comparison to other synthesizers. For each benchmark we report: *Spec*, specification size (or the number of input-output examples) for respective tool; *SpecS*, specification size for SYNQUID (from Tab. 1); *Time*, reported running time for respective tool; *TimeS*, running time for SYNQUID (from Tab. 1).

in the respective paper that were expressible in SYNQUID's refinement logic; for each of the three benchmarks we report the specification size (if available) and the synthesis time; for ease of comparison, we repeat the same two metrics for SYNQUID (copied over from Tab. 1). Note that the synthesis times are not directly comparable, since the results for other tools are taken from respective papers and were obtained on different hardware; however, the differences of an order of magnitude or more are still significant, since they can hardly be explained by improvements in single-core hardware performance.

We split the tools into two categories according to the specification and verification mechanism they rely on.

Formal Specifications with Deductive Verification. The first category includes LEON [20] and JENNISYS [21]; both tools use pre- and post-conditions (and data structure invariants) to describe computations, and rely on unbounded, SMT-based verification to validate candidate programs (and thus provide the same correctness guarantees as SYNQUID). Unlike LEON and SYNQUID, JENNISYS targets imperative, heap-based programs; the evaluation in [21], however, focuses on side-effect free benchmarks. Both tools use variants of condition abduction, which makes their exploration strategies similar to SYNQUID's.

For both tools, translating their three most complex benchmarks into SYNQUID proved to be straightforward. This suggests that our decidable refinement logic is not too limiting in practice, compared to other formal specification languages used for synthesis. Our specifications are on average slightly more concise than LEON's and significantly more concise than

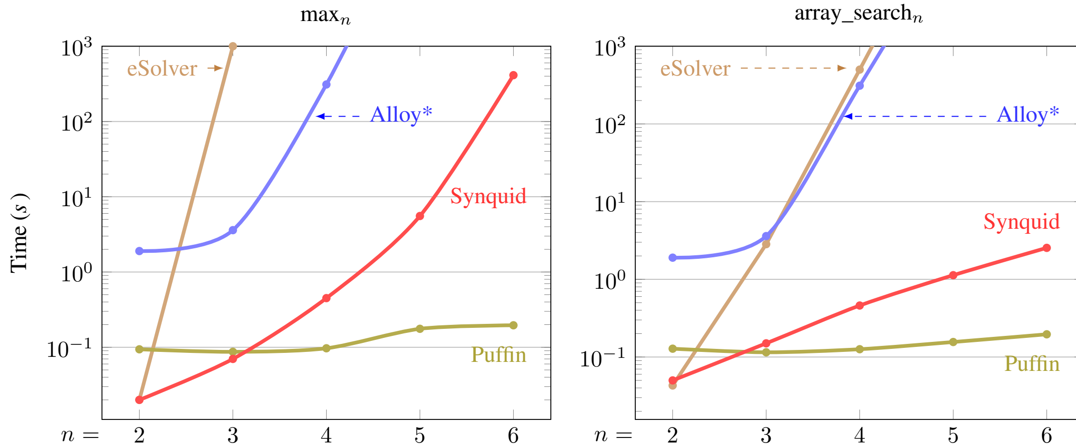

Figure 7. Evaluation on non-recursive benchmarks.

those in JENNISYS; the latter is largely due to the heap-based language, but the results still indicate that embedding predicates into types can help curb the verboseness of traditional Hoare-style specifications.

SYNQUID was able to synthesize solutions to all problems tackled by the other two tools in this category. The converse is not true: automatic verification of some of SYNQUID's benchmarks (such as the binary-search tree example in Sec. 2) requires invariant discovery, which is not supported by the other two tools. This suggests that SYNQUID *qualitatively* differs from other state-of-the-art synthesizers in terms of the class of programs for which a verified solution can be synthesized. On the benchmarks where the other tools are applicable, SYNQUID demonstrates considerably smaller running times, which suggests that fast verification and early pruning enabled by type-based specifications indeed improve the scalability of synthesis.

Input/Output Examples. Our second category of tools includes MYTH [27], λ^2 [12] and ESCHER [1], which synthesize programs from concrete input-output examples, as well as MYTH2 [14], which uses generalized input-output examples. Using refinement types, we were able to express 3 out of 3, 10, 5, and 7 of their most complex benchmarks, receptively. The functions we failed to specify either manipulate nested structures in a representation-specific way (such as "insert a tree under each leaf of another tree"), or perform filtering ("list of nodes in a tree that match a predicate").

At the same time, we found cases where refinement types are concise and intuitive, while providing input-output examples is extremely tedious. One of those cases is insertion into a binary search tree: MYTH requires 20 examples, each of which contains two bulky tree instances and has to define the precise position where the new element is to be inserted; the type-based specification for this problem, given in Sec. 2, is straightforward and only defines the abstract effect of the operation relevant to the user. This suggests that in general, logic-based specification techniques, including refinement types in SYNQUID, are a better fit for describing operations that maintain a complex representation invariant but have a simple abstract effect, while example-based approaches fare better when describing operations that inherently expose the complex representation of a data structure.

