Program Synthesis from Polymorphic Refinement Types

Abstract

We present a method for synthesizing recursive functions that provably satisfy a given specification in the form of a refinement type. We observe that such specifications are particularly suitable for program synthesis for two reasons. First, they support automatic inference of rich universal invariants, which enables synthesis of nontrivial programs with no additional hints from the user. Second, refinement types can be decomposed more effectively than other kinds of specifications, which is the key to pruning the search space of candidate programs. To support such decomposition, we propose a new algorithm for refinement type inference, which is applicable to partial programs.

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, binary-search tree manipulations, red-black tree rotation), as well as most of the benchmarks tackled by existing synthesizers, often starting from a more concise and intuitive user input.

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

1. Introduction

The key to scalable program synthesis is modular verification. Modularity enables the synthesizer to prune inviable candidate 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 [1, 6, 8, 9, 16]: not only do well-typed programs vastly outnumber ill-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; existing approaches supplement type information with other kinds of specifications, such as input-output examples [1, 6, 16], pre/post-conditions [11, 13], or executable assertions [9]. 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 system called SYNQUID that pushes the idea of type-directed synthesis one step further by taking advantage of *refinement types* [7, 19]: types decorated with predicates from logics efficiently decidable by SMT solvers. For example, type `Nat` can be defined as a refinement over the simple type `Int`, $\{\nu : \texttt{Int} \mid \nu \geq 0\}$, where the predicate $\nu \geq 0$ restricts the type to those values that are greater than or equal to zero[1]. Base refinement types, such as `Nat`, can be combined into *dependent function types*, written $x : T_1 \to T_2$, where the formal argument x may appear in the refinement predicates of T_2.

Verification techniques based on refinement types—in particular, the *liquid types* framework [10, 19, 23, 24]—have been successful at checking nontrivial properties of programs with little to no user input. Piggybacking quantifier-free predicates on top of types makes it possible to rely on the type system for automatically generalizing and instantiating rich universal invariants, while leaving the SMT solver to deal with subsumption queries over simple predicates. For example, the type `List Nat` encodes a universal invariant that all elements of a list are natural numbers; when a list of this type is constructed or, conversely, scrutinized, the type system automatically decomposes such an invariant into properties over individual list elements, simple enough to be expressed with quantifier-free predicates, and it does so using no other input than the type of the `Cons` constructor. The key insight behind SYNQUID is that program synthesis can harness the unique ability of refinement type systems to decompose complex specifications into simpler properties over subexpressions in order to prune the space of candidate programs more effectively than what can be achieved by input-output examples or even pre/post-conditions.

In SYNQUID the user specifies a synthesis goal by providing a type signature. For example, the function `replicate` can be

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

specified as follows:

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

Given an integer n and a value x of type α, `replicate` produces a list of length n where every value is of type α. The specification could be further strengthened with the constraint that each list element must be equal to x by changing the type parameter of the list to $\{\alpha \mid \nu = x\}$. Somewhat surprisingly, this would be redundant: since the type parameter α can be instantiated with *any refinement type*, the specification above guarantees that whatever property x might have (including the property of having a particular value), every element of the list will share that same property. Thus, the type as written above fully specifies the behavior of the function and is only marginally more complex than a conventional ML or Haskell type. We argue therefore that (within the domain of their applicability) refinement types offer a convenient interface to a program synthesizer; in particular, they are as straightforward and often more concise than input/output examples, especially considering that state-of-the-art example-based systems often require about a dozen examples to get the correct implementation [16].

In addition to the type signature, our algorithm takes as input an environment of functions and inductive datatypes that the synthesized program can use as components. Each of these components is described purely through its type signature: implementations of component functions need not be available.

As demonstrated by the specification of `replicate`, *parametric polymorphism* is crucial to the expressiveness of refinement types, providing means to abstract (quantify) over refinements. The liquid types inference algorithm [19] makes it possible to instantiate polymorphic types automatically, using a combination of Hidley-Milner-style unification with predicate abstraction, which is a key ingredient in automating the verification. Predicate abstraction constructs values of unknown refinements as conjunctions of atomic predicates called *qualifiers*, in general, provided by the user (in practice, however, qualifiers can almost always be extracted automatically from the types of components and the specification).

Unfortunately, liquid type inference in its existing form is impractical in the context of synthesis, since it does not fully exploit the modularity of refinement type checking: first, it only propagates type information bottom-up and, second, it requires a Hidley-Milner unification pass over the whole program before inferring refinements.

