Specification-Guided Component-Based Synthesis from Effectful Libraries

ASHISH MISHRA, Purdue University, USA
SURESH JAGANNATHAN, Purdue University, USA

Component-based synthesis seeks to build programs using the APIs provided by a set of libraries. Oftentimes, these APIs have effects, which make it challenging to reason about the correctness of potential synthesis candidates. This is because changes to global state made by effectful library procedures affect how they may be composed together, yielding an intractably large search space that can confound typical enumerative synthesis techniques. If the nature of these effects are exposed as part of their specification, however, deductive synthesis approaches can be used to help guide the search for components. In this paper, we present a new specification-guided synthesis procedure that uses Hoare-style pre- and post-conditions to express fine-grained effects of potential library component candidates to drive a bi-directional synthesis search strategy. The procedure alternates between a forward search process that seeks to build larger terms given an existing context but which is otherwise unaware of the actual goal, alongside a backward search mechanism that seeks terms consistent with the desired goal but which is otherwise unaware of the context from which these terms must be synthesized. To further improve efficiency and scalability, we integrate a conflict-driven learning procedure into the synthesis algorithm that provides a semantic characterization of previously encountered unsuccessful search paths that is used to prune the space of possible candidates as synthesis proceeds. We have implemented our ideas in a tool called Cobalt and demonstrate its effectiveness on a number of challenging synthesis problems defined over OCaml libraries equipped with effectful specifications.

CCS Concepts: • **Software and its engineering → General programming languages**.

Additional Key Words and Phrases: Component-based Synthesis, Type Specifications, Effects, Conflict-Driven Learning

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

1 INTRODUCTION

Many useful programming tasks can be efficiently expressed by intelligently composing the elements found in a library of available APIs (or components). Program synthesis queries, in particular, can benefit from the ability to use library function calls in synthesizing terms. This observation has led, in recent years, to the development of several useful component-based synthesis tools [Feng et al. 2017b; Guo et al. 2019; Guria et al. 2021; James et al. 2020a; Shi et al. 2019]. These methods typically use examples and/or type annotations to guide the synthesis procedure in an enumerative fashion.

However, APIs often have effects that must be taken into account when reasoning about their composition. Library implementations of imperative data structures, databases, or parsers are canonical examples where it is natural to have effectful APIs. In these domains, the effects performed

Authors' addresses: Ashish Mishra, Department of Computer Science, Purdue University, USA, mishr115@purdue.edu; Suresh Jagannathan, Department of Computer Science, Purdue University, USA, suresh@cs.purdue.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2022 Copyright held by the owner/author(s).
2475-1421/2022/10-ART147
https://doi.org/10.1145/3563310

by APIs can impose non-trivial constraints on the choice of synthesis candidates and the order in which they must be sequenced.

Ordinarily, the types associated with such functions would not expose these kinds of effects. A typical type for an update operation on the state maintained by a database library instance, for example, might simply recognize that it performs a write effect by declaring its return type to be unit, without providing specific details about how the database actually changes. Consequently, state-of-the-art purely type-directed component-based synthesis approaches [Feng et al. 2017b; Guo et al. 2019] when applied to these kinds of libraries could easily synthesize unsound programs because the necessary protocol constraints are not reflected in library function types. Simply embellishing a library with coarse read/write effect annotations [Guria et al. 2021] is also unlikely to be effective for problems like these because the lack of fine-grained effect tracking would require the synthesis procedure to explore an intractably large space of possible candidate compositions, since every successful call to an effectful operation potentially alters the underlying state.

However, advances in mechanized proof assistants and automated theorem provers have made it increasingly worthwhile for library developers to provide detailed specifications that can be used as part of a verification task. For example, F* [Swamy et al. 2013] and VOCal [Charguéraud et al. 2017] are significant efforts aimed at developing mechanically-verified effectful libraries of general-purpose data structures and algorithms. In this paper, we show how library specifications of the kind produced by these efforts can also be effectively repurposed to guide complex component-based synthesis tasks.