Experiments with example-based tools only report the number of examples required for synthesis and not their sizes; however, we can safely assume that each example contains multiple AST nodes, and thus conclude that type-based specifications for the benchmarks in Tab. 2 are more concise. By imposing more constraints on the set of examples (such as *trace completeness* [27]) and increasing its size, example-based synthesizers can trade off user effort for synthesis time. On the benchmarks under comparison, MYTH appears to favor performance, while λ^2 prefers smaller example sets. SYNQUID tries to offer the best of both world and achieves good performance with concise specifications.

4.4 Evaluation on Non-recursive Benchmarks

In order to asses the scalability of MUSFIX on larger search spaces, we evaluated SYNQUID on two parametrized benchmarks from the linear integer arithmetic track of the SyGuS'14 competition [3]: $\texttt{max}_n$ (find maximum of n integer arguments) and $\texttt{array_search}_n$ (find the position of a value in a sorted array with n elements). Both benchmarks target non-recursive programs that consist of a series of nested conditionals; moreover, the search space for the branch guards grows exponentially with n. This makes the two problems ideal benchmarks for condition abduction techniques.

Fig. 7 shows SYNQUID synthesis times on the two benchmarks for $n=2,3,\ldots,6$. For reference, we also plot the results for the enumerative solver (the fastest of the SyGuS baseline solvers), as well as the higher-order solver ALLOY* [24], and PUFFIN [4], a specialized synthesizer for conditional integer-arithmetic expressions[6]. The results show that SYNQUID's condition abduction scales relatively well compared to general synthesizers, but loses to PUFFIN's theory-specific abduction engine.

[6] The results for these tools are taken from their respective papers; only differences in the order of magnitude are significant.

5. Related Work

Our work is the first to leverage general decidable refinement types for synthesis, but it builds on a number of ideas from prior work as has been highlighted already throughout the paper. Specifically, our work combines ideas from two areas: synthesis of recursive functional programs and refinement type inference.

Synthesis of Recursive Functional Programs. A number of recent systems target recursive functional programs and use type information in some form to restrict the search space. The most closely related to our work are MYTH [27], MYTH2 [14], and LEON [20].

MYTH pioneered the idea of leveraging bidirectional type checking for synthesis. However, MYTH does not support polymorphism or refinement types. Instead, the system relies on examples in order to specify the desired functionality. For certain functions, providing examples is easy whereas writing a refinement type is cumbersome or, due to the limitations of decidable refinement logic, even impossible. That said, examples in general do not fully specify a program; thus programming by example always involves a manual verification step. Moreover, for some less intuitive problems, such as insertion into a balanced tree or AST transformations, providing input-output examples requires familiarity with all details and corner cases of the algorithm, whereas refinement types enable a more abstract specification. Additionally, MYTH expects the set of examples to be *trace complete*, which means that for any example the user provides, there should also be examples corresponding to any recursive calls made on that input. Other systems that use a combination of types and input-output examples, and thus have similar advantages and disadvantages relative to our system, include λ^2 [12] and ESCHER [1].

MYTH2 generalizes example-based synthesis: it treats examples as singleton types, and extends the input language with intersection and union types, as well as parametric polymorphism. This addresses some of the shortcomings of concrete input-output examples (in particular, their verboseness), however, in the absence of dependent function types most interesting programs still cannot be fully speficied. Combining SYNQUID's dependent types with singletons, intersection, and unions found in MYTH2 is an interesting direction for future work.

In LEON, synthesis problems are defined by first-order specifications with recursive predicates, and verification is based on a semi-decision procedure [35], implemented on top of an SMT solver. LEON's verification engine does not support invariant inference, which prevents it from generating provably correct implementations for problems such as insertion into a sorted list or a binary search tree. The general synthesis strategy is similar to ours: first decompose the specification and then switch to generate-and-check mode, enhanced with condition abduction. Unlike our system, LEON does not perform systematic specification decomposition in the generate-and-check mode, and lacks support for polymorphism and high-order functions.

The use of type information has also proved extremely useful for code completion [15, 23, 28], although none of these systems rely on a type system as expressive as ours, and they are designed for a very different set of tradeoffs compared to SYNQUID. For example, because the problem is highly under-constrained, these systems place significant emphasis on the ranking of solutions.

Another important body of related work related is *hole driven development*, as embodied in systems like Agda [26] and Idris [6], which leverage a rich type system to aid development, but are meant to be used interactively rather than to perform complete synthesis. Djinn [5] serves a similar purpose but uses the less expressive Haskell type system.