The needs of synthesis prompt us to develop a new type checking mechanism, which would propagate specifications top-down as much as possible and be applicable to incomplete programs. The new mechanism, which we dub *modular refinement type reconstruction*, is enabled by a combination of an existing *bidirectional* approach to type checking (which, however, has not been previously applied in the context of general decidable refinement types) and a novel, *incremental* algorithm for liquid type inference, which gradually discovers the shape and the refinements of unknown types as it analyzes different parts of the program. By making type checking as local as possible, the new mechanism facilitates scalable synthesis through interleaving generation and verification of candidate programs at a very fine level of granularity.

```
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. Code synthesized from the type signature of `replicate`

To make full use of top-down propagation of type information, the predicate abstraction engine of SYNQUID has to discover *weakest* refinements instead of strongest, as in liquid types, which is fundamentally more expensive. A secondary contribution of this paper is a practical technique for doing so, which relies on an existing mechanism for enumerating minimal unsatisfiable subsets [14].

The ability to discover weakest refinements naturally extends to inferring environment assumptions that are necessary to make a given solution correct. This search strategy, most commonly known as *condition abduction*, has been proven effective in prior work [2, 11, 13]; in SYNQUID it comes at virtually no cost since it is simply a byproduct of typechecking.

The combination of explicit candidate enumeration, modular checking, and condition abduction enables our system to produce an implementation of `replicate` shown in Fig. 1 in under a second. Sec. 2 shows how these techniques extend naturally to synthesizing programs that manipulate data structures with complex invariants (such as sorted and unique lists, binary search trees, heaps, and red-black trees) and use higher-order functions (such as maps and folds).

In total, we have evaluated SYNQUID on 50 different synthesis problems from a variety of sources. Our evaluation indicates that SYNQUID can synthesize programs that are more complex than those previously reported in the literature, including four different sorting algorithms, binary search tree manipulations, and red-black tree rotation. We also show that refinement types are expressive enough to specify a broad range of problems. We compare SYNQUID with its competitors, based on input-output examples, expressive specifications, and test harnesses, and demonstrate that we can handle the majority of their most challenging benchmarks. Compared to the state-of-the art tools based on input-output examples, our specifications are usually more concise and we generate provably correct code. Compared to the tools based on deductive reasoning, we can handle more complex reasoning due to refinement inference. More broadly, our system demonstrates a new milestone in the use of expressive type systems to support program synthesis, showing that expressive types do not have to make a programmer's life harder, but can in fact help automate some aspects of programming.

2. Overview

Consider a core ML-like language featuring conditionals, algebraic datatypes, **match**, ML-style polymorphism, and **fix**points. We equip our language with *general decidable* refinement types, closely following the liquid types framework [10, 19, 23]. Values of a base type B are described as $\{B \mid \psi\}$, where ψ is a *refinement predicate* over the program variables and a special *value variable* ν, which does not appear in the program. Functions are described using dependent types of the form $x\colon T_1 \to T_2$, 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 subsumption in this logic can be decided; examples in this paper use QFAUFLIA (the quantifier-free logic of arrays, uninterpreted functions, and linear integer arithmetic), as does our prototype implementation.

Synthesis procedure. The synthesis problem is given by (a) a goal refinement type (b) an environment, which initially contains a set of type signatures for component datatypes and functions, and (c) a set of qualifiers.

Depending on the goal type, our system performs one of the following steps: it either decomposes the synthesis problem into subproblems (with a simpler goal and/or enriched environment), or it switches to the guess-and-check mode, in which it enumerates all well-typed terms that have the desired *type shape*—i.e. the type stripped of its refinements—and then issues a *subtyping constraint* to make sure sure that the guess fulfills the goal. As in several previous approaches that rely on explicit exploration (e.g. [8, 16]), the terms are enumerated in order or increasing size and syntactically restricted to β-normal η-long form.

Importantly, even in the guess-and-check mode, our system does not abandon refinements completely; rather it uses the type system to automatically decompose the goal specification into properties that can be further propagated top-down and those that can only to be checked after the whole term is constructed. In addition, since guessing only enumerates terms that are well-typed in our refinement type system, the preconditions of component functions serve as filters for candidate arguments that can be applied right away. The *early filtering* enabled by these two features is key to the scalability of our synthesis procedure, which is confirmed by our evaluation (Sec. 4).

Solving subtyping constraints. Due to polymorphic components, in a subtyping constraint $\Gamma \vdash T' <: T$, the types T and T' may contain free type variables α_i; the interpretation of such a constraint is two-fold: (i) the shapes of T and T' must have a unifier in the traditional Hindley-Milner sense [17] and (ii) once the shapes are unified, the refinements of T' must subsume the ones of T. The subsumption constraints, in turn, may contain *predicate unknowns* κ_i, which stand for unknown refinements of the types introduced during unification or unknown branch conditions in the environment Γ. For example, a subtyping constraint $\alpha <: \texttt{Nat}$ gives rise to a unifier $[\alpha \to \{\texttt{Int} \mid \kappa_0\}]$, where κ_0 is a fresh predicate unknown, and to a subsumption constraint $\kappa_0 \Rightarrow \nu \geq 0$. Our system uses a greatest-fixed-point predicate abstraction procedure to assign to each κ_i the smallest conjunction of qualifiers that validates all the subsumption constraints; if no such assignment exists, it rejects the candidate program.

Issuing subtyping constraints early only helps if they can be solved early too. Polymorphism make this challenging: previous approaches to refinement type reconstruction [12, 19] rely on a whole-program shape inference phase to completely determine the shapes of all the types before they can insert predicate unknowns and proceed with refinement inference. Instead, our system solves subtyping constraints incrementally, interleaving the two phases. For example, a constraint $\alpha <: \{\texttt{List}\ \beta \mid \psi\}$, where both α and β are free type variables, will result in partial shape reconstruction, leading to a unifier $[\alpha \to \{\texttt{List}\ \gamma \mid \kappa_0\}]$, a subsumption constraint $\kappa_0 \Rightarrow \psi$, and another subtyping constraint $\gamma <: \beta$, to be solved later, as the shape of β is recovered.

The rest of the section goes though a series of examples that demonstrate various features of our type system and illustrate the synthesis procedure outlined above.

Example 1: Recursion and conditionals. 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 generic list datatype whose constructors are refined with length information, expressed by means of an uninterpreted function `len`, also called a *measure*:

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

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

The **decreases** annotation above enables recursion on lists by defining a *termination metric*, which maps lists to a type that has a predefined well-founded order in our language and thus can be used in termination checks. Examples in this paper integrate measure-based refinements (and termination metrics) with datatype definitions for simplicity; [10] shows how a measure definition can be syntactically decoupled from its datatype and then desugared automatically into the above representation, thus, allowing a single datatype to be decorated with multiple, custom measures depending on the problem at hand.

For the rest of the section, let us fix the set of qualifiers $\mathbb{Q}$ to $\{\star \leq \star, \star \geq \star, \star \neq \star\}$, 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 first performs a decomposition step: it adds $n\colon \texttt{Nat}; x\colon \alpha$ to the environment and proceeds to synthesize a body of type $\{\texttt{List}\ \alpha \mid \texttt{len}\ \nu = n\}$. This decomposition step does not restrict the search, 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. SYNQUID also makes the function recursive in the first argument (but not the

second), since its type shape `Int` has an associated well-founded order; to this end, the tool extends the environment with a binding `replicate` : $(m : \{\texttt{Int} \mid 0 \leq \nu < n\} \to y : \alpha \to \{\texttt{List}\ \alpha \mid \texttt{len}\ \nu = m\})$ (note the weakened type, which only allows calling `replicate` with arguments strictly smaller than n).

When synthesizing the body of `replicate`, SYNQUID first adds a fresh predicate unknown κ_C to the set of environment assumptions, which stands for the guard of the current branch. It then starts enumerating candidate application terms t of shape $\{\texttt{List}\ \alpha\}$ from simplest to more complex, and for each t it issues a subtyping constraint to check if the candidate has the desired refined type. In our example, the simplest list term is `Nil`; this choice results in a subtyping constraint $n : \texttt{Nat}; x : \alpha; \kappa_C \vdash \{\texttt{List}\ \alpha \mid \texttt{len}\ \nu = 0\} <: \{\texttt{List}\ \alpha \mid \texttt{len}\ \nu = n\}$ which reduces to the subsumption constraint: $0 \leq n \wedge \kappa_C \wedge \texttt{len}\ \nu = 0 \Rightarrow \texttt{len}\ \nu = n$ At this point, SYNQUID uses predicate abstraction to find the weakest valid *liquid assignment* to all predicate unknowns, that is, the smallest conjunction of qualifiers from $\mathbb{Q}$, instantiated with all suitable variables in scope, that makes the subsumption constraint hold. If such a weakest assignment is a contradiction, the candidate is discarded. In the example, predicate abstraction discovers the assignment $\mathcal{L} = [\kappa_C \mapsto n \leq 0]$, effectively abducing the necessary branching condition. Since the condition is not trivially true, the system proceeds to synthesize the **`else`**-branch of the conditional under the fixed assumption $\neg\mathcal{L}[\kappa_C]$. The technique for synthesizing branching programs based on condition abduction it has been successfully employed in several program synthesis tools [2, 11, 13], but to our knowledge SYNQUID is the first one to use predicate abstraction to systematically discover optimal branch conditions.

The remaining branch has to deal with the harder case of $n > 0$. When enumerating application terms for this branch, SYNQUID eventually decides to apply the `replicate` components (that is, make a recursive call). A 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 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:

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

This definition states that one can obtain a binary search tree either by taking an empty tree, or by composing a node with key x with two binary search trees, 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.

The following type specifies insertion into a binary search tree:

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

From this specification, SYNQUID generates the following implementation (where some symbols are numbered for future reference):

```
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 Node¹ y l (insert¹ x r)
        else Node² y (insert² x l) r
```

The challenging aspect of this example is reasoning about sortedness. For example, in order for $\texttt{Node}^1\ y\ l\ (\texttt{insert}^1\ x\ r)$ to be type-correct, the recursive call must have the type $\texttt{BST}\ \{\alpha \mid y < \nu\}$; this type is not implied by the user-provided signature for `insert`, and in fact only makes sense for this particular call site, since it mentions a local variable y. Inferring this type amounts to discovering a complex inductive property of `insert`, namely that inserting a key that is greater than some value z into a tree with keys greater than z again produces a tree with keys greater than z. The requirement to infer such complex invariants puts this and similar examples beyond reach of existing systems that synthesize provably correct code, such as LEON.

In our framework, this property is easily inferred using the combination of polymorphic recursion and automatic instantiation of type variables by means of predicate abstraction. When `insert` is added to the environment, its type is generalized into $\forall\beta.x : \beta \to t : \texttt{BST}\ \beta \to \{\texttt{BST}\ \beta \mid \texttt{keys}\ \nu = \texttt{keys}\ t + \{x\}\}$. At the site of the recursive call, β is instantiated with $\{\alpha \mid \kappa_0\}$; here the shape of β is determined by unification, but the refinement is unknown. Enforcing the precondition of $\texttt{Node}^1\ y\ l$ leads predicate abstraction to discover the liquid assignment $[\kappa_0 \mapsto y < \nu, \kappa_C \mapsto y <= x]$ (where κ_C is the current branch guard), whereby simultaneously instantiating the polymorphic recursive call and abducing the branch condition.

Example 3: Abstract refinements. Using refinement types 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 have shown that refinement types are the right tool 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 [23], enable explicit quantification over refinements of datatypes and function types. For

example, the `List` datatype can be parameterized by a binary predicate P, which describes the relation between every in-order pair of elements in the list:

```
data List a (P::a → a → Bool) where
  Nil::List a P
  Cons::x:a → xs:List {a | P x ν} P → List a P
```

On the one hand, as shown in [23], this enables concise definitions of list with various inductive properties; for example, increasing and unique lists can now be defined as an instantiations `IncList` $\alpha = $ `List` $\alpha(\leq)$ and `UniList` $\alpha = $ `List` $\alpha(\neq)$, respectively.

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

```
reverse::(P::a → a → Bool) . xs:List a P →
  {List a (λx λy . P y x) | len ν = len xs}
```

It says that whatever relation holds between every *in-order* elements of the input list, also has to hold between every *out-of-order* elements of the output list. This type does not restrict the applicability of `reverse`, since at the call site P can always be instantiated with `True`; the implementation of `reverse`, however, has to be correct to any value of P, 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.

[23] has also shown that abstract refinements make it possible to give a precise type to `foldr`, which folds a binary function over a list. Using the type they proposed, SYNQUID is able to synthesize both the standard implementation of `foldr`, as well as non-recursive implementations of standard list functions, such as `length` or `append` that use `foldr`. Synthesizing the higher-order argument to `foldr` presents no problem to SYNQUID, since it can be treated as a separate synthesis goal.

3. The SYNQUID Language

The goal of this section is to develop a high-level formalization of the procedure outlined in Sec. 2 in the form of *synthesis rules*. In doing so we follow previous work on type-directed synthesis [9, 16], which has shown how to turn type checking rules for a language into synthesis rules for the same language, by reinterpreting each rule that generates a type given a term as generating a term given a type.

We first present the syntax and static semantics of our core language with refinement types (Sec. 3.1); the language is based on NANOML developed within the liquid types framework [10, 19], thus our presentation skips common details, focusing on the differences. In the interest of space we omit abstract refinements (see Sec. 2) from the formalization; [23] has shown that integrating this mechanism into the type system that already supports parametric polymorphism is straightforward. Sec. 3.2 describes the core technical contribution of our paper: the *modular refinement type reconstruction* algorithm, which combines the ideas of bidirectional type checking [18] with an incremental procedure for solving subtyping constraints. Finally, Sec. 3.3 presents the synthesis rules derived from the modular type reconstruction rules.

ψ	$::=$ *(varies)*	*Refinement term:*
Δ	$::=$	*Sort:*
	$\mid \mathbb{B} \mid \mathbb{Z} \mid \ldots$ *(varies)*	interpreted
	$\mid \delta$	uninterpreted
t	$::=$	*Program term:*
	$\mid e$	E-term
	$\mid \lambda x.t$	abstraction
	$\mid$ **if** e **then** t **else** t	conditional
	$\mid$ **match** e **with** $\mid_i \mathtt{C}_i\langle x_i^j\rangle \mapsto t_i$	match
	$\mid$ **fix** $x.t$	fixpoint
e	$::=$	*E-term:*
	$\mid x$	variable
	$\mid e\ t$	application
B	$::=$	*Base type:*
	$\mid$ `Bool` $\mid$ `Int`	primitive
	$\mid D^m$	datatype
	$\mid \alpha$	type variable
$\mathbb{T}^{\square}$	$::=$	*Type Skeleton*
	$\mid \{B\ \mathbb{T}_i^{\square} \mid \square\}$	base
	$\mid x: \mathbb{T}^{\square} \to \mathbb{T}^{\square}$	function
$\mathbb{S}^{\square}$	$::= \forall\alpha_i.\mathbb{T}^{\square}$	*Schema Skeleton*
τ,σ	$::= \mathbb{T}^{()}, \mathbb{S}^{()}$	*Unref. Type, Schema*
T,S	$::= \mathbb{T}^{\psi}, \mathbb{S}^{\psi}$	*Ref. Type, Schema*
$\hat{T}$	$::= T \mid$ `let` $x:T$ `in` $\hat{T}$	*Contextual Ref. Type*
$\hat{S}$	$::= \forall\alpha_i.\hat{T}$	*Contextual Ref. Schema*

Figure 2. Syntax

3.1 Syntax and types

Fig. 2 shows the syntax of SYNQUID.

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 [5, 16] into *elimination* (E-term) and *introduction* (I-term) forms. Intuitively, E-terms—variables and applications—propagate information bottom-up, *composing* a complex property from properties of their components; I-terms propagate information top-down, *decomposing* a complex requirement into simpler requirements for their components.

Types and Schemas. Type and schema skeletons are parametrized by the space of refinements. Instantiating this space with a unit yields unrefined types and schemas; instantiating it with formulas ψ yields refinement types and schemas. The shape of a type T, obtained by erasing all refinements, is denoted $\mathtt{shape}(T)$.

A user-defined datatype is modeled as a base type D^m of arity $m \geq 0$ (other base types have an implicit arity of 0). Datatype constructors are represented simply as functions that must have the type $\forall\alpha_1 ... \alpha_m . T_1 \to ... \to T_k \to D\ \alpha_1 ... \alpha_m$.

In a dependent function type $x\colon T_1 \to T_2$, T_2 may reference the formal argument x, which goes out of scope once the function is applied. The usual way of eliminating this variable is to give an application $t_1 t_2$ the type $[t_2/x]T_2$, which requires equipping the language with `let`-expressions and A-normalizing the program, such that every t_2 is a variable. This approach is not suitable for SYNQUID, since it requires synthesizing arguments to unknown functions. To address this problem we introduce *contextual types*: the purpose of a contextual type $\texttt{let}\ x\colon T_1\ \texttt{in}\ T_2$, is to propagate the information about the formal argument x, which later can be used when deciding subtyping.

A *type environment* Γ is a sequence of path conditions ψ and variable bindings $x\colon T$.

Judgments. Our type system has four kinds of judgments: well-formedness, liquidness, subtyping, and typing. A formula ψ is *well-formed* in the environment Γ, written $\Gamma \vdash \psi$, if it is of a Boolean sort and all its free variables are bound in Γ. 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 by variables. Both of those judgment are extended to types in a standard way. The *subtyping* judgment $\Gamma \vdash T_1 <: T_2$ is also standard.

The *typing* judgment $\Gamma \vdash_{\mathbb{Q}} t :: \hat{S}$ states that the term t has (contextual) type schema S in the environment Γ using qualifiers $\mathbb{Q}$. Fig. 3 presents type checking rules for SYNQUID, which are based on liquid type checking for NANOML.

The distinction between the terms with constructible types and those with liquid types is present in the original liquid type system; in SYNQUID it corresponds to the distinction between E-terms and I-terms: E-terms have constructible (possibly contextual) types, while I-terms have liquid (non-contextual) types. Note that the rule VAR$\forall$, which handles polymorphic instantiations, is special: it is the only E-term rule that requires creating a fresh liquid type.

SYNQUID has two separate rules for first-order (APPB) and higher-order (APP) application, since only in the first case the formal argument of the function may appear in its result type, and thus must be bound using a contextual type.

The final distinction between SYNQUID and NANOML lies in the rule for fixpoints, which in our case 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".

Type Inference $\boxed{\Gamma \vdash_{\mathbb{Q}} t :: \hat{S}}$

