Close is Good Enough: Component-Based Synthesis Modulo Logical Similarity

ASHISH MISHRA, IIT Hyderabad, India
SURESH JAGANNATHAN, Purdue University, USA

Component-based synthesis (CBS) aims to generate loop-free programs from a set of libraries whose methods are annotated with specifications and whose output must satisfy a set of logical constraints, expressed as a query. The effectiveness of a CBS algorithm critically depends on the severity of the constraints imposed by the query. The more exact these constraints are, the sparser the space of feasible solutions. This maxim also applies when we enrich the expressivity of the specifications affixed to library methods. In both cases, the search must now contend with constraints that may only hold over a small number of the possible execution paths that can be enumerated by a CBS procedure.

In this paper, we address this challenge by equipping CBS search with the ability to reason about *logical similarities* among the paths it explores. Our setting considers library methods equipped with refinement-type specifications that enrich ordinary base types with a set of rich logical qualifiers to constrain the set of values accepted by that type. We perform a search over a tree automata variant called *Qualified Tree Automata* that intelligently records information about enumerated terms, leveraging subtyping constraints over the refinement types associated with these terms to enable reasoning about similarity among candidate solutions as search proceeds, thereby avoiding exploration of semantically similar paths.

We present an implementation of this idea in a tool called Hegel and provide a comprehensive evaluation that demonstrates Hegel's ability to synthesize solutions to complex CBS queries that go well-beyond the capabilities of the existing state-of-the-art.

ACM Reference Format:
Ashish Mishra and Suresh Jagannathan. 2025. Close is Good Enough: Component-Based Synthesis Modulo Logical Similarity. 1, 1 (August 2025), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Component-based synthesis (CBS) aims to generate loop-free programs from a library of *components*, typically defined as methods provided by an API. At the heart of any CBS implementation is a search problem over a hypothesis space of programs that "glue" components together using basic control primitives, such as conditionals and function applications. If the attributes defining the behavior of a component are not overly constrained, or when queries are reasonably general, the search for a feasible solution can be tractable. When this is not the case, however, the search can become substantially more difficult because the number of feasible programs that represent a solution is a much smaller fraction of the search space.

Intuitively, we can define CBS search as a reachability analysis over a graph that relates candidate methods based on their type or other similar defining attributes. For example, a node in this graph associated with a method that has a particular result type can be connected to any node

Authors' addresses: Ashish Mishra, IIT Hyderabad, India, mishraashish@cse.iith.ac.in; Suresh Jagannathan, Purdue University, USA, suresh@cs.purdue.edu.

XXXX-XXXX/2025/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

corresponding to a method that accepts an argument of this type. Such connections can be used by the synthesizer to produce a candidate solution that sequences these methods together, yielding a subgraph that connects input sources (e.g., function arguments) to output targets (i.e., queries).

Prior work [11, 16, 22] has considered the construction of such graphs using simple type-based specifications. In this paper, we propose to allow richer query specifications in the form of refinement types [21] that both decorate library methods and serve as the basis for synthesis queries. Fortunately, advances in automated theorem proving have made it increasingly common to have libraries be equipped with such rich specifications [5, 29], and there is no reason to believe that this trend will not continue to accelerate in the future, making our setup both practical and topical. Indeed, the idea of using refinement type specifications to guide a synthesis procedure has been explored in a number of other recent systems [25, 26]; our focus on devising an efficient CBS procedure that must contend with a sparse solution search space differentiates our work in a number of significant ways from these other efforts, as we describe below.

Devising an efficient CBS implementation in the presence of fine-grained, complex specifications enabled by the use of refinement types is challenging because the constraints defined by a type's refinement may greatly restrict the set of feasible solutions. Simple enumerative methods are, therefore, unlikely to be effective in this setting. To address this challenge, we devise a novel tree automata representation, called *Qualified Tree Automata* (QTA), as an extension of finite tree automata [6] with constraints. A distinguishing feature of a QTA is its support for logical implication constraints, which allows us to identify semantically-related portions of the automata. In particular, library methods and qualifiers in the method's refinement type are treated as symbols for the QTA; implication constraints over transitions allow modeling and reasoning about subtyping constraints directly within the automata. Consequently, any program accepted by a QTA is well-typed under the typing semantics of the refinement type system.

We leverage a QTA's structure to develop a CBS algorithm that tracks both (i) irrelevant portions of the automata, i.e., portions of the automata that do not correspond to well-typed terms, as well as (ii) semantic similarities between terms, leveraging the use of a logical subtyping relation during exploration to prune logically similar paths during search. Our main insight is that the notion of intersection available on finite tree automata naturally generalizes to a notion of *semantic* intersection over QTAs that can be exploited to yield an efficient enumeration of the term space. We present two QTA reduction procedures that concretize this intuition: a) a pruning strategy to eliminate unproductive (sub)automata and b) a semantic similarity mechanism that uses the type system's subtyping relation to identify logically similar terms. These techniques enable efficient exploration even when the set of feasible solutions is very small.

This paper makes the following contributions:

- We present a new approach to address scalability and expressivity limitations in existing CBS frameworks, especially in the presence of expressive type-based queries that impose significant semantic constraints on the set of feasible solutions.
- Our main insight entails directly embedding refinement-type specifications into a tree automata representation tailored to compactly represent sparse search spaces.
- We develop a specification-guided construction procedure of the search space used by the synthesizer using QTAs, instantiated with two reduction strategies to allow efficient pruning and compaction of the search space. We show that our algorithm is both sound and complete.
- We present a detailed evaluation study using Hegel, a tool that incorporates these ideas and provides CBS capabilities for OCaml programs. Our results demonstrate the feasibility of efficiently synthesizing complex CBS queries even when the solution space is very sparse.

These results support our claim that Hegel enables the solution of a variety of complex CBS problems that are outside the capabilities of existing approaches.

The remainder of the paper is organized as follows. In the next section, we provide additional motivation and a detailed overview of our synthesis approach. Section 3 provides background definitions. Section 4 formalizes QTAs. Section 5 presents our synthesis algorithm. Section 6 discusses our QTA reduction strategies. Soundness results are given in Section 7. Details about the implementation, along with benchmark results, are presented in Sections 8.4 and 9. Related work is given in Section 10, and conclusions are presented in Section 11.

2 MOTIVATION AND OVERVIEW

We motivate our approach using the synthesis problem shown in Figure 1. The synthesizer takes two inputs. The first is a library of OCaml functions specified using their type signatures. Figure 1(a) shows a portion of this library relevant to the example; it includes functions over lists, integers, tuples, etc. For example, splitAt is a function that takes an integer, a polymorphic list and returns a pair of polymorphic lists. For each function, we also present a commented-out refinement type specification, which can be ignored for the moment.

The second input to the synthesizer is a query, also represented as a type signature. This is given the name goal in Figure 1(b). A solution to the query is a synthesized OCaml function that, given two integers and a list, produces a pair of lists of the same type as its argument list. The query in this example is quite liberal in the solutions it admits. Figure 1(c) shows two out of many possible solutions.

To compute these solutions, the task of a component-based synthesizer is to *search* for a valid composition of functions found in the library, satisfying the type signature of each function such that the type of the composition aligns with the query type. A standard approach to solving CBS problems involves treating search as a kind of graph reachability problem in which a graph constructed over the types of library functions (henceforth type graph) is explored to find reachable paths between arguments and return types [11, 16, 22].

Figure 2a depicts this idea. We show the query arguments' types as source nodes and the required goal type ([a], [a]) as the target node. Edges connect “producers” and “consumers” of a type. We have replicated the nodes for some of the types like int, and [a] for the clarity of presentation. The figure shows multiple possible paths within the graph using library functions as intermediate transition nodes. Paths are color-coded, each representing a distinct candidate solution that leads from an input argument (e.g., x, y or z) to the target. For instance, the black solid edges represent the first solution shown in Figure 1(c). Similarly, the green paths represent the solution splitAt x (take y z). The pink path represents the solution splitAt y (drop (incr x)) z.

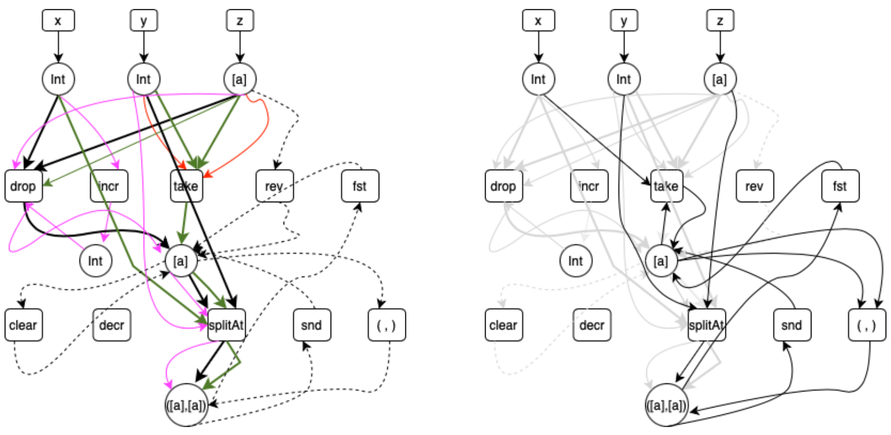
Dashed edges show other feasible paths in the graph, not all of which lead to a complete solution. Note that many paths have cycles and can thus represent a set of potential solutions. In this example, since there are multiple paths that lead to the target, a synthesis tool can easily find one and return it as a solution. Indeed, when tested with two state-of-the-art type-guided CBS tools [16] and [22], a correct solution was generated in less than 5 seconds.

Type-based CBS on refined queries. Now, consider a slight variation of the query that refines the desired solution by also requiring that in the returned pair, (i) the size of the first list is less than or equal to the first argument and (ii) the size of the second list is less than or equal to the original list length minus the second argument. One way to express this query is through the use of method predicates or type qualifiers like len, fst, snd, etc. to capture properties such as *length* of the list, *first* and *second* projections of a tuple, etc. The query represented in this way is given as the commented signature in Figure 1(b).

```
(a) A part of the library

(*take : (x : nat) -> (xs : [a]) ->
{v : [a] | len (v) ≤ x ∨ len (v) = 0}*)
val take : int -> [a] -> [a]

(*splitAt : (x : nat) -> (xs : [a]) ->
{v : (f : [a], s : [a]) | len (f) ≤ x ∧
(len (s) ≤ len (xs) - x)}*)
val splitAt : int -> [a] -> ([a],[a])

(*incr : (x : nat) -> {v:nat | x = x + 1}*)
val incr : int -> int

(*decr : (x : nat) -> {v:int | x = x - 1}*)
val decr : int -> int

(*fst : (x : ([a], [a])) ->
{v : [a] | v = fst (x)}*)
val fst : ([a], [a]) -> [a]

(*snd : (x : ([a], [a])) ->
{v : [a] | v = snd (x)}*)
val snd : ([a], [a]) -> [a]

(*flatten : (ts : a tree) -> {v : [a] |
elems(v)=elems(ts)}*)
val flatten : a tree -> [a]
```

```
(*parse : {xs : [a] | even_len(xs)} ->
{v : a tree | elems(v)=elems(xs)}*)
val parse : [a] -> a tree

(*clear : (xs : [a]) -> {v: [a] | len(v) = 0}*)
val clear : [a] -> [a]

(*drop : (x : nat) -> (xs : [a]) ->
{v : [a] | len (v) ≤ len (xs) - x}}*)
val drop : int -> [a] -> [a]

(*rev : (xs : [a]) -> {v : [a] | len(v)=len(xs) ∧
∀ u, w. ord (u, w, xs) => ord (w, u, v)}*)
val rev : [a] -> [a]


(b) A functional query type
(*goal : (x:nat)
-> (y : nat) -> (z : [a])
-> { v : (f : [a], s : [a]) | len (f) ≤ x
∧ (len (s) ≤ len (z) - y}*)
goal : int -> int -> [a] -> ([a], [a])

(c) A few correct solutions
 (*fun x y z ->
 ((take x
    (fst (splitAt y z))),
  snd (splitAt y z) )*)
 fun x y z -> splitAt y (drop x z)
 fun x y z -> splitAt x (take y z)
 ...
```

Fig. 1. Motivating Example.



(a) Types graph for the original example.

(b) Types graph for the refined example.

Fig. 2. Paths in the library for the original and refined query. We show multiple nodes for a type for ease of understanding.

Traditional enumerative exploration for this significantly more constrained query is unlikely to be successful for two fairly apparent reasons. First, the solution space, which was quite dense earlier, reflecting the fact that most paths chosen by the synthesizer for exploration would reach the target is now very sparse. Figure 2b shows the new type graph for this refined query. Notice that most of the earlier solutions shown in different colors are now invalid and greyed-out. Only the paths in solid back represent a feasible solution. The corresponding correct term is given in the comment in Figure 1(c). Second, the (base) type specifications associated with the libraries are too weak to meaningfully guide the synthesizer toward these sparse solutions or to filter out incorrect solutions. In particular, simple types are incapable of differentiating the correct (black) path from other previously-seen invalidated paths.