Our approach introduces a new specification-guided synthesis strategy that interprets a library's specification (expressed in terms of Hoare-style pre- and post-conditions) as type specifications [Nanevski et al. 2006], using a bi-directional search strategy to enable scalability and precision. Specifically, we use strongest postcondition forward-reasoning to build larger terms from existing ones maintained by a synthesis search context, and weakest precondition backward-reasoning from the postcondition (the synthesis query) to enable goal-directed search. Alone, each process has important weaknesses - forward reasoning lacks knowledge about the synthesis goal, while backward reasoning has an incomplete view of the context from which terms must be synthesized. We show how to mitigate these weaknesses, and exploit their underlying synergies, by integrating both within a unified synthesis framework. For improved efficiency and scalability, we additionally leverage a *conflict-driven* learning strategy [Biere et al. 2009; Feng et al. 2018; Ganzinger et al. 2004] in the context of effectful program synthesis, to build a knowledge base that records *discriminating propositions* associated with previously encountered incorrect synthesized terms that can be used to more intelligently guide the search process and safely prune the space of possible candidates we need to consider. The need for such careful integration arises from the unbounded search space that must be explored to satisfy a query - having effectful libraries means that every call to a library method in a synthesized term can potentially result in a new heap state that reflects the effectful behavior of the method, leading to a potential explosion of possible candidate programs that the synthesis procedure may have to consider.

This paper makes the following contributions:

- We present a novel bi-directional deductive synthesis strategy for specification-guided component synthesis of effectful libraries. The synthesis strategy alternates between forward and backward enumerative search, seeking to compose terms consistent with the library specifications.
- We address scalability issues by additionally integrating a CDCL-style learning component that builds a knowledge base of failed candidate terms that can be used as search proceeds to avoid reconsideration of previously identified infeasible terms.

- We present detailed experimental results on an implementation of these ideas called Cobalt that enables component-based synthesis of OCaml libraries equipped with effectful specifications. Our results demonstrate the utility of our approach over a range of different application domains with respect to both expressivity of the synthesis queries that can be handled, and scalability over the size of the search space that must be navigated.

The remainder of the paper is organized as follows. In the next section, we provide a detailed overview of our approach. Section 3 presents a declarative bi-directional type-checking formulation of the synthesis procedure. Section 4 formalizes the synthesis algorithm along with details of the CDCL-learning approach used to improve enumerative search. Additional details about the implementation, along with benchmark results, are presented in Section 5. Related work is given in Section 6, and conclusions are presented in Section 7.

2 OVERVIEW

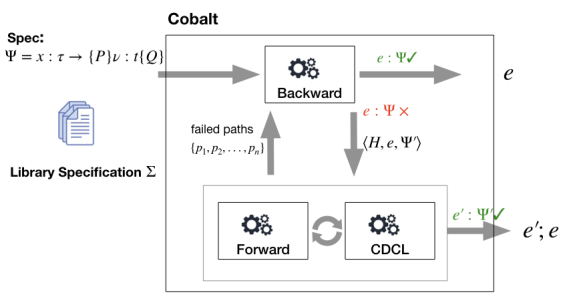


Fig. 1. An overview of the Cobalt synthesis process.

Figure 1 depicts Cobalt's synthesis procedure and its core components. Cobalt takes as input a Hoare-triple style query specification Ψ and a set of library function specifications Σ. For Σ, we rely on available verified libraries like [Charguéraud et al. 2017; Swamy et al. 2013] that come equipped with effectful specifications; from a user's perspective, a Cobalt user thus only needs to provide a declarative specification for the synthesis goal.

To motivate our approach, consider a string Table data structure adopted from [Sekiyama and Igarashi 2017] and implemented using a mutable string list in an ML-like language. The table maintains the invariant that its elements are pairwise distinct. It provides a set of effectful library functions to add a new string, to check membership, etc., on a table instance. These components have associated specifications capturing their semantics as shown in Figure 2a.

To capture the effectful behavior of these functions, we use specifications that express pre- and post-conditions over abstract heaps. For instance, the specification for the add_tbl function (refer Figure 2a) has a precondition that defines a stateful constraint on its input string s, requiring that it not be present in the table referenced by tbl in the input heap; specifications express these constraints in terms of first-order predicate logic formulae built from interpreted select/update operators [McCarthy 1993] on the heap (such as sel) and user-defined uninterpreted function symbols like mem and size defined over tables. The postcondition captures the behavior of adding s to the table, assuming the precondition holds, by relating the state of the table after the function completes (Tbl') to its state on entry (Tbl); specifically, it constrains the size of Tbl' to be one more than Tbl, and asserts that s is a member of Tbl'. The Tbl and Tbl' heap objects are accessed via input heap h and output heap h', respectively[1]. The specifications given for other library functions are similar but simplified to reduce clutter. In particular, we drop quantifiers when obvious and assume Tbl and Tbl' represent the table tbl in pre- and post-heap h, h', respectively.