$$\textsc{VarB}\ \frac{\Gamma(x) = \{B \mid \psi\}}{\Gamma \vdash_{\mathbb{Q}} x :: \{B \mid \nu = x\}}$$

$$\textsc{Var}\forall\ \frac{\Gamma(x) = \forall\alpha_i . T \qquad \Gamma \vdash_{\mathbb{Q}} T_i}{\Gamma \vdash_{\mathbb{Q}} x :: [T_i/\alpha]T}$$

$$\textsc{AppB}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathbb{Q}} e_1 :: \texttt{let}\ C_1\ \texttt{in}\ (x\colon \{B \mid \psi\} \to T) \\ \Gamma; C_1 \vdash_{\mathbb{Q}} e_2 :: \texttt{let}\ C_2\ \texttt{in}\ T_x \\ \Gamma; C_1; C_2 \vdash T_x <: \{B \mid \psi\}\end{array}}{\Gamma \vdash_{\mathbb{Q}} e_1\ e_2 :: \texttt{let}\ C_1; C_2; y\colon T_x\ \texttt{in}\ [y/x]T}$$

$$\textsc{App}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathbb{Q}} e :: \texttt{let}\ C_1\ \texttt{in}\ (\_\colon T_x \to T) \\ \Gamma; C_1 \vdash_{\mathbb{Q}} t :: T'_x \qquad \Gamma; C_1 \vdash T'_x <: T_x\end{array}}{\Gamma \vdash_{\mathbb{Q}} e\ t :: \texttt{let}\ C_1\ \texttt{in}\ T}$$

$$\textsc{Abs}\ \frac{\Gamma \vdash_{\mathbb{Q}} (x\colon T_x \to T) \qquad \Gamma; x\colon T_x \vdash_{\mathbb{Q}} t :: \hat{T}' \qquad \Gamma \vdash \hat{T}' <: T}{\Gamma \vdash_{\mathbb{Q}} \lambda x.t :: (x\colon T_x \to T)}$$

$$\textsc{If}\ \frac{\begin{array}{c}\Gamma \vdash \hat{\psi} \qquad \Gamma; \hat{\psi} \vdash_{\mathbb{Q}} t_1 :: \hat{T}_1 \qquad \Gamma; \neg\hat{\psi} \vdash_{\mathbb{Q}} t_2 :: \hat{T}_2 \\ \Gamma \vdash_{\mathbb{Q}} T \qquad \Gamma \vdash \hat{T}_1 <: T \qquad \Gamma \vdash \hat{T}_2 <: T\end{array}}{\Gamma \vdash_{\mathbb{Q}} \texttt{if}\ \hat{\psi}\ \texttt{then}\ t_1\ \texttt{else}\ t_2 :: T}$$

$$\textsc{Match}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathbb{Q}} e :: \texttt{let}\ C\ \texttt{in}\ \{D \mid \psi\} \\ \mathtt{C}_i = T_i^j \to \{D \mid \psi'_i\} \qquad \Gamma_i = \{x_i^j\colon T_i^j\}; [x'/\nu]\psi'_i \\ \Gamma; C; [x'/\nu]\psi; \Gamma_i \vdash_{\mathbb{Q}} t_i :: \hat{T}_i \\ \Gamma \vdash_{\mathbb{Q}} T \qquad\qquad \Gamma \vdash \hat{T}_i <: T\end{array}}{\Gamma \vdash_{\mathbb{Q}} \texttt{match}\ e\ \texttt{with}\ |_i\ \mathtt{C}_i\langle x_i^j\rangle \mapsto t_i :: T}$$

$$\textsc{Fix}\ \frac{\Gamma \vdash_{\mathbb{Q}} S \qquad \Gamma; x\colon S^{\prec} \vdash_{\mathbb{Q}} t :: S' \qquad \Gamma \vdash S' <: S}{\Gamma \vdash_{\mathbb{Q}} \texttt{fix}\ x.t :: S}$$

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

Figure 3. Liquid type inference for SYNQUID programs

3.2 Modular Refinement Type Reconstruction

Type inference rules presented above are impractical to use in the context of synthesis. First, type inference only propagates type information *bottom-up*. In the context of synthesis this amounts to enumerating all well-typed terms of a given shape, and the checking the inferred type of a complete term against the specification. With this approach the benefit of refinement types for guiding the synthesis is only minimal compared to simple types. It is clearly desirable to propagate refinement types *top-down* whenever possible. For example, in order to check $\Gamma \vdash_{\mathbb{Q}} \texttt{if}\ \hat{\psi}\ \texttt{then}\ t_1\ \texttt{else}\ t_2 :: T$, it is sufficient to check that both t_1 and t_2 have the type T (under appropriate path conditions). The type system should make it possible to reject this program in case t_1 fails the check, before even considering t_2.

Second, practical algorithms for refinement type inference employ a two-phase approach to inferring polymorphic isntantiations: in the *shape reconstruction* phase the shape of T is determined using Hidley-Milner-style unification [17]; in the second phase, *refinement inference*, the previously found shape is used to create a template for T by inserting fresh *predicate unknowns* in place of refinements; after reducing subtyping constraints to subsumption over known and unknown refinements, the values of predicate unknowns are found by means of predicate abstraction. Since Hindley-Milner unification is *global*,

the whole program must be available before the refinement inference phase can even start.

In this section we develop a type reconstruction algorithm for SYNQUID that tackles these two challenges.

Bidirectional type reconstruction. We address the first challenge by combining liquid type inference with the ideas from *bidirectional type checking* [18]. The use of bidirectional type checking for synthesis was pioneered by [16] in the context of simple types augmented with input-output examples; in the realm of richer type systems, it has been employed for type reconstruction of various flavors of refinement types [4, 5, 25] and general dependent types [3]. To the best of our knowledge, the present work is the first to suggest using bidirectional checking for type reconstruction of general decidable refinement types, as present in the liquid types framework.

The idea of bidirectional reconstruction is to interleave top-down and bottom-up propagation of type information depending on the syntactic structure of the program, with the goal of making type checks as local as possible. Bidirectional typing rules use two kinds of typing judgments: an *inference* judgment $\Gamma \vdash_{\mathbb{Q}} t \Rightarrow S$ states that the term t generates type schema S in the environment Γ; a *checking* judgment $\Gamma \vdash_{\mathbb{Q}} t \Leftarrow S$ states that the term t checks against a known schema S in the environment Γ.

The bidirectional typing rules for SYNQUID are given in Fig. 4. The rules are split into two categories: inference rules have an inference judgment as their conclusion and encode the way the types of E-terms are constructed from their components types (bottom-up propagation); checking rules encode the way a checking judgment for an I-term is decomposed into simpler checking judgments for their components (top-down propagation). The rule I-E bridges the two directions: whenever an E-term e is being checked against a known type T, the type system switches to inference mode and then checks that the generated type of e is a subtype of T.

Incremental constraint solving. In order to turn the rules of Fig. 4 into a practical type reconstruction algorithm, we have to eliminate the remaining source of non-determinism: polymorphic instantiation. To do so, we replace the rule E-VAR$\forall$ with a deterministic rule that replaces the universally quantified type variable with a fresh free type variable:

$$\text{E-VAR}\forall\text{-ALG}\ \frac{\Gamma(x) = \forall \alpha_i . T \qquad \alpha_i' \text{ not free in } \Gamma \qquad \Gamma \vdash_{\mathbb{Q}} \alpha_i'}{\Gamma \vdash_{\mathbb{Q}} x \Rightarrow [\alpha_i'/\alpha_i]T}$$

With this rule, free type variables α'[2] may appear in places of "generated" refinement types in the typing and subtyping judgments of the rules in Fig. 4.

This section details our technique for *solving* subtyping constraints with free type variables, that is, finding a type assignment that maps those variables to liquid types in a way that validates subtyping judgments, or concluding that such an assignment does not exist.