Exploration over an augmented search space. What we require is the ability to add additional structure to the graph beyond just the simple type signature that we currently have, to allow infeasible paths to be detected and pruned, and semantically-equivalent paths to be identified. Unfortunately, incorporating such additions comes at a cost both in terms of graph size and search complexity. To illustrate the issue with the former, note that a single type node int in the original graph may now expand into many nodes when qualified with semantic information that capture more refined properties, e.g., positive int, negative int, int less than v, etc. In fact, this set is unbounded in general. The issue with the latter directly impacts how the synthesizer is engineered. Existing techniques [11, 16, 22] search over a pre-built, fully expanded type graph of some size for the complete library coupled with a pruning mechanism over this graph. However, building such graphs in the presence of refined semantic information raises obvious scalability concerns. To address these issues requires a new graph representation that can concisely represent the space terms with refined specifications and a new search procedure tailored to operate over this new representation.

2.1 Approach

2.1.1 Synthesis Problem over Refinement Typed Libraries. A component-based synthesis problem over a refinement-type annotated library can be thus defined formally as follows:

Synthesis Problem. Given a type environment Γ that relates library functions $f_i = \lambda(\overline{x_{i,j}}).e_{f_i}$ with their refinement types $f_i : \overline{(x_{i,j} : \tau_{i,j})} \rightarrow \{v : t_i \mid \phi_i\} \in \Gamma$, and a synthesis query $\varphi = \overline{(y_i : \tau_i)} \rightarrow \{v : t \mid \phi\}$, a solution to a CBS problem seeks to synthesize an expression e, possibly using f_i, such that $\Gamma \vdash e : \varphi$ holds.

2.1.2 Compact Representation of the Search Space. A primary requirement towards solving the above problem is to compactly represent the space of well-typed terms. There are several data structure options that we might choose from that can serve this purpose. Version Space Algebras [?], e-graphs [3] and Finite Tree Automata (FTA) [6] are three well-studied examples. In particular, FTAs have been shown to be effective in representing the space of untyped programs and allowing efficient search over them, satisfying a set of input-output examples [12]. Furthermore, extensions of FTA with equality constraints, dubbed ECTAs [6, 22], have been shown to be a useful representation to represent simply-typed programs.

Unfortunately, these approaches are ineffective in representing the space of programs with refined specifications, such as those considered in our motivating example. For instance, VSA and standard FTAs lack the ability to relate subprograms, while constrained FTAs [6, 22], which allow *syntactic* equality constraints between subterms, are insufficient when *logical* equality/implication constraints are required. This requirement is clearly seen in the refined variant of our motivating example where the synthesis query establishes non-trivial semantic relationships between the list elements in the output. [1]

Solution: Qualified Tree Automata. To address these limitations, we introduce a new data structure, a *Qualified Tree Automata* (QTA), that allows us to capture such logical constraints. A QTA supports a richer alphabet than other FTA variants by incorporating logical qualifiers from decidable first-order theory fragments. While it also allows constraints on its transitions similar to other constrained tree automata [6, 22**?**], it additionally supports semantic reasoning over these constraints (e.g., logical entailment), rather than being limited to syntactic reasoning using equality or dis-equality constraints.

For instance, Figure 3 presents a QTA that captures the space of terms represented by the following sentence $\{\ f(t_1, t_2) \mid t_1, t_2 \in \{\phi_1, \phi_2, \phi_3\} \wedge t_1 \implies t_2\ \}$ where both sub-trees are constrained using a *logical entailment* constraint (defined later) on the transition ($\mathtt{l} \vDash \mathtt{r}$). Here, $\mathtt{l}$ and $\mathtt{r}$ are variables that capture a specific location in the automata and the constraint restricts which choice of t_1 and t_2 are acceptable.

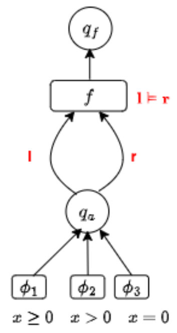


Fig. 3. A simple QTA for $\{\ f(\phi_1, \phi_2) | \phi_1, \phi_2 \in \text{FOL}, \phi_1, \implies \phi_2\ \}$

These formulae and the constraints on the transition together restricts the language accepted by the QTA. In this example, the automata accepts terms $f(\phi_2, \phi_1)$ and $f(\phi_3, \phi_1)$ since, in both cases, the constraint $\phi_i \implies \phi_j$ holds but the QTA rejects other syntactically valid terms like $f(\phi_1, \phi_2)$ and $f(\phi_1, \phi_3)$.

Embedding Refinement-typed space using QTA. Unlike traditional typed-program space embedding where syntactic constraints are sufficient to compare base types, the space of well-typed refinement specifications not only use constant base types like int, char, bool, etc., but also allow types to be qualified with logical formula (aka refinements) [?]. QTA are effective in expressing these structures and enable a compact embedding of these type structures.

To illustrate such an embedding, Figure 4 shows a transition in a trivial QTA, representing a variable x having a type $\{\ v : \text{int} \mid 10 \geq v \geq 0\ \}$. States are shown as circles, and transitions as rectangles. A transition can have zero or more incoming states, with each having a *position* label, shown over incoming arrows in transitions in red. For instance, the transition $(q_0 \xrightarrow{\text{type}} \boxed{\text{x}})$ is used to capture a standard type binding (x : τ). These types can be *refined*, by allowing transitions for an alphabet symbol τ (drawn from the set of base refinement types). τ is a ternary symbol, with three incoming states (incoming arrows from states) corresponding to the parameters, *variable*, *base-type* and a *logical formula* (i.e., refinement). The states themselves have incoming transitions, and so on. In this example, the automata rooted at q_0 corresponds to the refinement type $\{\ v : \text{int} \mid 10 \geq v \geq 0\ \}$.

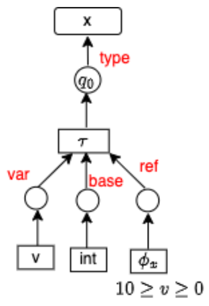


Fig. 4. A simple QTA for binding (x : $\{\ v : \text{int} \mid 10 \geq v \geq 0\ \}$)

2.1.3 *Efficient Enumeration.* Compactly representing the program space is only part of the solution, however. Another significant challenge for CBS is devising an efficient enumeration of the program space. For instance, Figure 5 shows some possible terms, involving functions take, incr, and clear, along with their refinement types, that a synthesizer may have to enumerate. In our setting, naïve enumeration is not feasible due to the sparseness of the solution space. In Figure 5, for instance,

[1]see supplemental material for a detailed comparison and limitation of these structures.

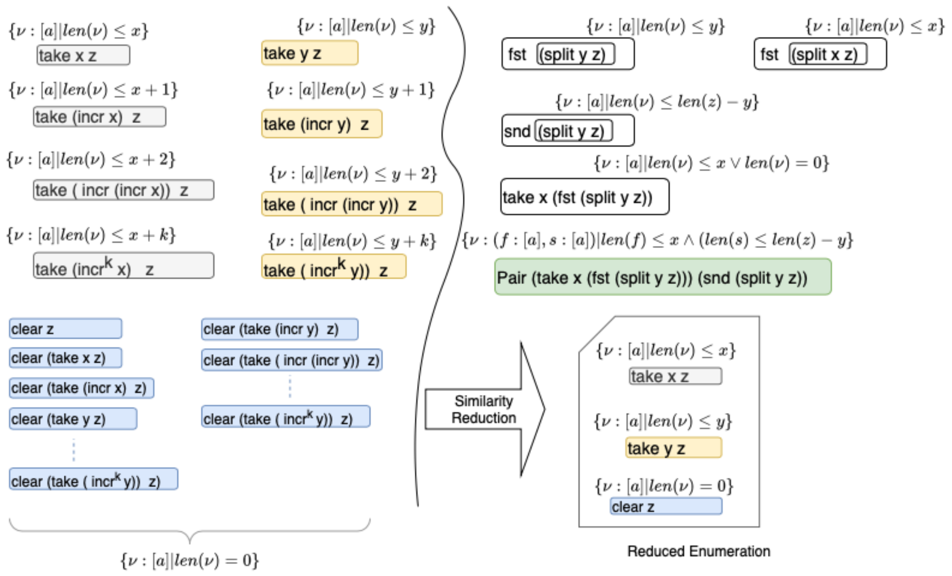

$\{\nu : [a] | len(\nu) \leq x\}$
take x z
$\{\nu : [a] | len(\nu) \leq x+1\}$
take (incr x) z
$\{\nu : [a] | len(\nu) \leq x+2\}$
take (incr (incr x)) z
$\{\nu : [a] | len(\nu) \leq x+k\}$
take (incr^k x) z
$\{\nu : [a] | len(\nu) \leq y\}$
take y z
$\{\nu : [a] | len(\nu) \leq y+1\}$
take (incr y) z
$\{\nu : [a] | len(\nu) \leq y+2\}$
take (incr (incr y)) z
$\{\nu : [a] | len(\nu) \leq y+k\}$
take (incr^k y)) z
clear z
clear (take x z)
clear (take (incr x) z)
clear (take y z)
clear (take (incr^k y)) z)
clear (take (incr y) z)
clear (take (incr (incr y)) z)
clear (take (incr^k y)) z)
$\{\nu : [a] | len(\nu) = 0\}$
$\{\nu : [a] | len(\nu) \leq y\}$
fst (split y z)
$\{\nu : [a] | len(\nu) \leq x\}$
fst (split x z)
$\{\nu : [a] | len(\nu) \leq len(z) - y\}$
snd (split y z)
$\{\nu : [a] | len(\nu) \leq x \lor len(\nu) = 0\}$
take x (fst (split y z))
$\{\nu : (f : [a], s : [a]) | len(f) \leq x \land (len(s) \leq len(z) - y\}$
Pair (take x (fst (split y z))) (snd (split y z))
Similarity Reduction
$\{\nu : [a] | len(\nu) \leq x\}$
take x z
$\{\nu : [a] | len(\nu) \leq y\}$
take y z
$\{\nu : [a] | len(\nu) = 0\}$
clear z
Reduced Enumeration

Fig. 5. A partial set of terms in the augmented type graph for the example along with their augmented refinement type. The curved line partitions the terms which lead to a solution (shown in green) towards the right (top) and other explored terms which although type-correct do not lead to a solution (left). The right bottom shows the list of terms produced by Hegel.

only a fraction of the overall space, specifically the terms on the right top of the curve in the figure, are actually relevant to the solution. The terms on the left of the curve in Figure 5, although type correct, are irrelevant to the result.

Furthermore, refinement type information alone is often insufficient to efficiently guide the synthesizer to a possible solution. For example, in the figure, there is little guidance available to determine that the terms on the left of the curve are unlikely to contribute to a solution compared to the terms given on the right. The cost of enumerating these ineffective terms becomes problematic as larger terms are built from them. To address the enumeration challenge, our synthesis procedure exploits opportunities to *eagerly* prune infeasible as well as redundant portions of the search space while still ensuring that the enumeration procedure is complete.

Eager equivalence reduction: This enumeration cost can be mitigated if we can somehow prune out infeasible and redundant terms during the enumeration process. Notice in Figure 5 that many of these terms, although syntactically different, have the same (or equivalent) refinement type. For example, consider all the clear function calls, which are shown in blue in the lower left of the figure. These are all distinct terms, in total $(2k + 1)$ in number for any k incr calls, but each has the same type, $(\{\ \nu : [a] \mid len(\nu) = 0\ \})$. We utilize these type-equivalences to prune out all but one of these terms. The high-level intuition is that for any given query and library, all or none of these nodes will lead to a solution, so exploring along any one is sufficient to reach a solution if one exists using them. For instance, all the blue terms can be replaced with a single term (clear z) shown in the lower right portion of the figure, showing the reduced space. We might think that such equivalences occur only between terms with overlapping structures, like all the terms in blue. However, note that this equivalence reduction can be generalized to any arbitrary pair of terms, e.g., see (take x z) on the left of the curve, and (fst (splitAt x z)) on the right.

Similarity, rather than equivalence: Tracking precise equivalences, although useful as shown above, allows only a marginal reduction of the augmented space. To see why, consider all the terms in Figure 5 in the upper left corner shown in the grey and yellow colored boxes.

Because these terms have valid refinement types, they are candidates for enumeration by the synthesis search procedure. However, none of these terms actually contribute to the solution. Unfortunately, exact equivalence tracking, as described earlier, is ineffective in pruning these terms since their type annotations are all distinct. Observe, however, that the types of these terms are related to one another under a refinement *subtyping* relation. For example, the refinement type for the first term in grey (take x z), is a subtype of the refinement types of all other grey terms. Similarly, the type of the first yellow term is a subtype of the type of all other yellow terms. By defining a weaker *similarity* relation between terms based on the subtype relation, we can avoid exploring possibly non-equivalent but *similar* terms without affecting the completeness of the search procedure. The use of a subtyping relation as a proxy for similarity allows us to collapse, for example, all grey and yellow terms into two representative candidates, as depicted in the bottom-right of the figure.

Our main insight is to map the similarities and equivalences between transitions in automata to the subtyping relation in the type system. This allows us to use operations available on QTAs like intersection and union to prune all but the most specific term (i.e., term with the lowest type in the subtype ordering).