[1]We capitalize variables that correspond to ghost state in specifications; these are intermediate computed heap values that do not appear as arguments or results of library functions.

```
type pair = Pair of float * int
type table = [string] ref

add_tbl : (tbl : table * s : string) →
{∀ h, Tbl. sel (h, tbl) = Tbl ∧ not (mem (Tbl, s)}
    v : unit
{∀ h, v, h', Tbl, Tbl'.
    sel (h', tbl) = Tbl' ∧ sel (h, tbl) = Tbl ∧
    mem (Tbl', s) ∧ size (Tbl') == size (Tbl) + 1};

mem_tbl : (tbl : table * s : string) →
{true} v : bool
{Tbl' = Tbl ∧ ([v=true] <=> mem(Tbl', s)) ∧
            ([v=false] <=> not (mem (Tbl', s)))};

size_tbl : (tbl : table) →
{true} v : int {Tbl' = Tbl ∧ v == size (Tbl)};

fresh_str : unit →
{true} v : string {mem (Tbl', v) = false ∧ Tbl' = Tbl};

avg_len_tbl : (tbl : table) →
{size (Tbl) > 0} v : float {Tbl' = Tbl ∧ minmax (Tbl', v)};

clear_tbl : (tbl : table) → {true} v : unit {size (Tbl') = 0};

(* remove less than *)
rlt_tbl : (tbl : table * s : string) →
{true} v : unit {size (Tbl') <= size (Tbl)};

(* remove greater than *)
rgt_tbl : (tbl : table * s : string) →
{true} v : unit {size (Tbl') <= size (Tbl)};
```

(a) Effectful specifications for a Table library.

```
(*A Safety Query-Spec*)
goal1 : (tbl : table * s : string) → v:
    float

(*A Functional Query-Spec*)
goal2 : (tbl : table * s : string) →
{True}
  v : pair
{∀ h, v, h', Tbl, Tbl'.
  sel (h, tbl) = Tbl ∧ sel (h', tbl) = Tbl'
      ∧
  mem (Tbl', s) ∧
  size (Tbl') == size (Tbl) + 1};

(*A Correct Solution*)
goal2 (tbl : table * s : string) =
  b1 ← mem_tbl (tbl, s);
  if (b1)
   then s1 ← fresh_str ();
        _ ← add_tbl(tbl, s1);
        x1 ← average_len_tbl (tbl);
        y1 ← size_tbl (tbl);
        return Pair (x1, y1)
   else _ ← add_tbl (tbl, s);
        x1 ← average_len_tbl (tbl);
        y1 ← size_tbl (tbl);
        return Pair (x1, y1)
```

(b) Functional query-spec and a solution.

Fig. 2. Effectful specifications for a Table library, Synthesis Queries and a Solution

Figure 3 depicts the behavior of the add_tbl (tbl, s) library function over an input example table. The reference tbl refers to a table (a list of strings) named Tbl before the execution of the function. An invocation add_tbl (tbl, Dec) is executed provided the precondition that the string to be added (Dec) is not in Tbl is true. If so, the function adds the string to the table, yielding the post-state in which tbl now refers to a new table (labeled as Tbl').

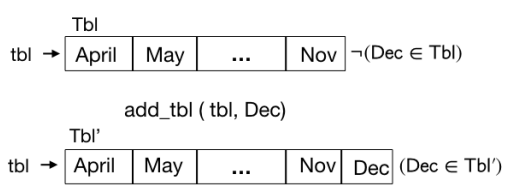


Fig. 3. A pictorial representation of the effects induced by a call to add_tbl.

Synthesis Problems. Given this library, there are two kinds of synthesis queries (*query-specs*) that can be made. One is a type inhabitation query similar to what is possible in other type-directed

component-based synthesis approaches [Feng et al. 2017b; Guo et al. 2019; Guria et al. 2021]. We call these queries *safety queries* since the synthesis goal is to generate a type-safe term using library components. Figure 2b shows such a query (`goal1`) for a function that given a table instance and as arguments returns a float. A solution to this query might be a function that applies add_tbl to the goal's arguments and returns the average length of the new table via avg_len_tbl.

Our focus in this paper, however, is on solving richer queries (which subsume these queries) that exploit interesting effect-based functional correctness properties desired by the client program. For instance, our goal might be to synthesize a function that adds its input string to a table, returning a pair of the average length and size of the new table. These kinds of *effectful queries* can be specified using a *query-spec* such as the one for `goal2` shown in Figure 2b.