Unlike existing approaches [12, 19], our algorithm interleaves the phases of shape reconstruction and refinement inference. It maintains the current set of subtyping constraints $\mathcal{C}$, the current *type assignment* $\mathcal{T} : \alpha' \to T$ and the current *liquid assignment* $\mathcal{L} : \kappa \to \psi$. At each step, the refinement type reconstruction algorithm may either decide to apply one of the inference or checking rules of Fig. 4, whereby possibly adding a new constraint to $\mathcal{C}$, or it may pick and remove an arbitrary c from $\mathcal{C}$ and solve it; solving c results either in failure (and conclusion that the program is ill-typed) or in extending $\mathcal{T}$, extending or strengthening $\mathcal{L}$, and/or adding new constraints to $\mathcal{C}$. This process is formalized in the procedure Solve in Fig. 5.

Intuitively, 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 repairs an invalid refinement subsumption by strengthening $\mathcal{L}$ (Eq. 8). The type reconstruction algorithm terminates when the entire type derivation is built, and any attempt to solve a constraint $c \in \mathcal{C}$ only inserts c back into $\mathcal{C}$ (that is, only Eq. 3, Eq. 9 apply).

Type Inference $\boxed{\Gamma \vdash_{\mathbb{Q}} e \Rightarrow \hat{T}}$

$$\text{E-VARB}\ \frac{\Gamma(x) = \{B \mid \psi\}}{\Gamma \vdash_{\mathbb{Q}} x \Rightarrow \{B \mid \nu = x\}}$$

$$\text{E-VAR}\forall\ \frac{\Gamma(x) = \forall \alpha_i . T \qquad \Gamma \vdash_{\mathbb{Q}} T_i}{\Gamma \vdash_{\mathbb{Q}} x \Rightarrow [T_i/\alpha]T}$$

$$\text{E-APPB}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathbb{Q}} e_1 \Rightarrow \texttt{let}\ C_1\ \texttt{in}\ (x\colon \{B \mid \psi\} \to T) \\ \Gamma; C_1 \vdash_{\mathbb{Q}} e_2 \Rightarrow \texttt{let}\ C_2\ \texttt{in}\ T_x \\ \Gamma; C_1; C_2 \vdash T_x <\colon \{B \mid \psi\}\end{array}}{\Gamma \vdash_{\mathbb{Q}} e_1\ e_2 \Rightarrow \texttt{let}\ C_1; C_2; y\colon T_x\ \texttt{in}\ [y/x]T}$$

$$\text{E-APP}\ \frac{\Gamma \vdash_{\mathbb{Q}} e \Rightarrow \texttt{let}\ C_1\ \texttt{in}\ (\_\colon T_x \to T) \qquad \Gamma; C_1 \vdash_{\mathbb{Q}} t \Leftarrow T_x}{\Gamma \vdash_{\mathbb{Q}} e\ t \Rightarrow \texttt{let}\ C_1\ \texttt{in}\ T}$$

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

$$\text{I-E}\ \frac{\Gamma \vdash_{\mathbb{Q}} e \Rightarrow \hat{T}' \qquad \Gamma \vdash \hat{T}' <\colon T}{\Gamma \vdash_{\mathbb{Q}} e \Leftarrow T}$$

$$\text{I-ABS}\ \frac{\Gamma; y\colon T_x \vdash_{\mathbb{Q}} t \Leftarrow [y/x]T}{\Gamma \vdash_{\mathbb{Q}} \lambda y.t \Leftarrow (x\colon T_x \to T)}$$

$$\text{I-IF}\ \frac{\Gamma; \psi \vdash_{\mathbb{Q}} t_1 \Leftarrow T \qquad \Gamma; \neg\psi \vdash_{\mathbb{Q}} t_2 \Leftarrow T}{\Gamma \vdash_{\mathbb{Q}} \texttt{if}\ \psi\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \Leftarrow T}$$

$$\text{I-MATCH}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathbb{Q}} e \Rightarrow \texttt{let}\ C\ \texttt{in}\ \{D \mid \psi\} \\ \mathtt{C}_i = T_i^j \to \{D \mid \psi_i'\} \qquad \Gamma_i = \{x_i^j\colon T_i^j\}; [x'/\nu]\psi_i' \\ \Gamma; C; [x'/\nu]\psi; \Gamma_i \vdash_{\mathbb{Q}} t_i \Leftarrow T\end{array}}{\Gamma \vdash_{\mathbb{Q}} \texttt{match}\ e\ \texttt{with}\ |_i\ \mathtt{C}_i\langle x_i^j \rangle \mapsto t_i \Leftarrow T}$$

$$\text{I-FIX}\ \frac{\Gamma; x\colon S^{\prec} \vdash_{\mathbb{Q}} t \Leftarrow S}{\Gamma \vdash_{\mathbb{Q}} \texttt{fix}\ x.t \Leftarrow S}$$

$$\text{I-TABS}\ \frac{\Gamma \vdash_{\mathbb{Q}} t \Leftarrow T \qquad \alpha_i \text{ not free in } \Gamma}{\Gamma \vdash_{\mathbb{Q}} t \Leftarrow \forall \alpha_i . T}$$

Figure 4. Bidirectional type reconstruction for SYNQUID programs

[2] We prime the names of free type variables to differentiate them from the universally quantified type variables of the outermost specification type.

$$
\begin{array}{ll}
\mathsf{Solve}(\Gamma \vdash c) = \texttt{match}\ c\ \texttt{with} & \\
\quad \mid \{\alpha' \mid \psi\} <: T, \alpha' \in \mathsf{dom}(\mathcal{T}) \longrightarrow & \\
\quad\quad \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 & \\
\quad\quad \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 & \\
\quad\quad \mathcal{T} \leftarrow \mathcal{T} \cup [\alpha' \to \mathsf{Fresh}(T)]; & \\
\quad\quad \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\colon T_x \to T_1) <: (y\colon T_y \to T_2) \longrightarrow & \\
\quad\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{\Gamma \vdash T_y <: T_x, & \\
\quad\quad\quad\quad \Gamma; y\colon T_y \vdash [y/x]T_1 <: T_2\} & (6) \\
\quad \mid \{D^m\ T_1^i \mid \psi_1\} <: \{D^m\ T_2^i \mid \psi_2\} \longrightarrow & \\
\quad\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{\Gamma \vdash \{D^m \mid \psi_1\} <: \{D^m \mid \psi_2\}, & \\
\quad\quad\quad\quad \Gamma \vdash T_1^i <: T_2^i\} & (7) \\
\quad \mid \{B \mid \psi_1\} <: \{B \mid \psi_2\}, \neg([\![\Gamma]\!] \wedge \psi_1 \Rightarrow \psi_2) \longrightarrow & \\
\quad\quad \mathcal{L} \leftarrow \mathsf{Strengthen}(\mathcal{L}, [\![\Gamma]\!] \wedge \psi_1 \Rightarrow \psi_2) & (8) \\
\quad \mid \{B \mid \psi_1\} <: \{B \mid \psi_2\}, [\![\Gamma]\!] \wedge \psi_1 \Rightarrow \psi_2 \longrightarrow & \\
\quad\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{\Gamma \vdash \{B \mid \psi_1\} <: \{B \mid \psi_2\}\} & (9) \\
\quad \mid \text{otherwise} \longrightarrow \text{fail} & (10)
\end{array}
$$

Figure 5. Incremental constraint solving

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. For a fresh predicate unknown κ, $\mathcal{L}(\kappa)$ is initialized with an empty conjunction, and the search space for κ is derived from the liquidness constraints issued by the rule E-VAR$\forall$-ALG.

As an example, starting from empty $\mathcal{T}$ and $\mathcal{L}$, $\mathsf{Solve}(\vdash \alpha' <: \texttt{List}\ \beta' \mid \texttt{len}\ \nu > 0)$ instantiates α' by Eq. 4 leading to $\mathcal{T} = [\alpha' \to \{\texttt{List}\ \gamma' \mid \kappa_0\}]$ and recycles the subtyping constraint; next by Eq. 1 and Eq. 7, the constraint is decomposed into $\vdash \texttt{List} \mid \kappa_0 <: \texttt{List} \mid \texttt{len}\ \nu > 0$ and $\vdash \gamma' <: \beta'$. If further type reconstruction produces a subtyping constraint on β', say $\texttt{Nat} <: \beta'$, $\mathcal{T}$ will be extended with an assignment $[\beta' \to \{\texttt{Int} \mid \kappa_1\}]$, which in turn will lead to transforming the constraint on γ' into $\vdash \gamma' <: \{\texttt{Int} \mid \kappa_1\}$ and instantiating $[\gamma' \to \{\texttt{Int} \mid \kappa_2\}]$, at which point all free type variables have been eliminated.

Predicate abstraction. The Strengthen procedure denotes the predicate abstraction step: $\mathsf{Strengthen}(\mathcal{L}, \psi_1 \Rightarrow \psi_2)$ makes sure that $\psi_1 \Rightarrow \psi_2$ holds by strengthening $\mathcal{L}(\kappa)$ for some conjunct κ of the left-hand-side, or fails if that is not possible. Finding weakest solutions for predicate unknowns is more expensive than finding strongest solutions, as in liquid types (exponential instead linear in the number of qualifiers [21]). In order to be practical, our implementation of Strengthen uses MARCO: an efficient algorithm for enumerating all minimal unsatisfiable cores due to Liffiton et al. [14].

3.3 Modular Program Synthesis

With a modular type reconstruction calculus at hand, we can now propose a modular synthesis calculus, following the approach of [16]. Bidirectional synthesis rules use two kinds of synthesis judgments. a *refinement-directed synthesis* judgment $\Gamma \vdash_{\mathbb{Q}} T \rightsquigarrow t$ states that in the environment Γ, refinement type T is inhabited by term t; for example, the rule for generating a conditional is:

$$\text{SI-IF}\ \frac{\Gamma \vdash_{\mathbb{Q}} \kappa \qquad \Gamma; \kappa \vdash_{\mathbb{Q}} T \rightsquigarrow t_1 \qquad \Gamma; \neg\kappa \vdash_{\mathbb{Q}} T \rightsquigarrow t_2}{\Gamma \vdash_{\mathbb{Q}} T \rightsquigarrow \texttt{if}\ \kappa\ \texttt{then}\ t_1\ \texttt{else}\ t_2}$$

A *shape-directed synthesis* judgment $\Gamma \vdash_{\mathbb{Q}} \tau \rightsquigarrow t :: T$ states that in the environment Γ, shape τ is inhabited by term t of type T (such that $\mathsf{shape}(T) = \tau$); for example, the rule for generating a variable is:

$$\text{SE-VARB}\ \frac{\tau = B \qquad \Gamma(x) = \{B \mid \psi\}}{\Gamma \vdash \tau \rightsquigarrow x :: \{B \mid \nu = x\}}$$

Since the transformation from type checking rules is straightforward, we omit the rest of the rules.

4. Evaluation

We evaluated SYNQUID on 50 benchmarks that include operations on standard and user-defined data structures and algorithms, with the goal of assessing *performance* and *expressiveness* of SYNQUID, as well as the *impact of different features* incorporated into the algorithm. Additionally, we compare SYNQUID to existing synthesis systems, with the aim to *cover broad range of synthesis techniques*, either by comparing to them directly or comparing to techniques that reportedly subsume them. We developed synthesis benchmarks which include benchmarks from prior work (adapted to refinement types) together with new synthesis problems. In the interest of space, we omit the full set of benchmarks, which are publicly available.

Our evaluation tries to answer the following questions: 1) How expressive and scalable is Synquid in terms of programs that it can synthesize (with respect to prior work)? 2) How expressive and concise are liquid types as synthesis specifications (with respect to other specification forms)? 3) How effective are different features incorporated on top of the core bidirectional synthesis algorithm?

4.1 Evaluating SYNQUID

We selected benchmarks from multiple categories, with the goal assessing *expressiveness* of SYNQUID for synthesis in various different domains. Integer benchmarks demonstrate reasoning about integers and non-structural recursive calls. List and other data structures leverage matching, structural recursion, the ability to make use of polymorphic functions (e.g. list map) and reason about universal properties of structure elements, and general measures (such as set of contained elements). Operations such as sorting and insertion into balanced

	Name	*Spec*	*#m*	*#cmp*	*components*	*Code*	*T-all*	*T-def*	*T-nis*	*T-ncc*	*T-nuc*	*T-nm*
Integer	maximum of 2 elements	7	0	0		11	0.02	0.02	0.03	0.03	0.02	0.03
	maximum of 3 elements	11	0	0		27	0.09	0.08	0.11	0.1	0.16	0.09
	maximum of 4 elements	15	0	0		51	0.6	0.48	0.44	0.45	46.31	0.44
	maximum of 5 elements	19	0	0		83	5.46	5.55	5.38	5.42	t/o	5.55
	addition	11	0	3	integer	26	1.38	1.42	15.72	2.41	t/o	0.00
List	is list empty	6	1	2	bool	6	0.02	0.02	0.02	0.02	0.02	0.02
	contains an element	6	2	2	bool	18	0.03	0.03	0.03	0.03	t/o	0.03
	duplicate each element	7	1	0		16	0.06	0.05	0.13	0.1	0.04	0.07
	list of element repetitions	7	1	3	integer	21	0.06	0.06	0.21	0.05	t/o	0.06
	append two lists	8	1	0		15	0.05	0.07	0.1	0.05	0.04	0.04
	concatenate list of lists	5	3	1	append	12	0.05	0.06	0.05	0.04	0.04	0.05
	take first n elements	11	1	2	integer	24	0.18	0.19	1.02	0.14	t/o	0.18
	drop last n elements	14	1	2	integer	20	0.29	0.3	t/o	0.29	t/o	0.28
	delete given element	8	2	0		26	0.09	0.1	0.12	0.1	t/o	0.09
	list map	5	1	0		22	0.02	0.02	0.03	0.03	0.02	0.02
	list zip with	10	1	0		33	0.06	0.06	t/o	0.21	0.06	0.06
	zip two lists	10	3	0		22	0.09	0.09	t/o	0.12	0.08	0.08
	list of integers to nats	8	1	2	map, negate	19	0.02	0.02	0.02	0.02	t/o	0.02
	Cartesian product	8	3	2	map, append	26	0.36	0.35	0.81	0.25	0.29	0.33
	reverse a list	11	2	1	snoc	12	0.29	0.29	0.61	0.39	0.81	0.73
	length with fold	4	2	3	fold, integer	19	0.1	0.78	0.27	0.19	t/o	0.2
Unique list	insertion	8	2	0		26	0.12	0.12	0.15	0.09	t/o	0.11
	deletion	8	2	0		22	0.1	0.09	0.23	0.11	t/o	0.08
	deduplication	8	4	4	bool, elem	30	0.65	3.26	t/o	1.11	t/o	0.70
	dedup subsequences	5	5	0		34	0.59	0.59	t/o	0.38	t/o	0.47
Sorting	insertion	8	2	0		49	0.26	0.27	0.46	0.26	t/o	0.30
	insertion sort	5	4	1	insert	12	0.07	0.07	0.07	0.05	0.06	0.28
	insertion (strict order)	8	2	0		49	0.27	0.28	0.46	0.22	t/o	0.25
	deletion (strict order)	8	2	0		37	0.13	0.13	0.48	0.13	t/o	0.12
	balanced split	31	4	0		33	1.85	1.18	t/o	2.52	4.8	1.89
	sorted merge	17	2	0		45	2.97	t/o	23.54	4.22	t/o	2.54
	merge sort	11	6	2	split, merge	25	2.52	2.77	16.61	2.58	2.14	2.71
	partition	27	4	0		40	4.47	14.6	t/o	4.97	t/o	3.74
	append pivot	28	2	0		22	0.27	0.29	0.89	0.36	0.35	0.26
	quick sort	11	6	2	partition, pivot append	22	3.83	27.05	t/o	3.42	8.43	3.88
Trees	membership	6	2	3	bool	28	0.33	0.35	0.85	0.33	t/o	0.32
	flatten to a list	5	2	1	append	18	0.23	0.27	t/o	0.93	0.19	0.22
	create balanced tree	7	2	2	integer	22	0.13	0.14	0.67	0.19	t/o	0.13
BST	membership	6	2	2	bool	37	0.12	0.12	0.15	0.11	t/o	0.11
	insertion	8	2	0		55	1.59	1.83	12.41	1.19	t/o	1.41
	deletion	8	2	1	merge	47	2.51	2.75	t/o	4.04	t/o	3.46
	BST sort	5	6	4	toBST, flatten, insert, merge	10	2.84	2.27	86.7	1.92	t/o	6.13
RBT	balance as two cases	65	3	2	colors	68	19.1	37.18	t/o	40.37	t/o	19.59
	insert	14	5	5	colors, singleton	51	13.4	42.2	t/o	62.5	t/o	14.2
Heap	membership	6	2	3		28	0.66	0.53	1.27	0.42	t/o	0.41
	insertion	8	2	0		47	0.82	0.83	4.23	0.6	t/o	0.70
	constructor	6	2	3		28	0.61	0.53	1.27	0.41	t/o	0.50
	constructor, 3 args	7	2	0		271	2.47	2.39	4.03	2.38	t/o	2.05
User	desugar AST	5	4	3	integer	46	0.94	1.09	107.61	0.85	0.59	0.72
	desugar AST with variables	11	6	3	integer	49	2.49	2.68	t/o	2.08	1.65	1.47

Table 1. Aggregated benchmark results of synthesizing functions with SYNQUID. For each benchmark given in *Name*, we report size of specification in *Spec* and synthesized function in *Code* (in AST nodes), number of defined measures in *#m*, extra components in *#cmp*, and running times (given in seconds, with *t/o* as timeout of 2 min.) of SYNQUID with default context in *T-def*, without X feature in *T-nm*, without incremental constraint solving in *T-Whole* and all features turned on in *T-Mod*.

trees demonstrate synthesis of non-trivial operations in the presence of complex invariants. Finally, "real world" benchmarks strive to demonstrate expressiveness for interesting synthesis problems that go beyond common data structure operations and algorithms. To ensure the validity of the evaluation, in addition to newly introduced benchmarks, we included 14 out of 22 of all the benchmarks from the liquid types verification tutorial [19] (some of them were not suitable for synthesis since they use insufficiently strong specifications, like vector bounds).

Results of evaluation of SYNQUID are summarized in Tab. 1. For each benchmark, the table reports the cumulative size of all refinements in the specification (in the size of the AST nodes), number of measures defined in addition to datatypes definitions, as well as the size of the generated code. Running times are reported for four variants of the main algorithm: *T-all*, with all features enabled; *T-nis*, where incremental solving is disabled; *T-ncc*, where checking consistency of function's type with current goal is disabled; *T-nuc*, where condition exploration with UNSAT-cores is disabled; and *T-nm*, where memoization of enumerated terms per context is disabled. *T-def* uses the same algorithm as *T-all*, with a unified context of qualifiers and parameters that is the same for all benchmarks in the same category. While SYNQUID infers qualifiers for types, qualifiers for conditionals need to be given manually. For each benchmark, the set of given qualifiers correspond to the semantic of declared data structures (e.g. $\leq$ relationship is given in benchmarks with sorted lists).