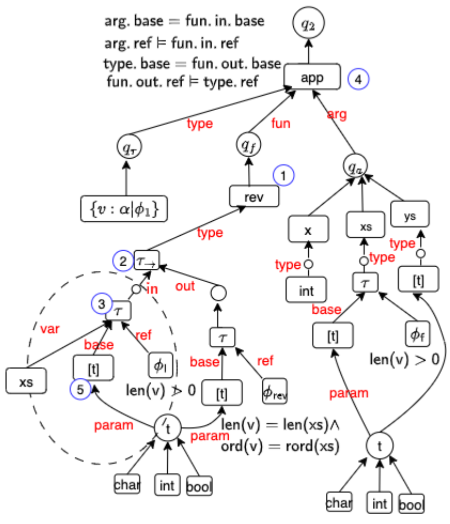


Fig. 6. A partially-defined QTA for our motivating example. States are shown as circles, and transitions as rectangles. Some unimportant states and transitions are omitted for clarity. A transition can have zero or more incoming states, with each having a label denoting its *position*, shown in red. Transitions can also have constraints over these positions. (1) are labels used for elucidation.

Irrelevant candidate elimination: Another thing to notice in the example is that many functions in our example library, e.g., parse, flatten, etc. are irrelevant, as there are no available arguments of the required type in the query or other library methods to invoke them. This can happen due to typing restrictions on the arguments, for example, the parse function requires that length of the input list to be 'even'. Or, it can happen simply because there are no terms of the required type in the library or the terms that have been enumerated. Pruning away such functions early on allows us to avoid not only reducing the candidate space but potentially avoids the need to enumerate many other larger terms built using them for example, (parse (flatten . . .)), (clear (flatten (parse . . .))), (take x (flatten (parse . . .))), (take y (flatten (parse . . .))), etc.

However, a naïve enumeration would generate potentially many irrelevant functions and terms, not all of which are even well-typed. Efficiently filtering these terms is likely to be infeasible, in general, given the enumerative structure of the approach, and the cost of performing type checks, both of which involve an SMT query. Our next insight is that we can use operations over the QTA to prune out portions of the QTA representing such terms.

2.1.4 *QTA Through Examples.*

We highlight some of the details of *Qualified Tree Automata* using a detailed example.

Consider Figure 6 that depicts a portion of the QTA for our motivating example, showing the annotated library function rev as a transition $\boxed{\text{rev}}$ (we have used labels like ① for elucidation), along with its refinement type as another transition ($\boxed{\tau_{\rightarrow}}$) (②). Note that because this is a function type, it has two children, one for its input *argument* type and another for its *return* type, with location labels in and out respectively. The argument's type is represented in the usual way, as shown in Figure 4 (the dashed circle at ③). Transition $\boxed{\text{rev}}$ also has an outgoing edge to state q_f, representing the set of unary functions. In general, for each $n-$ary function symbol, under q_f, the in and out positions shows the sub-automata modeling the argument(s) and result type for the function.

QTAs also support polymorphic refinement types. For instance, a polymorphic list type [a] is represented by a transition with list constructor [t] (⑤) with a type parameter t and another automata for all base type t that can be present on the incoming edge representing the constructor's type parameter.

A QTA also efficiently captures refinement typing semantics. For instance, transition ($\boxed{\text{app}}$)(④) captures the expected type application rule. It has three incoming states, q_f, representing a set of unary functions, q_a, a set of possible variables (along with their types) as arguments ((x : int), (xs : { [a] | len (v) > 0}) and (ys : [a])), and q_τ that represent the inferred type for the application. Application typing constraints are added as constraints to the app transition, which captures not only equality for base types, function return types, the return type for the application term (type.base = fun.out.base), the function input type, and the argument base type, but also allows constraints with logical entailment $\models$ and other constraints generated through logical connectives like $\wedge$, $\vee$ etc. Thus, arg.ref $\models$ fun.in.ref establishes a logical entailment constraint between a function's formal and actual refinements.

3 PRELIMINARIES

3.1 Background: Finite Tree Automata (FTA)

A ranked alphabet ($\mathcal{F}$, Arity : $\mathcal{F} \mapsto \mathbb{N}$) is a pair consisting of a set of symbols ($\mathcal{F}$), and a map that relates these symbols to their arity. In the context of CBS, these symbols are those found in expressions, library functions provided by the synthesis problem, query arguments, and associated types and refinements for each expression. For example, constants and variables are symbols of arity zero, as are constructors without arguments like Nil, etc. These form a set $\mathcal{F}_0$. A *base refinement type* is represented as a symbol τ with arity 3 as shown in in Figure 1. In general, the set of all symbols of arity n is denoted as $\mathcal{F}_n$.

Definition 3.1 (Finite Tree Automata). A bottom-up finite tree automaton (FTA), $\mathcal{A}$ over a signature of ranked alphabet ($\mathcal{F}$, Arity : $\mathcal{F} \mapsto \mathbb{N}$ is a tuple $(Q, Q_f, \mathcal{F}, \Delta)$ where Q is a set of states, $Q_f \subseteq Q$ is a set of final states and Δ is a set of transition rules of the following form : $f(q_1(x1), ..., q_n(xn)) \longrightarrow q(f(x1, ..., xn))$, where n ≥ 0, $f \in \mathcal{F}_n$, $q, q_1, ..., q_n \in Q$, $x_1, ..., x_n \in \mathcal{X}$. where $\mathcal{X}$ is a set of variables (symbols with arity 0).

Example 3.2. Let the signature over which a tree automata is defined be given by a pair of symbols $\mathcal{F} = \{f, g, a\}$ and Arity = $\{f \mapsto 2, g \mapsto 1, a \mapsto 0\}$. The set of states is $Q = \{q_a, q_g, q_f\}$, let the final state be $q_f \in Q_f$ and let Δ be given by the following transition rules:

$$\{a \longrightarrow q_a(a) \; ; \; g(q_a(x)) \longrightarrow q_g(g(x)) \; ; \; g(q_g(x)) \longrightarrow q_g(g(x)) \; ; \; f(q_g(x), q_g(y)) \longrightarrow q_f(f(x, y))\}$$

Analogous to finite automata that accept strings over an alphabet of characters, an FTA accepts trees, which are terms over $\mathcal{F}$. In the example given above, the FTA accepts all valid trees of the form $f(...)$, generated using functions f, g and the constant a.

3.2 Synthesis Language, λ_{qta}

The target language of our synthesizer is a standard A-normalized [14] call-by-value typed λ-calculus with constructors, constants and variables, conditional expressions, and function abstraction and application. [2] To simplify the presentation, in the following, we assume all variables have a single unique binding-site.

λ_{qta} types include standard base types like int, bool etc., along with algebraic types like lists and trees over these base types. Refinement types τ, include *base refinements* and *arrow refinements*. A base refinement $\{\ v : \mathsf{t} \mid \phi\ \}$ qualifies a term of base type t with a refinement qualifier $\phi \in \Phi$. An arrow refinement refines a function type, where the argument x can occur free in the return type. Qualifiers (Φ) is a set of first-order predicate logic formulae over base-typed variables along with method predicates (Q), which are user-defined, uninterpreted function symbols such as len and ord over lists used in our motivating example. A type context Γ records term variables and library functions g with their types. It also records a set of propositions relevant to a specific context.

4 QUALIFIED TREE AUTOMATA

Definition 4.1 (Positions in a term). A position p in a term t is of the form $i.j.k....n$, a sequence of positive integers describing a path from the root of t to a sub-term. This describes what symbols are present at each position, relative to the root.

For easier comprehension, we give human-readable labels to each number in a position, e.g., consider Figure 6 again. The sequences used in the constraints like arg.base = fun.in.base are positions. type.ref is a synonym for position 1.3, fun.in.base for 2.1.2, etc.

$$
\begin{array}{ll}
p, p_i, ...p_n \in \mathit{Position} & \\
\psi_a ::= & (\mathit{Atoms}) \\
\mathsf{true} \mid \mathsf{false} & \\
\mid p = p & (\mathit{Syntactic\ equality}) \\
\mid p \vDash p & (\mathit{Semantic\ entailment}) \\
\psi \in \Psi ::= & \\
\psi_a \mid \neg\, \psi & \\
\mid \psi \wedge \psi \mid \psi \vee \psi & \\
\sigma \in \mathit{Schema} ::= & \\
\star \mid \star = \star \mid \star \vDash \star &
\end{array}
$$

Fig. 7. QTA Constraints Ψ

Definition 4.2 (Constraints in Qualified Tree Automata). A QTA constraint $\psi \in \Psi$ is a predicate on terms in λ_{qta} . It is defined inductively over positions and Boolean connectives, as shown in Figure 7. A valid *atomic constraint* ψ_a includes Boolean constants like true and false, as well as *syntactic equality* between positions, given by $p = p$ and *semantic entailment* (*Sem-ent*) over positions, given by $p \vDash p$. A *constraint* ψ is either a ψ_a or a constraint generated using Boolean connectives over other constraints. Consequently, we can also classify *atomic constraints* into *kinds* using three constraint schemas, $\star$, $\star = \star$ and $\star \vDash \star$, respectively.

Definition 4.3 (Qualified Tree Automata). A *Qualified Tree Automata*, $\mathcal{A}$ defined over a finite ranked alphabet $\mathcal{F}$ derived from λ_{qta} , is a tuple (Q, $\mathcal{F}$, Q_f, Δ), where:

- Q is a finite set of states.
- $Q_f \subseteq Q$ is a set of final states.
- $\Delta \subseteq Q^n \times \mathcal{F} \times \Psi \mapsto Q$, is a set of constrained *transitions*. Each transition rule is of the form $f(q_1, q_2, ...q_n) \overset{\psi}{\hookrightarrow} q$, where $f \in \mathcal{F}$ with arity n, and a set of states $q_1, q_2, \ldots q_n \in Q$ and $\psi \in \Psi$ is a valid constraint. Here q is the *target state.*

[2]Details about the language and its type system are provided in the supplemental material.

$$
\begin{array}{ll}
[\![q]\!] & ::= \bigcup_i \, ([\![\delta_i]\!]) \\
 & \text{where } \delta_i = \\
 & (f(q_{i_1}, q_{i_2}, \ldots, q_{i_n}) \overset{\psi}{\hookrightarrow} q) \\
[\![\delta]\!] & ::= \{\, f\ \overline{t_i} \mid t_i \in [\![q_i]\!], f\ \overline{t_i} \Vdash \psi, \\
 & i \in [1 \ldots n] \,\} \\
 & \delta = (f(q_1, q_2, \ldots, q_n) \overset{\psi}{\hookrightarrow} q) \\
[\![\delta_\perp]\!] & := \varnothing \\
[\![\mathcal{A}]\!] ::= & \bigcup_i \{[\![q_i]\!] \mid q_i \in Q_f\}
\end{array}
$$

Fig. 8. QTA denotation.

Language of a *Qualified Tree Automaton* $\mathbb{L}$ ($\mathcal{A}$). The language accepted by a QTA $\mathcal{A}$, is the set of all terms in λ_{qta} with some successful run of $\mathcal{A}$. This is, in fact, the set of all well-typed λ_{qta} terms, constructed using the methods found in a library. Formally, we define the language accepted by $\mathcal{A}$ using its denotation $[\![\mathcal{A}]\!]$ (see Figure 8). The denotation of a state q, $[\![q]\!]$ is the set union of the denotations of each of the transitions δ_i, $[\![\delta_i]\!]$, for which q is a target state. The denotation of a transition δ builds a set of all terms, using the symbol at the current transition (f) and terms in the denotation of states incoming in δ, filtering all terms that do not satisfy the transition constraint ψ. Symbols $f \in \mathcal{F}$ include all λ_{qta} terms including variables (x), constants (c), conditional expressions (if b then e else e), function abstractions and applications, and types (τ). Intuitively, the satisfaction of a constraint by a term $t \Vdash \psi$ maps syntactic equality constraints to equality of symbols and semantic entailment between qualifiers to logical entailment of FOL formulas. We also have a special bottom transition $\delta_\perp$, whose denotation is an empty set. Since a QTA can have multiple final states given by the set Q_f, the language of a QTA is the union of the denotation for all its *final states*.

5 SYNTHESIS USING *QUALIFIED TREE AUTOMATA*

5.1 Component-based synthesis using QTA

To formalize synthesis using QTAs, we revise our earlier definition. First, we define a consistency relation between a QTA $\mathcal{A}$ and a typing environment Γ.

Definition 5.1 (Consistency between a QTA $\mathcal{A}$ and Type Environment Γ). A type environment Γ is *consistent* with a QTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ iff $\forall e\ .\ \Gamma(e)$ [3] $= \tau \iff \exists \mathcal{A}'$ such that $\mathcal{A}'$ is a sub-automaton of $\mathcal{A}$ and $e \in [\![\mathcal{A}']\!]$.