This goal specifies a function that takes a table instance (tbl) and string parameter (s) as arguments and whose body satisfies the provided pre- and post-conditions. The precondition imposes no constraints on the table or string. The postcondition is a relation between the table in the pre-heap (Tbl) and the post-heap (Tbl'); it requires the synthesized function to produce a Tbl' whose size is one greater than Tbl and that contains the string s. The result type of the function, however, additionally requires that the function return a *pair* of float and int values. Observe that there are no functions in the given interface that explicitly returns a pair, although size_tbl returns the size of the table as an int, and avg_len_tbl returns a value of type float, provided that its table argument is not empty.

2.1 Solution Overview

To explain how Cobalt synthesizes a suitable function (shown in Figure 2b) given these various constraints, we first explain the details of its bi-directional search strategy and CDCL-inspired search algorithm.

2.1.1 Weakest Precondition Guided Search. The synthesis procedure begins in a *backward* phase, starting from the query's postcondition and return type. It maintains a *context*, a list of library functions and arguments provided by the user in the *query-spec*, as well as path conditions. The procedure searches for a function f under an initial context that contains the arguments declared in the query (e.g. s) and libraries, such that f can be invoked in this context, leading to the required postcondition. To make this decision, Cobalt uses a weakest precondition call rule and performs the following check, assuming the library specification for f is $(\overline{x_i : \tau_i}) \rightarrow \{Pre_f\}$ v : t $\{Post_f\}$:

$\forall$ h. ([y_i/x_i]Pre_f) h =>
  ($\forall$ v:t. h'. ([y_i/x_i]$Post_f$) h v h' => Post h v h')

Given synthesized arguments y_i, it searches for a function f to which these arguments can be applied such that $f's$ precondition is satisfied by the existing context, and f's postcondition implies the postcondition of the query. To illustrate, suppose we have a query; goal:

(y$_i$: int) $\rightarrow$ { dom (h, i) = true} {v : 'a} { $y_i \geq 5 \wedge$ sel (h', i) $\leq$ 20 }

If the synthesis context includes a condition (x $\geq$ 10), then given two functions with signatures,

f : (x : int) $\rightarrow$ { true } v : int { x == 10 $\wedge$ sel (h', i) == 10 }
g : (x : int) $\rightarrow$ { true } v : int { x == 4 }

the procedure synthesizes the call, f(x), discarding the synthesis candidate g(x). Note that this is a very strong requirement since the postcondition might impose additional constraints not considered by f. Indeed, we might have a situation where the above check does not hold, but in which Post can be translated into a form R $\wedge$ Post' where R is a *frame* [Reynolds 2002]. In such a case, we can choose a function candidate if it satisfies the following weaker check:

```
∀ h. R ∧ ([y_i/x_i]Pre_f) h =>
    (∀ v:t. h'. ([y_i/x_i]Post_f) h v h' => Post' h v h')
```

This is an instance of a *framing* problem and in Section 3.2.1, we discuss important optimizations that allow us to soundly weaken this rule to allow partial satisfaction of the query's goal.

If Cobalt does not find any effectful function satisfying the check, it searches for a pure function with the required return type and generates a subprogram with holes called *hypotheses* and a weakest precondition for this pure function call. In our example, there is no effectful library function that immediately satisfies the goal. Thus, Cobalt chooses the pure Pair constructor and generates a term represented by the following derivation:

```
[(s: string)] ⊢ {True} (v : pair)
{size (Tbl') = size (Tbl) + 1 ∧ mem (Tbl' s)}
    ⇝
[(s: string)] ⊢
{ True }
 x1 ←{(??) : float}; y1 ←{(??) : int}
{size (Tbl') = size (Tbl) + 1 ∧ mem (Tbl', s)}
--------------------------
return Pair (x1, y1)}
```

Listing 1. A backward (postcondition)-guided derivation; we omit introduction of the table instance tbl for perspicuity - Tbl and Tbl' represent tbl's value in the pre- and post-heap, resp.