SYNQUID synthesized (and fully verified) programs in all the benchmarks. The results demonstrate that not only SYNQUID is *efficient* in synthesizing a variety of operations (45 out of 50 benchmarks are synthesized within 3 seconds), but also *scales* to specifications of large programs, including complex recursive (insertion into ordered trees of size 55) and non-recursive operations (binary heap construction of size 271). Even though sizes of specifications for some (simpler) benchmarks approach the size of the synthesized code (e.g. checking if list is empty and append pivot for quicksort), the benefits of *conciseness of liquid type specifications* grow significantly with the complexity of the synthesis problem at hand. Note that while definition of datatypes together with measure functions are reusable across synthesis problems, abstract refinement provided by SYNQUID allows further reusability of types instantiated with different predicates. Although default context (*T-def*) inevitably slows down synthesis, on most of the benchmarks the performance penalties were not drastic (note that SYNQUID timed out only on one benchmark). In most benchmarks the required additional components were either reused after being synthesized individually or represent straightforward structure operations (e.g. toBST and flatten in BST benchmarks).

Results demonstrate that crucial performance benefits come from the incremental solving and early consistency checking: solving the subtyping constraints only after application terms have been constructed and disabling early consistency checks, indeed leads to significant performance loss and failing synthesis within the imposed timeout (26 out of 50 ,*T-nis*, and 12 out of 50 benchmarks, *T-ncc*, respectively). Disabling UNSAT-cores (*T-nuc*) and memoization (*T-nm*) does not incur significant performance penalties, which suggests that on most of the benchmarks SYNQUID's modular synthesis allows exploring relatively small terms for both conditions and program terms, decreasing impact of further optimizations.

	Benchmark	*Spec*	*Time*	*TimeS*
LEON	deletion (strictly ordered)	14	15.1	0.13
	insertion (strictly ordered)	14	14.1	0.25
	merge sort	10	14.3	2.5
JEN	BST find	51	64.8	0.12
	bin. heap cstr (3 args)	80	61.6	2.21
	bin. heap find	76	51.9	0.54
MYTH	list sorted insert	12	0.12/22	0.28
	list compress	13	0.07/128.3	0.47
	BST insert	20	0.37/9.3	1.71
LS	list deduplication	7/-	231/-	0.70
	drop last/take except last	6/-	316.4/-	0.18
	tree membership	12/75	4.7/18	0.13
ESC	list compress	n/a	1	0.44
	create heigh-balanced tree	n/a	0.244	0.13
	list stutter	n/a	0.16	0.65
AL*	max of 4 elements	n/a	1.57	0.48
	max of 5 elements	n/a	4.15	5.54
STN	max of 4 elements	n/a	0.1	0.48
	max of 5 elements	n/a	0.18	5.54

Table 2. Comparison to other synthesizers. For each benchmark columns show: *Spec*, the size of the specification (or the number of input-output examples); *Time* reported running time of respective synthesizer (with default context [16] and random examples [6]); *TimeS*, running time of SYNQUID (as given in Tab. 1).

4.2 Comparative evaluation

We compare SYNQUID to state-of-the-art synthesizers that include not just those that focus on synthesis of recursive functional programs [1, 2, 6, 11, 16], but also imperative programs [13], as well as bounded synthesis [20] and constraints solving [15]. These systems include ones that use specifications expressed as logic formulas [11, 13] and constraints [15], input-output examples [1, 6, 16], and test-harnesses [2, 20]. While only two of the synthesizers provide full-functional verification of synthesized solutions [11, 13], other techniques validate solutions with test-execution [1, 16], deductive rules [6] and bounded verification [2, 15, 20].

Comparative evaluation is summarized in Tab. 1. For each synthesizer, we picked top three most complex of the respective benchmarks (judged by the synthesis running time) that are supported by the decidable fragment of logic used in SYNQUID. We provide running times only as a reference, since those represent existing running times, reported for each respective synthesizer.

Full-functional verification. The results show that specifications with refinement types can not only be *more concise*, but

can also lead to *more efficient* synthesis. Although both LEON and JENNISYS use variants of condition abduction and rely on SMT solving for verification (where LEON searches unbounded and unconstrained spaces of programs and relies on runtime execution), SYNQUID's representation of the search space that allows modular checking and early pruning significantly improves *scalability*: SYNQUID outperformed other synthesizers on all compared benchmarks. The comparison suggests that while specifications for imperative programs might become be elaborate, consolidating specifications with refinement types and measures can be shorter than explicit contracts and invariants.

Example-driven and test-driven synthesis. Our benchmarks include 3 out of 3, 10, and 5 most complex benchmarks synthesized by MYTH, λ^2 and ESCHER, respectively. SYNQUID could not synthesize include domain-specific structure manipulations (e.g. adding elements to tree of lists, suitable for operations such as map and fold) and operations not supported in the underlying logic (e.g. all nodes at a level in a tree). The comparison suggests that in addition to benchmarks that are more suitable for synthesis from logical constraints, SYNQUID is sufficiently expressive for synthesizing various benchmarks tailored for example-driven synthesis. SYNQUID's performance is comparable or better on all benchmarks, suggesting good *scalability* of modular synthesis even with guarantees of full-functional verification. While on some benchmarks, like `take` and `drop`, SYNQUID synthesizes correct programs when specifications are weaker (e.g. do not talk about which elements to take/drop), example-driven synthesis becomes slow or fails when given sub-optimal set of examples, in addition to require "trace-completeness" (`take` and `drop` require a minimum of 12 and 13 input-output examples, respectively [16]). This confirms that synthesis with SYNQUID is much less sensitive to variation in specification, albeit at the cost of expressiveness of synthesizing different but logically equivalent functions.

Bounded verification. To evaluate scalability in exploring large spaces of programs with simpler specifications, we compared SYNQUID to STUN [2] and ALLOY* [15] on synthesis of finding maximums of n variables. The comparison shows that performance of space exploration combined with additional reasoning in SYNQUID is indeed competitive even for large size of input variables. For some benchmarks, SYNQUID synthesizes unbounded versions, order of magnitude faster than same functions over constant numbers of elements (for searching list of only two values, STUN and ALLOY* need 0.13s and 1.6s, respectively).

5. Related Work

Our work is the first to leverage refinement types and polymorphism 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 reconstruction.