Revised Synthesis Problem: Given a type environment Γ that relates library functions $f_i = \lambda(\overline{x_{i,j}}).e_{f_i}$ with their refinement types $f_i : \overline{(x_{i,j} : \tau_{i,j})} \rightarrow \{\nu : t_i \mid \phi_i\} \in \Gamma$, and a synthesis query $\varphi = \overline{(y_i : \tau_i)} \rightarrow \{\nu : t \mid \phi\}$, a solution to a CBS problem is a QTA $\mathcal{A}$, such that forall all $e \in [\![\mathcal{A}]\!]$, $\Gamma \vdash e : \varphi$ and Γ is consistent with $\mathcal{A}$.

5.2 Synthesis Algorithm

The main synthesis algorithm, QTASynthesize is shown in Algorithm 1. It takes as input an alphabet $\mathcal{F}$, which includes symbols from λ_{qta} and a library of functions with refinement type annotations; a synthesis query specification φ, and a bound on the size of the terms k to synthesize.

The algorithm works in two phases: (1) an *exploration* phase adds states and transitions, expanding the automata. The resulting QTA is then pruned/reduced by (2) a reduction phase. The algorithm also keeps track of *similar* but not yet reduced transitions through an equivalence set $\mathcal{E}$, lifting the subtyping relation to an ordering relation between transitions in QTA.

The output of the algorithm is a pair consisting of (i) a QTA, $\mathcal{A}_{\text{min}}$ for the synthesis query based on $\mathcal{F}$ and the typing semantics of λ_{qta} , such that the language of the QTA are solutions to

[3]Expression e and variable binding e are used interchangeably.

the synthesis problem, and (ii) a set of solution terms in λ_{qta}, possibly using $\mathcal{F}$ that satisfies the query specification, The algorithm returns a failure value ($\perp$) if it cannot find a solution within the given max-depth k.

The algorithm begins (line 1) by initializing $\mathcal{E}$ to an empty set and constructing an initial QTA covering all terms of size one using a call to a well-formedness function (WF), passing it the library $\mathcal{F}$ and an empty automata $\mathcal{A}_\perp$. This function is a deterministic implementation of the rules rules shown in Figure 9. These rules formalize when to add transitions corresponding to well-formed primitive types (WF-PRIM), predicates (WF-PRED), and variables (WF-VAR), in the component library. They also define how to add states and transitions for each of the query arguments x_i and their types τ_i. Additionally, there is also a rule Q-GOAL suggesting how to add a transition (and corresponding states) for the query specification. Consequently, initialization adds states and leaf transitions for the arguments in the synthesis query φ, library functions, base types, etc. Initialization also adds a final state and transition with the top-level constraint corresponding to the given query φ. This generates the initial QTA $\mathcal{A}_0$.

```
QTASynthesize(⟨$\mathcal{F}, \varphi = \overline{(\mathsf{x_i} : \tau_i)} \rightarrow \{\mathsf{v} : \mathsf{t} | \phi\}, k$⟩)
    // Initialize
(1) $\mathcal{A}_0 \leftarrow$ WF ($\mathcal{F}, \mathcal{A}_\perp$); $\mathcal{E} \leftarrow \emptyset$
    // Check solution in Initial $\mathcal{A}_0$
(2) if $\hat{Q}$ = NEmpty ($\mathcal{A}_0$) then
(3)     return ($\mathcal{A}_0, \bigcup_{q \in \hat{Q}} [\![\mathcal{A}_q]\!]$)
    // Iteratively explore-reduce-check
(4) return Enumerate ($\mathcal{A}_0, \varphi, k$)
Enumerate(⟨$\mathcal{F}, \mathcal{A}, \varphi = \overline{(\mathsf{x_i} : \tau_i)} \rightarrow \{\mathsf{v} : \mathsf{t} | \phi\}, k$⟩)
(5) if depth ($\mathcal{A}$) < $k$ then
(6)     $\mathcal{A} \leftarrow$ Transition ($\mathcal{F}, \mathcal{A}$);
(7)     $\mathcal{A} \leftarrow$ Prune ($\mathcal{A}$);
(8)     $\mathcal{E} \leftarrow$ Similarity ($\mathcal{A}, \mathcal{E}$);
(9)     ($\mathcal{A}_{\text{min}}, \mathcal{E}$) $\leftarrow$ Minimize ($\mathcal{A}, \mathcal{E}$);
(10)    if $\hat{Q}$ = NEmpty ($\mathcal{A}_{\text{min}}$) then
(11)        return ($\mathcal{A}_{\text{min}}, \bigcup_{q \in \hat{Q}} [\![\mathcal{A}_q]\!]$)
(12)    Enumerate ($\mathcal{A}_{\text{min}}, \varphi, k$)
    else
(13)    return $\perp$
NEmptyQTA(⟨$\mathcal{A}$⟩)
(14) $\hat{Q} \leftarrow \{q_f \mid q_f \in Q_f, [\![q_f]\!] \neq \emptyset\}$
(15) return $\hat{Q}$
```

Algorithm 1: Main Synthesis Algorithm.

Next, at line 2, the algorithm checks if the language of the initial QTA $\mathcal{A}_0$ is non-empty using another routine NEmpty (lines 14 and 15), that collects all final states with a non-empty language using the earlier defined $[\![.]\!]$ function.

If this set is non-empty, the algorithm extracts the set of solution terms in the language of $\mathcal{A}_0$ using the QTA denotation $[\![\ ]\!]$ (Figure 8), finally returning $(\mathcal{A}_0, [\![\mathcal{A}_0]\!])$ (line 3). Otherwise (line 4), it calls procedure Enumerate (lines 5-13) that is the main *explore-reduce-check* loop where the construction (and reduction) of the QTA occurs. This procedure returns $\perp$, representing synthesis failure, if it is invoked when the max depth of the QTA has already been reached.

However, if the depth of $\mathcal{A}$ is less than the max-depth k, enumerate enters the exploration phase and expands $\mathcal{A}$ by adding new transitions, using a procedure Transition (line 6). This is a deterministic implementation of the transition rules in Figure 9 for capturing typing semantics for the λ_{qta}. These new transitions update the QTA thus adding larger terms.

At this point, Enumerate does not further expand $\mathcal{A}$; instead, the algorithm enters a *reduction* phase. It first reduces the size of $\mathcal{A}$, using the Prune function (line 7); we present its definition in Section 6. Next, the procedure performs similarity checking and reduction using the Similarity (line 8) and Minimize (line 9) functions. Similarity is a deterministic implementation of the rules given in Figure 10. It returns an updated similarity set $\mathcal{E}$ carrying all the similarity information in the structures. Minimize (line 9) is an implementation of the minimization inference rules (M-Trans) and (M-QTA) given in Figure 10. We describe these rules in the next section. In the process, the new QTA may drop certain states and transitions that are logically equivalent to more specific states or may merge several transitions. The result is a minimized automata $\mathcal{A}_{\mathsf{min}}$. Finally, ENUMERATE again checks (line 10) if the language of QTA $\mathcal{A}_{\mathsf{min}}$ is non-empty and returns the QTA along with the set of the extracted terms in $\mathbb{L}\ (\mathcal{A}_{\mathsf{min}})$ (line 11) using the denotation $[\![.]\!]\ \mathcal{A}_{\mathsf{min}}$. Otherwise, the algorithm iterates again on the minimized automata entering the *exploration phase.*

5.3 QTA Construction

Well-formedness $\boxed{\mathcal{F}, \mathcal{A} \vdash^{\mathsf{wf}} f \in \mathcal{F}(\overline{q_i}) \overset{\psi}{\hookrightarrow} q}$

$$\textsc{wf-prim}\ \frac{\mathsf{t} \in \mathsf{T}_{\mathcal{F}} \qquad q_{\mathsf{t}} \notin Q}{\mathcal{F}, \mathcal{A} \vdash^{\mathsf{wf}} \mathsf{t}() \hookrightarrow q_{\mathsf{t}}} \qquad \textsc{wf-pred}\ \frac{\phi \in \Phi_{\mathcal{F}} \qquad q_\phi \notin Q}{\mathcal{F}, \mathcal{A} \vdash^{\mathsf{wf}} \phi \hookrightarrow q_\phi} \qquad \textsc{wf-var}\ \frac{x \in \mathsf{Vars}_{\mathcal{F}} \qquad q_x \notin Q}{\mathcal{F}, \mathcal{A} \vdash^{\mathsf{wf}} x \hookrightarrow q_x}$$

$$\textsc{wf-base}\ \frac{(\tau \equiv \{x : t \mid \phi\}) \in \tau_{\mathcal{F}} \qquad q_\tau \notin Q}{\mathcal{F}, \mathcal{A} \vdash^{\mathsf{wf}} \tau(q_x, q_t, q_\phi) \hookrightarrow q_\tau} \qquad \textsc{wf-arrow}\ \frac{\begin{array}{c}(\tau_\rightarrow \equiv \tau_i \rightarrow \tau_j) \in \tau_{\mathcal{F}} \\ \tau_i, \tau_j \in \tau_{\mathcal{F}} \qquad q_{\tau_\rightarrow} \notin Q\end{array}}{\mathcal{F}, \mathcal{A} \vdash^{\mathsf{wf}} \tau_\rightarrow(q_{\tau_i}, q_{\tau_j}) \hookrightarrow q_{\tau_\rightarrow}}$$

$$\textsc{wf-t-abs}\ \frac{\begin{array}{c}q_\alpha, q_\tau \in Q \\ \psi = q_{\mathsf{tabs}} \blacktriangleright \mathsf{tvar.type} = q_{\mathsf{tabs}} \blacktriangleright \mathsf{type.base}\end{array}}{\mathcal{F}, \mathcal{A} \vdash^{\mathsf{wf}} \mathsf{tabs}(q_\alpha, q_\tau) \overset{\psi}{\hookrightarrow} q_{\mathsf{tabs}}} \qquad \textsc{Q-goal}\ \frac{\begin{array}{c}\varphi = \overline{(\mathsf{x_i} : \tau_i)} \rightarrow \tau \qquad q_{\mathsf{term}_k}, q_\tau \in Q \\ q_{\mathsf{goal}} \in Q_f \\ \psi = \textsc{SubType}(q_{\mathsf{term}_k} \blacktriangleright \mathsf{type}, q_{\mathsf{goal}} \blacktriangleright \mathsf{type})\end{array}}{\mathcal{F}, \mathcal{A} \vdash^{\mathsf{wf}} \mathsf{goal}(q_\tau, q_{\mathsf{term}_k}) \overset{\psi}{\hookrightarrow} q_{\mathsf{goal}}}$$

Transitions $\boxed{\mathcal{F}, \mathcal{A} \vdash f \in \mathcal{F}(\overline{q_i}) \overset{\psi}{\hookrightarrow} q}$

$$\textsc{e-var}\ \frac{x : \tau \in \mathcal{F} \qquad q_x \notin Q}{\mathcal{F}, \mathcal{A} \vdash x(q_\tau) \hookrightarrow q_x} \qquad \textsc{e-const}\ \frac{\vdash c : \tau \in \mathcal{F} \qquad q_c \notin Q}{\mathcal{F}, \mathcal{A} \vdash c(q_\tau) \hookrightarrow q_c}$$

$$\textsc{e-app}\ \frac{\begin{array}{c}q_f, q_a \in Q \qquad \tau(\overline{q_i}) \hookrightarrow q_\tau \in \Delta \\ \psi = \textsc{SubType}(q_f \blacktriangleright \mathsf{out}, q_{\mathsf{app}} \blacktriangleright \mathsf{type}) \wedge \\ \textsc{SubType}(q_a \blacktriangleright \mathsf{type}, q_f \blacktriangleright \mathsf{in})\end{array}}{\mathcal{F}, \mathcal{A} \vdash \mathsf{app}\ (q_\tau, q_f, q_a) \overset{\psi}{\hookrightarrow} q_{\mathsf{app}}} \qquad \textsc{e-if}\ \frac{\begin{array}{c}q_b, q_t, q_f \in Q \qquad \tau(\overline{q_i}) \hookrightarrow q_\tau \in \Delta \\ \psi = ((q_b \blacktriangleright \mathsf{ref}) \wedge \textsc{SubType}(q_t \blacktriangleright \mathsf{type}, q_{\mathsf{if}} \blacktriangleright \mathsf{type})) \wedge \\ (\neg(q_b \blacktriangleright \mathsf{ref}) \wedge \textsc{SubType}(q_f \blacktriangleright \mathsf{type}, q_{\mathsf{if}} \blacktriangleright \mathsf{type}))\end{array}}{\mathcal{F}, \mathcal{A} \vdash \mathsf{if}\ (q_\tau, q_b, q_t, q_f) \overset{\psi}{\hookrightarrow} q_{\mathsf{if}}}$$

$$\textsc{SubType}(\delta_i, \delta_j) = \begin{cases} (\tau_i(\overline{q_i}) \overset{\psi_i}{\hookrightarrow} q_{\tau_i}, \tau_j(\overline{q_j}) \overset{\psi_j}{\hookrightarrow} q_{\tau_j}) & \mathsf{i.type.t} = \mathsf{j.type.t} \\ & \wedge\ \mathsf{i.type.ref} \vDash \mathsf{j.type.ref} \\ (\tau_{\rightarrow_i}(\overline{q_i}) \overset{\psi_i}{\hookrightarrow} q_{\tau_{\rightarrow_i}}, \tau_{\rightarrow_j}(\overline{q_j}) \overset{\psi_j}{\hookrightarrow} q_{\tau_{\rightarrow_j}}) & \textsc{SubType}(\delta_j \blacktriangleright \mathsf{in}, \delta_i \blacktriangleright \mathsf{in}) \\ & \wedge\ \textsc{SubType}(\delta_i \blacktriangleright \mathsf{out}, \delta_j \blacktriangleright \mathsf{out}) \\ (\_, \_) & \mathsf{true} \end{cases} \tag{1}$$

Fig. 9. Selected rules for constructing transitions Δ, basis for WF and TRANSITION.