Refinement Type Checking. Our type checking algorithm is based on liquid type inference [19, 33, 36–38], which pioneered combining Hindley-Miler unification with predicate abstraction. We integrate their ideas with bidirectional type checking [30], which has been used before both for other flavors of refinement types [9, 11, 41] and for unrestricted dependent types [8], but not for general decidable refinement types. Another difference with liquid types is that we use greatest-fixpoint predicate abstraction procedure inspired by [34], and improved using an algorithm for efficient MUS enumeration [22].

Logical Abduction. The concept of abduction in logical reasoning has found numerous applications in programming languages, including specification inference [2, 10] and program synthesis [4, 20, 21]. Abduction techniques based on quantifier elimination [2, 10] and theory-specific unification operators [4], are precise and efficient, but only applicable to restricted domains. SYNQUID performs abduction using predicate abstraction and MUS enumeration, which can be applied to a wider range of specification logics, but its precision is limited to the given set of logical qualifiers.

Acknowledgements

We thank Aleksandar Milicevic, Shachar Itzhaky, Ranjit Jhala, and the anonymous reviewers for their valuable input. This work was funded by DARPA Grant *FA8750-14-2-0242* (MUSE) and NSF Grants *CCF-1139056* and *CCF-1438969*.

References

[1] A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In *CAV*, 2013.

[2] A. Albarghouthi, I. Dillig, and A. Gurfinkel. Maximal specification synthesis. In *POPL*, 2016.

[3] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, 2013.

[4] R. Alur, P. Černý, and A. Radhakrishna. Synthesis through unification. In *CAV*, 2015.

[5] L. Augustsson. djinn, the official Haskell package webpage. http://hackage.haskell.org/package/djinn, 2014.

[6] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013.

[7] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Hipspec: Automating inductive proofs of program properties. In *ATx/WInG*, 2012.

[8] T. Coquand. An algorithm for type-checking dependent types. *Sci. Comput. Program.*, 26(1-3):167–177, 1996.

[9] R. Davies and F. Pfenning. Intersection types and computational effects. In *ICFP*, 2000.

[10] I. Dillig, T. Dillig, B. Li, and K. L. McMillan. Inductive invariant generation via abductive inference. In *OOPSLA*, 2013.

[11] J. Dunfield and F. Pfenning. Tridirectional typechecking. In *POPL*, 2004.

[12] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, 2015.

[13] C. Flanagan. Hybrid type checking. In *POPL*, 2006.

[14] J. Frankle, P. Osera, D. Walker, and S. Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *POPL*, 2016.

[15] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *PLDI*, 2013.

[16] J. Heras, E. Komendantskaya, M. Johansson, and E. Maclean. Proof-pattern recognition and lemma discovery in ACL2. In *LPAR*, 2013.

[17] J. P. Inala, X. Qiu, B. Lerner, and A. Solar-Lezama. Type assisted synthesis of recursive transformers on algebraic data types. *CoRR*, abs/1507.05527, 2015.

[18] R. Jhala, E. Seidel, and N. Vazou. Programming with refinement types (an introduction to liquidhaskell). `https://ucsd-progsys.github.io/liquidhaskell-tutorial`, 2015.

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

[20] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *OOPSLA*, 2013.

[21] K. R. M. Leino and A. Milicevic. Program extrapolation with Jennisys. In *OOPSLA*, 2012.

[22] M. Liffiton, A. Previti, A. Malik, and J. Marques-Silva. Fast, flexible mus enumeration. *Constraints*, pages 1–28, 2015.

[23] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: Helping to navigate the api jungle. In *PLDI*, 2005.

[24] A. Milicevic, J. P. Near, E. Kang, and D. Jackson. Alloy*: A general-purpose higher-order relational constraint solver. In *ICSE*, 2015.

[25] O. Montano-Rivas, R. L. McCasland, L. Dixon, and A. Bundy. Scheme-based theorem discovery and concept invention. *Expert Syst. Appl.*, 39(2):1637–1646, 2012.

[26] U. Norell. Dependently typed programming in agda. In *AFP*, 2009.

[27] P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015.

[28] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *PLDI*, 2012.

[29] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[30] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[31] N. Polikarpova and I. Kuraj. Synquid code repository. `https://bitbucket.org/nadiapolikarpova/synquid/`, 2015.

[32] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. *CoRR*, abs/1510.08419, 2016.

[33] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[34] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, 2009.

[35] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS*, 2011.

[36] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *ESOP*, 2013.

[37] N. Vazou, E. L. Seidel, and R. Jhala. Liquidhaskell: experience with refinement types in the real world. In *Haskell*, 2014.

[38] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. P. Jones. Refinement types for haskell. In *ICFP*, 2014.

[39] N. Vazou, A. Bakst, and R. Jhala. Bounded refinement types. In *ICFP*, 2015.

[40] P. Wadler. Theorems for free! In *FPCA*, 1989.

[41] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, 1999.