There have been a number of recent systems that target recursive functional programs and use type information in some form to restrict the searh space. The most closely related to our work are MYTH [16], LEON [11] and SYNTREC [9], as well as Escher [1] and λ^2 [6].

MYTH pioneered the idea of leveraging bidirectional type-checking as a way to help the synthesizer prune the search space. 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 can be simpler than writing a refinement type, and whereas there are some functions for which we cannot write a refinement type in our system, one can always provide more examples. That said, examples in general can never fully specify a program, so programming by example systems will always require a final step of manual checking to ensure that the generated solution is a correct one. Moreover, for some less intuitive problems, providing input output examples essentially requires that the programmer already know the algorithm and can step through it in her head—think red-black tree insertion, for example. Additionally, MYTH requires the set of examples provided 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. EScher and λ^2 are also programming-by-example systems, so relative to our system, they also have the same advantages and disadvantages outlined above.

SYNTREC also represents a different set of tradeoffs compared to our system. For example, SYNTREC requires a template of an implementation, and its main focus is in making such templates reusable and easy to write. SYNTREC also relies on bounded checking, which allows it to tackle problems for which full verification would be extremely challenging, but it also means that it cannot provide strong correctness guarantees. Additionally, SYNTREC relies on being able to symbolically evaluate the program in its entirety, so it does not have the same modularity properties that refinement types afford. Like our system, SYNTREC also uses a combination of symbolic and explicit search, but with a much stronger bias towards symbolic search.

In LEON the search is guided by expressive specifications with recursive predicates, and verification is based on a semi-decision procedure [22] on top of SMT solvers. The general idea is similar to ours: first decompose the specification and then switch to guess-and-check mode. Like our system, it also does symbolic search in the guesing phase using CEGIS, as well as condition abduction. That said, decomposition and guess-and-check are not as integrated as they are in our work. In particular, the decomposition does not leverage type information, while the guessing phase uses types but only takes advantage of the specification at the top level. LEON also does not support polymorphism or high-order functions, so it also cannot take advantage of the strong restrictions imposed by polymorphic types. Overall, LEON is complementary to our system in the sense that they pick a different trade-off in terms of specification expressiveness vs. decidability, which makes our system less expressive but more predictable.

References

[1] A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 934–950, 2013. doi: 10.1007/978-3-642-39799-8_67. URL http://dx.doi.org/10.1007/978-3-642-39799-8_67.

[2] R. Alur, P. Cerný, and A. Radhakrishna. Synthesis through unification. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 163–179, 2015. doi: 10.1007/978-3-319-21668-3_10. URL http://dx.doi.org/10.1007/978-3-319-21668-3_10.

[3] T. Coquand. An algorithm for type-checking dependent types. *Sci. Comput. Program.*, 26(1-3):167–177, 1996. doi: 10.1016/0167-6423(95)00021-6. URL http://dx.doi.org/10.1016/0167-6423(95)00021-6.

[4] R. Davies and F. Pfenning. Intersection types and computational effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 198–208, 2000. doi: 10.1145/351240.351259. URL http://doi.acm.org/10.1145/351240.351259.

[5] J. Dunfield and F. Pfenning. Tridirectional typechecking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 281–292, 2004. doi: 10.1145/964001.964025. URL http://doi.acm.org/10.1145/964001.964025.

[6] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015. doi: 10.1145/2737924.2737977. URL http://doi.acm.org/10.1145/2737924.2737977.

[7] C. Flanagan. Hybrid type checking. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 245–256, 2006. doi: 10.1145/1111037.1111059. URL http://doi.acm.org/10.1145/1111037.1111059.

[8] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 27–38, 2013. doi: 10.1145/2462156.2462192. URL http://doi.acm.org/10.1145/2462156.2462192.

[9] J. P. Inala, X. Qiu, B. Lerner, and A. Solar-Lezama. Type assisted synthesis of recursive transformers on algebraic data types. http://arxiv.org/abs/1507.05527, 2015.

[10] M. Kawaguchi, P. M. Rondon, and R. Jhala. Type-based data structure verification. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 304–315, 2009. doi: 10.1145/1542476.1542510. URL http://doi.acm.org/10.1145/1542476.1542510.

[11] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 407–426, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509555. URL http://doi.acm.org/10.1145/2509136.2509555.

[12] K. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 505–519, 2007. doi: 10.1007/978-3-540-71316-6_34. URL http://dx.doi.org/10.1007/978-3-540-71316-6_34.

[13] K. R. M. Leino and A. Milicevic. Program extrapolation with Jennisys. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 411–430, 2012. doi: 10.1145/2384616.2384646. URL http://doi.acm.org/10.1145/2384616.2384646.

[14] M. Liffiton, A. Previti, A. Malik, and J. Marques-Silva. Fast, flexible mus enumeration. *Constraints*, pages 1–28, 2015. ISSN 1383-7133. doi: 10.1007/s10601-015-9183-0. URL http://dx.doi.org/10.1007/s10601-015-9183-0.

[15] A. Milicevic, J. P. Near, E. Kang, and D. Jackson. Alloy *: A Higher-Order Relational Constraint Solver. Technical report, 2015.

[16] P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630, 2015. doi: 10.1145/2737924.2738007. URL http://doi.acm.org/10.1145/2737924.2738007.

[17] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

[18] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000. doi: 10.1145/345099.345100. URL http://doi.acm.org/10.1145/345099.345100.

[19] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169, 2008. doi: 10.1145/1375581.1375602. URL http://doi.acm.org/10.1145/1375581.1375602.

[20] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[21] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 223–234, 2009. doi: 10.1145/1542476.1542501. URL http://doi.acm.org/10.1145/1542476.1542501.

[22] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *Static Analysis - 18th International Symposium,*

SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings, pages 298–315, 2011. doi: 10.1007/978-3-642-23702-7_23. URL http://dx.doi.org/10.1007/978-3-642-23702-7_23.

[23] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 209–228, 2013. doi: 10.1007/978-3-642-37036-6_13. URL http://dx.doi.org/10.1007/978-3-642-37036-6_13.

[24] N. Vazou, E. L. Seidel, and R. Jhala. Liquidhaskell: experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 39–51, 2014. doi: 10.1145/2633357.2633366. URL http://doi.acm.org/10.1145/2633357.2633366.

[25] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 214–227, New York, NY, USA, 1999. ACM. ISBN 1-58113-095-3. doi: 10.1145/292540.292560. URL http://doi.acm.org/10.1145/292540.292560.