The transitions Δ for a QTA $\mathcal{A}$ are constructed using the **Well-formedness** and **Transitions** judgments given in Figure 9. The latter judgment holds if, given library $\mathcal{F}$ and automata $\mathcal{A}$, a new n-ary transition can be added to $\mathcal{A}$ corresponding to an n-ary symbol $f \in \mathcal{F}$, with $q_1, q_2, ...q_n$ being the incoming states in the transition and q being the target state, such that ψ captures the typing constraints for the valid terms in the language of the transition $[\![.]\!]$

5.3.1 Well-formedness Rules. Given the current alphabet $\mathcal{F}$, which contains the component-library and the query φ, a current QTA $\mathcal{A}$; and the well-formedness typing semantics in λ_{qta} (represented using $\vdash^{\mathsf{wf}}$), judgments of the form $\mathcal{F}, \mathcal{A} \vdash^{\mathsf{wf}} f \in \mathcal{F}(\overline{q_i}) \overset{\psi}{\hookrightarrow} q$ direct how we can add a new transition to $\mathcal{A}$. These rules capture the conditions under which (leaf) transitions can be added to well-formed base types t, predicates ϕ, variables, base and arrow refinement types, and polymorphic types. These rules closely follow the well-formedness typing rules for λ_{qta}.

Rule WF-PRIM adds transitions corresponding to each primitive type t in the library. For each such t, the rules creates a new state q_{t} and adds a nullary transition with symbol t and q_{t} as the target state. Rule WF-PRED similarly adds leaf transitions for refinement formula ϕ in some annotation in $\mathcal{F}$ or in the query. Rule WF-VAR, adds transition for variables in the library or query.

Rule WF-BASE, picks each well-formed base refinement type τ in the library or query annotation and creates a new state (q_τ) for this type and adds a transition for τ with three incoming states, corresponding to the three elements in a base refinement $\{ x : t \mid \phi \}$. Namely, q_x for the bound-variable x, q_t for the base-type t and q_ϕ for the refinement formula ϕ. Note that the rule builds these states constructed by earlier rules WF-VAR, WF-PRIM and WF-PRED as defined above.

Rule WF-ARROW generates similar transitions for each arrow refinement type, with a symbol $\tau_{\rightarrow}$ and two incoming states for the argument-type and the result-type for the arrow. Rule Q-GOAL adds a special goal transition and a final state q_{goal} for the given query and sub-automata rooted at q_{term_k} for all terms of size k. The state q_τ captures the return type for φ. The transition constraint ψ captures the standard subtype check between the type of the synthesized terms and the qoal's annotated return type, using a constrain generation function SubType.

Our extended λ_{qta} and typing semantics also include *type* and *refinement* abstractions, allowing parametric polymorphism in the refinement types setting [30?]. Consequently, the above rules also extend naturally to these abstractions and their corresponding type and refinement applications. To illustrate, Rule WF-T-ABS gives the rule to construct a transition representing a polymorphic type $\forall\alpha.\tau$, using the well-formedness semantics for type abstractions. Given two states, q_α for the type-variable and q_τ for the type with α occurring free, it constructs a transition representing the polymorphic type $\forall\alpha.\tau$.[4]

5.3.2 Expression Transition Rules. Transition judgments are similar in structure to the Well-formedness judgments, but simulate the refinement type judgments for the expressions in the λ_{qta}. These rules say how to add transitions corresponding to λ_{qta} expressions along with their types. Each n-ary expression transition has n+1 incoming states, with the state at the position zero capturing the sub-automata for the possible types of the expression.

Rule E-VAR and E-CONST adds transitions for variables and constants in the library along with their types. Note that E-VAR rule will add transitions for both scalars as well as variables bound to functions in the library, consequently, the type τ in Rule E-VAR will be modeled either as a base refinement (via WF-BASE) or an arrow refinement (via WF-ARROW).

Rule (E-APP) adds transitions related to function applications. It assumes the states q_f and q_a are already present for the function f and argument a. It first constructs a transition corresponding to the type (τ) of the function-application using the well-formedness judgments with q_τ as the target state. The inferred transition uses app as the symbol, with three incoming states, corresponding to the resulting type (q_τ), function q_f and argument q_a. The point to note in this rule is that constraints ψ added to the transition, relating the three sub-automata rooted at these states. These constraints use an auxiliary function SubType (Equation 1), which given two transitions, returns constraints sufficient to capture the subtype relation between their types. For example, ψ in E-APP captures

[4]Please see the supplemental material for a complete description.

two main relations. The first is a subtyping constraint between a function's formal argument and the actual expression passed as the argument ($q_{\mathsf{a}} \blacktriangleright \mathsf{type}, q_f \blacktriangleright \mathsf{in}$). This specifies that given a state q_{a}, ($q_{\mathsf{a}} \blacktriangleright \mathsf{type}$) represents the transition at position **type** from q_{a}. The second subtype relation is between the function's result type and the type of the expression ($q_f \blacktriangleright \mathsf{out}, q_{\mathsf{app}} \blacktriangleright \mathsf{type}$). The details of the auxiliary operation SubType are given in Equation 1. Note that the E-App rule is sufficiently general to allow synthesis to support *partial* (i.e., higher-order) function applications.

Rule E-if builds a transition for a conditional expression using the sub-automata for the Boolean condition (rooted at state q_b), and the true and the false branches, rooted resp. at q_t and q_f. The constraints on the transition captures the standard refinement typing semantics for conditional expressions. The true branch adds the constraints that the Boolean condition is true while the false branch specifies the negation. [5]

6 QTA REDUCTIONS

6.1 Prune

The QTA formulation in Section 3 accepts only well-typed terms from the λ_{qta} . However, we can make the synthesis procedure more efficient by eagerly reducing portions of the automata (i.e. sub-automata) which are irrelevant to the construction of any solution.

The inference rules for pruning irrelevant code, which form the basis for the Prune routine in Algorithm 1, are given as judgments in Figure 10. These rules have two judgment forms, $\mathcal{A} \vdash \Delta \rightsquigarrow \Delta'$ takes the current transition set and reduces it to a pruned set. The other judgment form, $\mathcal{A} \vdash \delta \rightsquigarrow^{\psi_a} \delta'$ reduces an individual transition by an atomic constraint ψ_a giving a pruned transition.

The p-trans rule takes a transition δ with transition constraint ψ. The rule assumes ψ as a conjunction of atomic constraints ψ_j and reduces the chosen transition δ by each ψ_j (defined next) and updates the original $\delta \in \Delta$ with the reduced δ_{r}.

Reduction of a transition by an atomic constraint is defined in the remaining two rules, matching the shape of the atomic constraint. Rule p-syn-eq handles syntactic equality constraints over positions p_1 and p_2. It performs a syntactic intersection [6, 22] over the two transitions at these position. Syntactic intersection ($\sqcap_{\mathsf{Syntax}}$) is a standard tree intersection operation. Intuitively, it compares the two transitions for syntactic equality of transition symbols while recursively intersecting each incoming state in the transition.

Rule p-sym-ent handles the alternate case, when ψ_j expresses a semantic entailment. When the constraint is $p_1 \vDash p_2$, standard equality based intersection is ineffective, as the formulas at positions p_1 and p_2 cannot be compared syntactically. Thus, we define a semantic intersection operation $\sqcap_{\mathsf{Semantics}}$, which only compares transitions having refinement qualifiers and compares them logically, checking the logical entailment of the formula at position p_2 by the formula at position p_1, keeping the transition at lower position p_1. Intuitively, this operation compares transitions modeling refinements of two types, and keeps the sub-type transition if the logical constraint hold, i.e. $\delta_{\mathsf{r}} = \delta \blacktriangleright p_1$, if the formula at $\delta \blacktriangleright p_2$ implies the formula at $\delta \blacktriangleright p_1$. Otherwise, it returns a special bottom transition, i.e. $\delta_{\mathsf{r}} = \delta_{\perp}$. The operation also runs a normalization routine and trims all $\perp$ transitions. Combined, these rules return a Δ with transitions reduced to $\delta_{\perp}$, forming the basis of the Prune procedure in QTASynthesize.

[5]The complete set of typing judgments, construction rules and example showing the QTA construction using these rules are provided in the supplemental material.

Pruning $\boxed{\mathcal{A} \vdash \Delta \rightsquigarrow \Delta' \quad | \quad \mathcal{A} \vdash \delta \rightsquigarrow^{\psi_a} \delta'}$

$$\text{P-TRANS}\ \frac{\delta \in \Delta \equiv f_i(q_{i1}, q_{i2}, ...q_{in}) \overset{\psi}{\hookrightarrow} q \quad \psi \equiv \bigwedge_{j\in[1...m]} \psi_i \quad \mathcal{A} \vdash \delta \rightsquigarrow^{\psi_j} \delta_\mathrm{r}}{\mathcal{A} \vdash \Delta \rightsquigarrow \Delta[\delta_\mathrm{r}/\delta]} \qquad \frac{\psi_j \equiv p_1 = p_2 \quad \delta_\mathrm{r} = \sqcap_{\mathrm{Syntax}}(\delta \blacktriangleright p_1, \delta \blacktriangleright p_2)}{\mathcal{A} \vdash \delta \rightsquigarrow^{\psi_j} \delta_\mathrm{r}}\ \text{P-SYN-EQ}$$

$$\frac{\psi_j \equiv p_1 \vDash p_2 \quad \delta_\mathrm{r} = \sqcap_{\mathrm{Semantics}}(\delta \blacktriangleright p_1, \delta \blacktriangleright p_2)}{\mathcal{A} \vdash \delta \rightsquigarrow^{\psi_j} \delta_\mathrm{r}}\ \text{P-SEM-ENT}$$

Similarity $\boxed{\mathcal{A} \vdash^{\mathbf{sim}} \delta_i \lesssim \delta_j \quad | \quad \mathcal{A} \vdash \mathcal{E} \rightsquigarrow \mathcal{E}'}$

$$\text{S-TRANS}\ \frac{\psi_{<:} = \textsc{SubType}(\delta_i \blacktriangleright \mathsf{type}, \delta_j \blacktriangleright \mathsf{type}) \quad \delta_i \blacktriangleright \mathsf{type} \sqcap_{\psi_{<:}} \delta_j \blacktriangleright \mathsf{type} \neq \delta_\perp}{\mathcal{A} \vdash^{\mathbf{sim}} \delta_i \lesssim \delta_j} \qquad \text{S-EQ}\ \frac{(\delta_i, \delta_j) \notin \mathcal{E} \quad \mathcal{A} \vdash^{\mathbf{sim}} \delta_i \lesssim \delta_j}{\mathcal{A} \vdash \mathcal{E} \rightsquigarrow \mathcal{E} \cup \{(\delta_i, \delta_j)\}}$$

Minimization $\boxed{(\mathcal{A}, \mathcal{E}) \vdash \Delta \rightsquigarrow \Delta' \quad | \quad \vdash (\mathcal{A}, \mathcal{E}) \rightsquigarrow (\mathcal{A}', \mathcal{E}')}$

$$\text{M-TRANS}\ \frac{\begin{array}{c} \delta_i, \delta_j \in \Delta \qquad (\delta_i, \delta_j) \in \mathcal{E} \\ \delta_i \equiv f(q_1, q_2, \ldots q_j \ldots q_n) \overset{\psi_i}{\hookrightarrow} q_i \\ \delta_j \equiv f'(q'_1, q'_2, \ldots q'_m) \overset{\psi_j}{\hookrightarrow} q_j \\ \Delta' = \bigcup_k \{\delta_k[q_j \mapsto q_i] \\ \mid \delta_k = \hat{f}(\hat{q_1}, \mathbf{q_j}, \ldots \hat{q_m}) \hookrightarrow \hat{q_k}\} \end{array}}{(\mathcal{A}, \mathcal{E}) \vdash \Delta \rightsquigarrow (\Delta \cup \Delta') \setminus \{\delta_j\}} \qquad \frac{\mathcal{A} \equiv (Q, \mathcal{F}, Q_f, \Delta) \quad (\mathcal{A}, \mathcal{E}) \vdash \Delta \rightsquigarrow^\star \Delta'}{\vdash (\mathcal{A}, \mathcal{E}) \rightsquigarrow ((Q, \mathcal{F}, Q_f, \Delta'), \varnothing)}\ \text{M-QTA}$$

Fig. 10. Selective rules Similarity inference and QTA Minimization.

6.2 Similarity and Minimize

The similarity inference rules are given in the Similarity judgments in Figure 10. The S-trans rules suggests when two transitions are *similar* based on their associated type sub-automata.

The crux of the definition rests on the constraint $\psi_{<:} = (\textsc{SubType}(\delta_i \blacktriangleright \mathsf{type}, \delta_j \blacktriangleright \mathsf{type})$. This constraint checks if the transition at position **type** for δ_i and **type** for δ_j are related by the standard subtype checks in a λ_{qta} typing semantics. Rule (S-eq) picks transition pairs from $\mathcal{A}$ not in $\mathcal{E}$ and checks their similarity, adding the pair to the similarity set $\mathcal{E}$ if the similarity relation holds.

Similarity Reduction: The above definition of similarity between transitions gives us an algorithmic way to minimize a QTA, reducing all similar transitions but one. The **Minimization** rules in Figure 10 define how to minimize a given QTA using the similarity relation between transitions. The rules have two judgment forms; $(\mathcal{A}, \mathcal{E}) \vdash \Delta \rightsquigarrow \Delta'$, which takes the original QTA $\mathcal{A}$ along with the similarity relation $\mathcal{E}$ and translates the original transition set to a new one, dropping the transition(s) with the super-type while keeping the subtype transitions; this is represented in Rule M-trans. It also updates the transition set Δ such that whenever the state q_j from the supertype transition, δ_j flows in originally, in $\mathcal{A}$, the rule now adds and incoming edge from q_i.

The second judgment defines how the complete automata is minimized and the similarity set is updated on minimization. This is shown in the M-qta rule, which creates a minimized transition set Δ' using the transitive closure of the transition update rules ($\Delta \hookrightarrow \Delta'$) defined by ($\hookrightarrow^\star$). The updated QTA symbol set $\mathcal{F}$ remains the same as the original. Finally, the updated similarity set becomes empty since all similar states are either deleted or merged with other states.

Note that the equivalence of terms (and hence sub-automata) is a stronger property than similarity. Consequently, similarity inference and reduction also reduces any equivalent sub-automata, thus allowing us to prune away both kind of redundant terms (see Figure 5) while giving an efficient enumeration in a reduced search space.

7 SOUNDNESS AND COMPLETENESS

For a given upper bound k on the size of programs being synthesized, the QTASynthesize algorithm is both sound and complete assuming the validity of each library function against their specifications.[6]

Theorem 7.1 (Soundness). *Given a type environment Γ that relates library functions $f_i = \lambda(\overline{x_{i,j}}).e_{f_i}$ with their refinement types $f_i : \overline{(x_{i,j} : \tau_{i,j})} \to \{v : t_i \mid \phi_i\} \in \Gamma$, and a synthesis query $\varphi = \overline{(y_i : \tau_i)} \to \{v : t \mid \phi\}$, if* QTASynthesize $(\Gamma, \varphi, \mathsf{k})$ *=($\mathcal{A}_{\mathsf{min}}$,* Terms $= \{e \mid e \in [\![\mathcal{A}_{\mathsf{min}}]\!]\}$*), then $\forall e \in$* Terms*, $\Gamma \vdash e : \varphi$, where Γ is consistent with $\mathcal{A}_{\mathsf{min}}$.*

Theorem 7.2 (Completeness). *Given a type environment Γ that relates library functions $f_i = \lambda(\overline{x_{i,j}}).e_{f_i}$ with their refinement types $f_i : \overline{(x_{i,j} : \tau_{i,j})} \to \{v : t_i \mid \phi_i\} \in \Gamma$, and a synthesis query $\varphi = \overline{(y_i : \tau_i)} \to \{v : t \mid \phi\}$, if* QTASynthesize $(\Gamma, \varphi, \mathsf{k})$*, $= \bot$, then $\nexists$ a term $e \in [\![\mathcal{A}_{\mathsf{complete}}]\!]$ containing fewer than $k + 1$ library function calls, such that $\Gamma \vdash e : \varphi$ and Γ is consistent with $\mathcal{A}_{\mathsf{complete}}$. Where $\mathcal{A}_{\mathsf{complete}}$ is the complete QTA of size k, for the given Γ, generated without any reduction.*

8 SYNTHESIS DETAILS AND IMPLEMENTATION

In the QTA construction and pruning rules and the QTASynthesize algorithm above, for ease of illustration, we have abstracted away several details about how we maintain variable scoping, infer types for transitions, and do efficient term extraction. Below, we discuss some of these in detail. For illustration, we will consider a library $\mathcal{F}$ = [f : (n : int) $\to$ (l : { v : [a] | len (v) > 0 }) $\to$ { v : [a] | len(v) = len(l) }; xs : { v : [int] | len (v) = 1} ; ys : [char]; g : (l : [char]) $\to$ [char]]. Figure 11 shows a portion of minimized QTA for terms of size two with transition $\boxed{\text{app}}$ for this library.

8.1 Variable Scoping and Typing Environment in QTA

The details of variable scoping during QTA construction and pruning are important to understand how enumeration works with refinement types in a QTA. To simplify scoping decisions and manage the typing environment, we made several design choices. First, we require terms in our synthesis language λ_{qta} to be in A-normal form, and to have a unique binding variable t_i for each application and conditional term. We also refactor each library function specification, alpha-renaming all bounded variables in the arguments with unique argument(s). This allows us to avoid unwanted variable capture across library functions without explicitly keeping track of scope information about bound and free variables.

Additionally, to build the typing environment, each term binding variable in ANF is ascribed a set of possible types that can be associated with it. To implement this structurally in QTA, we extend each n-ary expression transition (e.g.app transition) in the QTA, to (n+1) arity with an additional incoming edge for the type of the resulting expression (e.g., function application term). See, for example, the arrows with label type in Figure 11 for the app transition that has an edge ($q_\tau \to \boxed{\text{app}}$), q_τ to

[6]Proofs can be found in the supplemental material.

represent a set of valid types that can be ascribed to the application term (described next). Finally, we build a global typing environment mapping each expression type (annotated and inferred) with the binding variable using a typing environment construction function. This function is derived from a relation relating QTAs to typing environments; details are provided in the supplemental material.

8.2 Resolving Refinement Predicates for type Edges in Transitions

QTA construction and pruning also rely on inferring a set of feasible types for each transition. For instance see the incoming state ($q_\tau \rightarrow \boxed{\text{app}}$) in the example. The possible set of types is shown using a transition ($\boxed{\{v : \alpha \mid \phi_1\}} \rightarrow$ q_τ).

Inferring this type set precisely requires inferring the base type (shown by a type variable α in the example) and the refinement predicate (here ϕ_1). The former is straightforward, given the transition constraints, using syntactic comparison between terms at constrained location, e.g., the given constraint relating function and argument type structure allows us to infer in this case that the type of the application term is $[t]$, a list over some base type t. Inferring the refinement predicates is more convoluted and needs some elucidation. We assign abstract predicates to each missing refinement, e.g. ϕ_1 in the example, for the refinement predicate. The logical constraint over transitions allows us to maintain logical *implication* relations over these abstract predicates with other similarly constrained predicates.

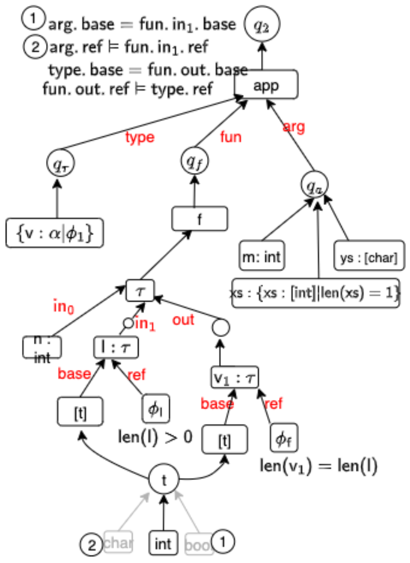


Fig. 11. Partial QTA for the example library, variables renamed to avoid variable capture.

For instance, when inferring the type for the term let t_1 = f m x (under a typing environment constructed over the refactored QTA as explained earlier), we have two *implication* constraints (i) len (v_1) = lenl $\implies \phi_1$ and (ii) len(xs) = 1 $\implies$ true. When solving for ϕ_1, in the context, we can unify l with xs, and with v_1 with t_1 to precisely ascribe/infer ϕ_1 as len (t_1) = len(xs) $\wedge$ len(xs) = 1 which reduces to len (t_1) = 1. This satisfies both constraints (i) and (ii). We can generally have a set of such ascribed types for cases with multiple functions and arguments in an app transition.

8.3 QTA Term Extraction

Once a minimized QTA $\mathcal{A}_{\text{min}}$ is constructed for a given size k, library $\mathcal{F}$, and the given ϕ, Algorithm QTASynthesize checks for the non-emptiness of the QTA and if successful, returns the QTA and the terms in the language of the QTA. The QTA denotation function call, in the Algorithm, is defined in Figure 8 and gives a naive term extraction strategy where: a) we enumerate all terms based on underlying unconstrained tree-automata, b) filter out those terms which violate the constraints of each transition. Unfortunately, even on minimized QTA, this will face a challenge in scaling to interesting scenarios. To address this, our implementation uses a more clever strategy employing a hierarchical combination of; a) a lazy choice of concrete types for polymorphic type variables, based on equality constraints [22]. This allows an efficient enumeration of valid base-typed terms. However, these well base-typed terms may still contain ill-typed terms in the refinement-type world. To filter out these terms: b) we have a secondary enumeration (using the denotation function) over these valid, based-typed terms, discarding terms that do not satisfy the logical constraints.

Figure 11 shows $\boxed{\mathsf{app}}$ containing both equality (①) and logical constraints (②). While enumerating monomorphic type for f, our strategy first uses ① to deduce that that the type variable t cannot reduce to the base type bool as there are no argument under $\textcircled{q_a}$ of [bool]. This is shown by a greyed out transition ($\boxed{\mathsf{bool}} \rightarrow \textcircled{t}$). We next enumerate the remaining terms using the $[\![.]\!]$ function, this further filters out the choice for t to be char, as type for ys does not satisfy ②, shown by a corresponding greyed out transition ($\boxed{\mathsf{char}} \rightarrow \textcircled{t}$). This leaves only a single enumeration choice for t as int, giving a unique monomorphic type for f.

8.4 Implementation

We have implemented these ideas in a tool (Hegel) that comprises approximately 5KLoC in OCaml. The input to Hegel is a specification file containing a library of functions and data constructors, along with their specifications, followed by a goal query. The specification and query languages are type-based, using refinement types for pure functions [21], extended with support for polymorphic type specifications. We rely on OCaml lexing and parsing libraries OCamllex [23] for handling the front end of our query specification language. We use Z3 [8] to discharge SMT queries.

Hegel Library Specifications. Hegel performs component-based synthesis using libraries annotated with refinement-type specifications. Fortunately, there are multiple open-source projects that provide such specifications for our use [5, 21, 22, 25]. Our experiments adapt approximately 300 refinement type-annotated library functions drawn from these projects. These functions span operations on data structures (e.g., arrays, lists, trees, queues, vectors, zippers, byte strings, etc). We additionally include approximately 40 more specialized functions that target a specific database application class [18]. Approximately, 25% of these library functions are higher-order.

9 EVALUATION

Our evaluation considers the following three research questions:

RQ1 **Effectiveness** of Hegel: How effectively can Hegel synthesize complex refinement type queries? How does Hegel compare against other specification-guided, component-based synthesis tools in its ability to synthesize programs when queries define fine-grained constraints?

RQ2 **Scaling** to query complexity: Complex queries correlate to the size and control-flow complexity in synthesized outputs. How effective is Hegel in synthesizing programs as query complexity scales?

RQ3 **Impact** of QTA reduction strategies on synthesis efficiency and quality: How significant are the benefits of pruning and similarity reductions in reducing QTA size and the search space over which the synthesis procedure operates?

9.1 Benchmarks

To effectively answer RQ[1-3], we required component-based synthesis benchmarks of the kind shown in Section 2. To this end, we have collected a set of Hoogle+ [19**?**] and Hectare [22] benchmarks originally used for (simple) type-guided synthesis over Haskell libraries, and re-implemented them in OCaml for use by Hegel. To avoid selection bias towards queries with only a particular kind of features (e.g., Hoogle+ queries are primarily first-order while Hectare mostly avoids first-order queries), we divided the original benchmarks from these two sources into three main categories: *standard* first-order queries primarily from Hoogle+; *higher-order* queries primarily from Hectare and *polymorphic* queries from both these sources.

To create realistic, refined queries from these, and compare Hegel against other tools, we refine each of these to include three different user-provided refinements. These refinements are carefully chosen so that they *invalidate* the synthesized output on the original (unrefined) query. We specify the intent of these refinements in the Hoogle+ runs using 4 I/O examples; Hectare does not accept query refinements or I/O examples and performs synthesis only based on the (non-refined) standard types given in the library. To make the comparison as fair as possible, we drop queries in our benchmark set that cannot either be easily refined using I/O examples, or where the refinement type specification was trivial compared to the I/O example. Our goal was to choose queries that collectively cover all three categories with non-trivial refined queries, and with a similar degree of complexity in both I/O and refinement type specifications. These requirements led us to select 14 queries across the three categories that we found to be amenable for such comparison. [7] We then compare the performance of Hoogle+, Synquid [26], a deductive synthesis tool that uses refinement types for its specification, and Hegel on 42 (14×3) different refined queries, making our comparison extensive. Table 12, shows these original queries type specifications along with description of each of the refined queries.

9.2 RQ1: Effectiveness and Comparison with Other tools

Results. The table in Figure 12 presents the main experimental results for RQ1. Tables in Fig 4 and 5 in [1] show a) the number of SMT calls, b) time spent for constraints solving, c) Number of original QTA states ($|Q|$), and d) QTA states after minimization ($|Q|$ min) for RQ1 and RQ2 benchmarks respectively. The first column gives a symbolic name for the refined benchmark derived from the original Hoogle+ and ECTA benchmarks. For example, the Nth1 benchmark requires a solution that performs a non-trivial swapping of the values at given indexes in the input list [8]. Several benchmarks also include higher-order functions; these are given as the last five ($\times$ three benchmarks) set of queries, marked with †.

The input to Hegel is a refinement-type query and a pre-annotated set of libraries. To quantify query specification burden, the next column in the table provides a measure of specification size in terms of the number of conjunctions($\wedge$) and disjunctions($\vee$). The average size of these specs is around four-five conjuncts making this overhead small, particularly when compared against size of the library, the complexity of the search, and the number of examples needed to capture these in i/o settings like Hegel. The next three columns show the results for running Hegel(He), Hoogle+(H+) and Synquid (Sn) on each benchmark. For each of these runs, we used a timeout limit of three minutes and maximum term size bound of five library function calls. All results were performed on a standard off-the-shelf laptop with 8GB of memory.

As the table shows, Hegel is able to solve all benchmarks under 11 seconds, with an average time of around 7.6 seconds. Hoogle+ (H+), was able to solve only a fraction (6/42) of the benchmarks, taking approximately **6x** more time to yield a solution than Hegel.

There is no data to report for Hectare because it does not support logical refinements (either as types or I/O examples) on queries. Column 'Sn' shows the synthesis time for Synquid, which solves around 20/42 of these benchmarks, while taking substantially more time, around 4.5x, on average. This is understandable, given that Synquid's goal is not efficient CBS and hence does not employ any efficient pruning or search mechanism.

[7] See supplemental material for a detailed description of our benchmark set, along with one example with explanation.
[8] See supplemental material for details on the original simply-typed queries and a description of the refinements for each benchmark

The next four columns provide statistics for the QTA reduction and constraint solving overhead in Hegel. These provide a) the number of SMT calls, b) time spent for constraint solving, c) the number of original QTA states (|Q|), and d) the number of QTA states remaining after minimization (|Q| min) for RQ1 and RQ2 benchmarks, resp. Overall, on average, Hegel makes approximately 30 SMT calls per benchmark while spending close to 3.8 seconds on average per benchmark; thus, approximately 40% of overall synthesis time is spent in constraint solving. The final column (#S) gives the size of Hegel's solution in terms of the number of library function calls in the synthesized result. These numbers, in most cases, differ from the solutions for the original query. In fact, in several cases the solutions also introduce new control-flow branches (2 branches in each case) that were absent in the original; we label these benchmarks with a $\star$.

Name	#$\wedge$/$\vee$	**Refined Time(s)/#**			**QTA & SMT Stats**				#S
		He	H+	Sn	#SMT	SMT Time	$\lvert Q\rvert$	$\lvert Q\rvert_{\text{min}}$	
Nth1	4	5.1	39.4	21.3	22	3.07	239	53	4
Nth2	3	7.6		22.7	25	3.40	199	56	5
Nth3	4	9.1			33	4.01	210	78	5
RevApp1	3	5.1	49.5		18	2.87	110	56	3
RevApp2	3	7.3			22	3.90	166	61	4
RevApp3	4	5.4			18	3.03	201	43	4
RevZip1	3	6.1	43.2	22.6	31	4.21	278	69	4
RevZip2	4	6.9		28.1	21	3.80	189	69	5
RevZip3	4	8.5			39	5.07	301	62	6
SplitAt1	4	6.8		32.1	19	2.10	156	47	5
SplitAt2	5	7.3		24.3	28	4.13	176	76	4
SplitAt3	4	7.8			25	4.40	173	54	5
Nth_Incr1	3	5.2	35.4	36.8	19	3.70	143	38	3
Nth_Incr2	4	7.4	56.3		37	5.12	187	87	5
Nth_Incr3	4	7.8		39.5	26	4.30	165	73	5
CEdge1	4	6.4		21.5	21	3.10	110	39	4
CEdge2	4	5.2		23.7	11	2.87	145	34	3
CEdge3	5	8.6			24	2.98	167	42	5
AppendN1	4	6.0	47.7		21	3.04	267	61	4
AppendN2	4	10.4			39	4.70	310	76	7*
AppendN3	4	7.5		25.8	31	4.60	234	57	5
SplitStr1	4	6.5			17	3.10	212	41	5
SplitStr2	4	5.2		36.2	18	1.90	245	63	5
SplitStr3	6	7.1			21	3.00	277	87	5
LookRange1	4	8.2		43.6	37	2.80	413	99	6*
LookRange2	6	9.1			65	3.30	512	114	7*
LookRange3	4	8.6			57	3.80	511	114	7*
Map1†	5	7.5			41	3.60	219	110	6
Map2†	6	5.2	45.1	34.1	25	2.20	198	55	4
Map3†	5	5.0	33.5	29.8	22	1.40	176	53	4
MapDouble1†	4	10.8			36	3.02	399	89	5
MapDouble2†	4	8.9			51	3.60	465	88	4
MapDouble3†	6	8.4		42.8	55	2.60	452	101	4
ApplyNAdd1†	6	7.9			27	3.10	259	76	5
ApplyNAdd2†	6	6.9	63.1	34.2	17	3.60	331	59	5
ApplyNAdd3†	6	9.1			34	5.20	323	79	6
ApplyNInv1†	5	9.5			34	4.80	299	83	5
ApplyNInv2†	5	7.6		32.4	11	6.30	134	38	5
ApplyNInv3†	6	8.1		55.6	13	4.80	156	41	5
ApplyList1†	4	12.3			65	7.50	519	132	8*
ApplyList2†	4	10.5			59	7.70	483	122	8*
ApplyList3†	5	7.9		42.6	43	4.10	277	75	6*

Fig. 12. Results for experiments with Refined Hoogle+ (H+) and ECTA benchmarks. The details of the original queries and a description of the refinements for each benchmark is given in the supplemental material.

Name	Desc.	#∧/∨	Results Hegel				QTA & SMT Stats				T(Sn)
			T(He)	#C	#B	#R	#SMT	SMT(s)	$\lvert Q\rvert$	$\lvert Q\rvert_{\text{min}}$	
NLInsert	Add a newsletter and user	6	22.7	16	2	31	110	11.12	779	212	126.2
NLRemove	Remove a newsletter and user	4	39.9	20	4	35	189	19.34	1201	372	_
NLR_Remove	Read articles list and remove	5	42.8	19	4	21	213	18.20	1331	381	_
NLInv	Remove with uniqueness invariant	8	52.5	25	4	36	154	24.19	1398	435	_
FWInsert	Insert a normal device	4	31.2	15	2	33	166	12.34	945	298	124.6
FWMkCentral	Insert a central device	4	65.2	33	4	53	259	27.13	1611	401	
FWInsConn	Insert a device connected to all	5	36.8	14	2	59	218	15.12	806	261	
FWInvert	Invert the connections	4	33.9	14	4	47	197	15.70	1176	323	
FWInvertDel	Delete, and invert connections	6	47.3	17	4	38	184	22.90	1352	421	

Fig. 13. Results for tailored specification-guided synthesis benchmarks, The #C and #B gives the total number of function calls and branches in the synthesized solution. #R gives the number of transitions Hegel merged during the Similarity reduction and Irrelevant code pruning phases.

9.3 RQ2, Scaling Hegel to larger and complex queries

To answer RQ2, we consider synthesis query benchmarks whose solutions require longer call sequences and more complex control flows than the queries given in the table shown in Figure 12. We have collected and refined eight queries adapted from verification benchmarks [18] that are particularly amenable for CBS techniques. Figure 13 lists these benchmarks, which are defined over two database applications [18]. We only show Hegel and Synquid results because Hoogle+ was not able to produce solutions within the timeout bound (3 minutes).

The first is a *Newsletter* database (queries with names NL…in the table) that defines a single table NS equipped with various attributes (e.g., *newsletter, user, subscribed, articles, code, clear_email, add_email*, etc.). For example, query NLRRemove is shown as a comment in Figure 14, it encodes the following problem: *Synthesize a function that returns a list of articles for a given newsletter (*n*) and a given user (*u*) in a database (*d*), along with an updated database that does contain* u *and* nl*, while keeping the email address if* u *has opted for promotions.* The second application implements a network firewall database (queries with names FW…in the table) that manages two tables, a table of devices and a table storing sender-receiver links. These benchmarks are adapted from their original definition to employ a monadic style that threads database state through calls.

Synthesizing programs from queries of this kind must take into account appropriate protocols associated with the libraries, e.g., to establish a connection to a device that is not currently in the device table requires that the device first be added. These constraints also require conditional-control flows in the solutions.

Results. The first two columns in Figure 13 give the name and a small high-level description of the benchmark. The next two columns 'He' gives the synthesis time in seconds. Hegel succeeds in finding solutions to all queries with a synthesis time in the range of 22.7 seconds to little more than a minute. Its ability to solve these queries can be attributed to an efficient exploration enumeration strategy that effectively reduced, either via Prune or Similarity reduction strategies, a large number of tranistions during synthesis. These numbers are given in the (#R) column and ranges from 21 in NLR_Remove, to as high as 59 transitions in FWInsConn. Note that the effective actual saving in enumeration is much more that this, as an exponential number of other transitions built over these are never constructed.

To quantify the complexity of the synthesized solutions, we also list the properties of the solutions synthesized in terms of the total number of function calls (i.e., the size of the solution) as well as solution complexity in terms of the number of control flow branches. For instance, given the query NLRRemove, the challenge is to synthesize a solution that maintains a specific contract associated with each library function; these include the requirement that a) the user must be unsubscribed before removal, b) if the user has not opted for *promotions*, the email for the user must be cleared, etc. Figure 14 shows the synthesized program generated by Hegel for this query. Note that the solution includes a total of 19 function calls and exhibits complex control flows (4 branches).

The next four columns show: a) the number of SMT calls, b) the time spent for constraint solving, c) the number of original QTA states (|Q|), and d) the number of QTA states remaining after minimization (|Q| min). The average number of QTA states produced is around 1200, while Hegel's pruning and minimization results in a roughly 70% reduction. The last column shows synthesis times for Synquid, which is the only other tool that can work with refined specifications. It produces results for only the smallest of the benchmarks, taking a little over two minutes on those, while timing-out on all other cases.

```
(*nLRRemove : (n : nl) -> (u : user) ->
  (d : {v: [nlrecord] | mem (v , n , u)}) ->
  {v : (  f : article * s : [nlrecord]) |
  mem (f, articles (s))
  ∧  ¬ nlmem (s, n, u)
  ∧ (promotions (s, u) => email (s, u))
  }*)
fun n u d ->
    let x = read (d, n, u) in
    let x0 = fst (x) in
    let d0 = snd (x) in
    let d1 = confirmU (d0, n, u) in
    let x1 = promotions (d1, n, u) in
    if (not (x1)) then
        let subscribed = subscribed (d1, n, u) in
        if (length (subscribed) > 0) then
            let d2 = clear_email (d1, n, u) in
            let d3 = unsubscribe (d2, n, u) in
            let d4 = remove (d3, n, u) in
            (x0, d4)
        else
            let d5 = unsubscribe (d1, n, u) in
            let d6 = remove (d5, n, u) in
            (x0, d6)

    else
        let d7 = unsubscribe (d1, n, u) in
        let d8 = remove (d7, n, u) in
        (x0, d8)
```

Fig. 14. Synthesized Program for NLR_Remove

9.4 RQ3: Impact of irrelevant code pruning and similarity reduction

Because RQ3 cuts across both set of benchmarks, we perform several ablation experiments over the queries described in the Figures 12 and 13. We create three variants of Hegel, *viz.* (i) Hegel(-P), a QTA-based synthesis implementation without the irrelevant code reduction (i.e. comment out the Prune call at line 7 in Algorithm 1), but retaining *similarity reduction*; (ii) Hegel(-S), a variant of Hegel with support for pruning but *without* similarity reduction (i.e., lines 8 and 9 in Algorithm 1) are commented out); and, (iii) Hegel(-All), a baseline variant that constructs the QTA without performing any reduction (i.e., removes lines 7-9 in the algorithm). We compare these variants in terms of two main metrics, overall *synthesis times* and the size of the search space in each case after the reduction, shown by *number of program terms enumerated* during search, compared to the base-line (Hegel(-All).

The first two charts in Figure 15 show results for overall average synthesis times across the two sets of benchmark queries described earlier. We note that both Hegel(-S), and Hegel(-P) can solve all queries from RQ1, but at a cost which is 2 - 3X greater than Hegel. Hegel(-All) on the other hand fails on almost half of the benchmarks. In contrast, although Hegel(-P) was also able to solve the full complement of queries studied in RQ2, it did so with a considerable larger overhead compared to Hegel, while here the the irrelevant code pruning (Hegel(-S)) alone is insufficient to scale the variant to these challenging benchmarks and it fails to solve 3/9 benchmarks. The second pair of charts and show the average number of terms enumerated by these different variants, showing the reduction of search space by each reduction strategy, with Hegel(-All) as the baseline. Here we see, with the combined reduction strategies, Hegel sees the maximum search space reduction, while the other two variants Hegel(-P) and Hegel(-S) having much larger search spaces, (anywhere from 2-4.5X more) without necessarily solving the same number of queries.

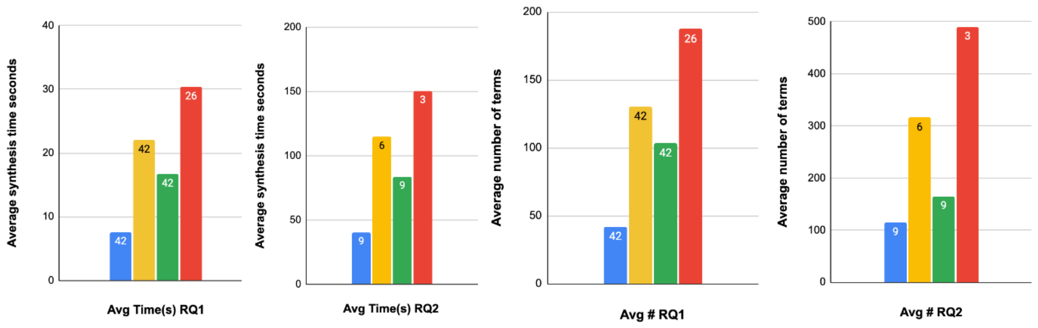


Fig. 15. Comparison of Hegel and its variants Hegel, Hegel(-S), Hegel(-P), Hegel(-ALL), on average synthesis times and the number of candidate terms generated on RQ1 and RQ2. The labels on each bar show the number of benchmarks solved by these variants out of 42 in the case of RQ1 and 9 in the case of RQ2.

10 RELATED WORK

Component-based Synthesis. There is a long line of work on the use of CBS in the context of domain-specific languages [10, 20] as well as general-purpose programming domains [9, 11, 16, 17, 25, 28, 31]. Despite various technical differences, these approaches all include some form of search at their core, which they tackle using basic type information [11, 16], or a combination of types and limited effects [17]. Our contributions in this paper extend prior work by enabling CBS to be applied when specifications and queries are equipped with logical refinements, substantially increasing the complexity of the search process. While some prior work [9, 25] also use logical properties to prune the feasible search space, they provide only limited benefits in the absence of a data structure like QTA with respect to query complexity scaling. For example, running [25] on the benchmarks in Figure 13 produces results similar to Hegel(-ALL).

Using similarity/equality information for search. Recent interest in utilizing equality information, and using ways to calculate equality saturation sets using e-graphs [33] allows efficient reduction of an enumeration space similar to the motivation underlying our approach. Equality saturation has been applied to enable efficient abstraction learning [4] inductive synthesis [3] and program analysis [36]. However, all these works depend crucially on fast calculation of the equality saturation set, making them suitable for syntactic similarity relations, but it is not obvious how to extend this technique to provide support for semantic (subtype-based) similarity checks as in our setting. Nonetheless, we leave as future work, questions related to how QTA-like structures can benefit from the general capabilities provided E-graphs. Another line of work in this vein [**?**] asks users to provide logical equivalences between operators that can be then used to accelerate the synthesis process. This requirement has utility when dealing with small DSLs where a user is likely to understand operator semantics and their equivalences, but we expect will be infeasible in a component-based synthesis setting that deals with libraries with a large number of methods.

Our idea of similarity reduction is also related to the notion of *observational equivalence* found in programming-by-example synthesis approaches [1, 12, 24]; these techniques compare synthesized programs on a given set of inputs and prune the search-space in a bottom-up, inductive synthesis setting. Its inherent unsoundness makes this mechanism infeasible for specification-guided synthesis.

Tree automata for program synthesis. As described earlier, prior work has leveraged tree automata to compactly capture the space of programs [13, 22, 32] in a synthesis setting; such automata have also been used in the synthesis of reactive systems [13]. In fact, recent work [22] has also proposed techniques to qualify the states of such automata, albeit with syntactic equality constraints [6, 7] to capture dependences between sub-spaces, allowing efficient reduction of the search space over base types. We see our contribution as a continuation of these efforts, generalizing them to handle richer semantic dependences expressed in specifications and queries.

Closely related to our work is Blaze [?] which introduces an abstract finite tree automata (AFTA) to help implement its synthesis procedure. AFTA's states, like QTA's, are also defined by a type plus a predicate. However, there are a number of important differences between the two approaches; first, unlike the qualifiers in QTA, which are refinement types, the abstract predicates in AFTA are formulas capturing an abstract domain [?]. The annotations on the states are much more complex in QTAs, expressed as quantified formulas over program variables; in contrast, AFTA's predicates are quantifier-free formulas over an abstract domain. QTA qualifiers, on the other hand, are used to navigate a compact representation of the search space and to help find similarities between subspaces in a refinement-type guided deductive synthesis procedure. Second, unlike AFTA's, QTAs also support function types, higher-order programs, and let bindings. These differences lead to substantially different synthesis algorithms in these two approaches. CTA [?] also adds constrains to traditional tree automata similar to how symbolic finite automata extends finite automata. Particularly, CTA allows transitions guarded by a logical formulas from a decidable theory, allowing them to effectively capture relational properties with deciadable acceptance checking. Unfortunately, these logical formulas cannot relate sub-automata, like in QTA or ECTA, thus, making them less effective to capture typing semantics. Allowing CTA-like constraints in QTA, however, may allow us to extend our synthesis approach to yield recursive and mutual recursive functions; we leave this as part of future work..

Refinement types and conflict-driven learning for synthesis. Several earlier works have used refinement types for program synthesis [22, 25, 26]. Although, Synquid's synthesis is also guided by Refinement Types, its goals differ from ours, and it does not aim towards efficient CBS in refinement-typed space. A family of works using CDCL may also be somewhat related, however, the CDCL avoidance [11, 25] of equivalent failing terms alone has limited pruning capabilities, as the synthesizer still has to explore a large number of terms that have neither failed (they may still lead to a correct solution when further explored) nor reached the maximum term-size depth. In such cases, a CDCL-based synthesizer is still forced to explore many equivalent (non-failed) terms; e.g, all the terms shown in blue, gray or yellow boxes in Figure 5 may still be generated using such techniques. Roughly speaking, the overall performance of such an approach is on par with Hegel(-S), the green bar in Figure 15.

11 CONCLUSIONS

This paper describes a new component-based synthesis algorithm and tool (Hegel) designed to operate on libraries and queries that are equipped with refinement-type specifications. These specifications can impose significant constraints on the set of feasible solutions making naïve enumeration of the search space impractical. We propose a new tree automata variant (QTA) to succinctly represent the search space in this setting, and propose a number of semantics-based optimizations to greatly reduce search overhead. allow efficient construction and enable semantic-based similarity checking among candidate terms to greatly reduce search overhead. Our experimental results demonstrate that Hegel is able to successfully synthesize correct outputs given complex query inputs over a range of application benchmarks that exceed the capabilities of existing systems.

DATA AVAILABILITY STATEMENT

Our supplementary material includes an anonymized artifact. This artifact contains the OCaml source code for Hegel and our suite of benchmark programs. We intend to submit this artifact with additional scripts to automatically generate the results for evaluation by the artifact evaluation committee should this paper be accepted.

REFERENCES

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 934–950.

[2] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2009. *Handbook of Satisfiability.* Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press.

[3] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. 2023. Top-Down Synthesis for Library Learning. *Proc. ACM Program. Lang.* 7, POPL, Article 41 (jan 2023), 32 pages. https://doi.org/10.1145/3571234

[4] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2023. Babble: Learning Better Abstractions with E-Graphs and Anti-Unification. *Proc. ACM Program. Lang.* 7, POPL, Article 14 (jan 2023), 29 pages. https://doi.org/10.1145/3571207

[5] Arthur Charguéraud, Jean-Christophe Filliâtre, Mário Pereira, and François Pottier. 2017. VOCAL – A Verified OCaml Library. ML Family Workshop.

[6] Hubert Comon. 1997. Tree automata techniques and applications. https://api.semanticscholar.org/CorpusID:2092186

[7] Max Dauchet, Anne-Cécile Caron, and Jean-Luc Coquidé. 1995. Automata for Reduction Properties Solving. *Journal of Symbolic Computation* 20, 2 (1995), 215–233. https://doi.org/10.1006/jsco.1995.1048

[8] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[9] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-Driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 420–435. https://doi.org/10.1145/3192366.3192382

[10] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 422–436. https://doi.org/10.1145/3062341.3062351

[11] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-Based Synthesis for Complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 599–612. https://doi.org/10.1145/3009837.3009851

[12] Jack Feser, Işıl Dillig, and Armando Solar-Lezama. 2023. Inductive Program Synthesis Guided by Observational Program Similarity. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 254 (oct 2023), 29 pages. https://doi.org/10.1145/3622830

[13] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2019. Synthesizing Functional Reactive Programs. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell* (Berlin, Germany) *(Haskell 2019)*. Association for Computing Machinery, New York, NY, USA, 162–175. https://doi.org/10.1145/3331545.3342601

[14] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

[15] David Furcy and Sven Koenig. 2005. Limited Discrepancy Beam Search. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence* (Edinburgh, Scotland) *(IJCAI'05)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 125–131.

[16] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program Synthesis by Type-Guided Abstraction Refinement. *Proc. ACM Program. Lang.* 4, POPL, Article 12 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371080

[17] Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2021. RbSyn: Type- and Effect-Guided Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3453483.3454048

[18] Shachar Itzhaky, Tomer Kotek, Noam Rinetzky, Mooly Sagiv, Orr Tamir, Helmut Veith, and Florian Zuleger. 2017. On the Automated Verification of Web Applications with Embedded SQL. In *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy (LIPIcs, Vol. 68)*, Michael Benedikt and Giorgio Orsi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:18. https://doi.org/10.4230/LIPIcs.ICDT.2017.16

[19] Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. 2020. Digging for Fold: Synthesis-Aided API Discovery for Haskell. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 205 (nov 2020), 27 pages. https://doi.org/10.1145/3428273

[20] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

[21] Ranjit Jhala and Niki Vazou. 2021. Refinement Types: A Tutorial. *Found. Trends Program. Lang.* 6, 3-4 (2021), 159–317. https://doi.org/10.1561/2500000032

[22] James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, and Nadia Polikarpova. 2022. Searching Entangled Program Spaces. *Proc. ACM Program. Lang.* 6, ICFP, Article 91 (aug 2022), 29 pages. https://doi.org/10.1145/3547622

[23] Xavier Leroy, Didier Rémy Alain Frisch, Jacques Garrigue, and Jérôme Vouillon. 2022. Parsing with Ocamllex. https://ocaml.org/manual/lexyacc.html

[24] Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. *Proc. ACM Program. Lang.* 6, POPL, Article 21 (jan 2022), 29 pages. https://doi.org/10.1145/3498682

[25] Ashish Mishra and Suresh Jagannathan. 2022. Specification-Guided Component-Based Synthesis from Effectful Libraries. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 147 (oct 2022), 30 pages. https://doi.org/10.1145/3563310

[26] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

[27] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

[28] Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: Component-Based Synthesis with Control Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 73 (jan 2019), 29 pages. https://doi.org/10.1145/3290386

[29] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying Higher-Order Programs with the Dijkstra Monad. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 387–398. https://doi.org/10.1145/2491956.2491978

[30] Niki Vazou, Alexander Bakst, and Ranjit Jhala. 2015. Bounded Refinement Types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 48–61. https://doi.org/10.1145/2784731.2784745

[31] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2019. Visualization by Example. *Proc. ACM Program. Lang.* 4, POPL, Article 49 (dec 2019), 28 pages. https://doi.org/10.1145/3371117

[32] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of Data Completion Scripts Using Finite Tree Automata. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 62 (oct 2017), 26 pages. https://doi.org/10.1145/3133886

[33] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (jan 2021), 29 pages. https://doi.org/10.1145/3434304

[34] Steven Wolfman, Pedro Domingos, and Daniel Weld. 2001. Programming By Demonstration Using Version Space Algebra. *Machine Learning* 53 (12 2001). https://doi.org/10.1023/A:1025671410623

[35] Yongwei Yuan, Arjun Radhakrishna, and Roopsha Samanta. 2023. Trace-Guided Inductive Synthesis of Recursive Functional Programs. 7, PLDI (2023). https://doi.org/10.1145/3591255

[36] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proc. ACM Program. Lang.* 7, PLDI, Article 125 (jun 2023), 25 pages. https://doi.org/10.1145/3591239