The derivation is of the form,

$$\Gamma \vdash \{\mathsf{P}\}(v : \tau)\{\mathsf{Q}\}) \rightsquigarrow (\Gamma \ \vdash \{\mathsf{P}\}H; \mathsf{WP}(t, \mathsf{Q}) \mid t'$$

and yields a hypothesis H, a possibly *holed* term, along with a predicate ($\mathsf{WP}(t, \mathsf{Q})$) constructed by applying the weakest precondition semantics for the term chosen by the synthesis algorithm; in the above example, t' = return Pair (x1, y1). Since there are no available terms in the context corresponding to the required constructor arguments (an int and float), a new term is created with two holes of appropriate argument types; these terms are bound to fresh variables and used as arguments to the Pair constructor. Thus, H captures potential program shapes for the current synthesis choice. Observe that neither avg_len_tbl nor size_tbl, which can contribute to the appropriate return type, have specifications that align with the currently synthesized term; for example, avg_len_tbl, although returning a float, also requires that its input table size is greater than 0, a property that is not ensured by the holed term's precondition (True).

Consequently, further progress on the backward derivation stalls. Rather than aborting and searching for a new candidate, we instead proceed to apply a forward search starting from the existing precondition of the current term. To make sure that we avail of the information learnt from the backward derivation, we pass H (the sequence of holed terms constructing x1 and y1) and the current weakest precondition - `size (Tbl') = size (Tbl) +1 ∧ mem (Tbl', s) -` as the new postcondition for the forward search.

Although our backward search in this example simply does a pattern match over the data constructor Pair, our synthesis procedure can, in fact, generate interesting non-trivial programs of larger size by searching for a function in the library iteratively at each step, such that its specification allows for valid weakest-precondition reasoning for the given post-condition. For instance, consider a simple synthetic example with two library functions, m1 and m2:

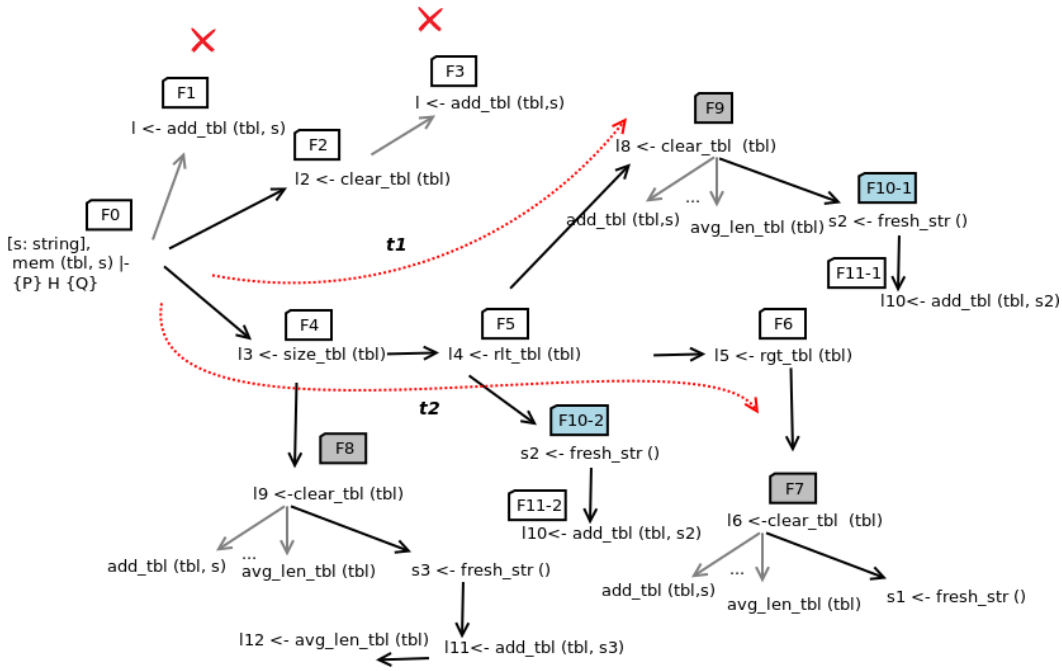


Fig. 4. A partial forward enumeration. gray edges show type- and/or specification-incorrect discarded transitions, **black** edges show type-correct, potentially explorable transitions, and red-dashed edges give examples of type-correct paths that do not lead to required solution. Labeled nodes with same color show equivalent-modulo-stuckness nodes.

res : ref int;
flag : ref bool;
m1 : unit ⟶ {res = 7} v : int {res' = res + 3}
m2 : unit ⟶ {res = 5 ∧ flag = false}
v : int
{res' = res + 2 ∧ flag' = true}

The methods' specifications highlight how the functions manipulate two mutable references: an integer res, and a boolean flag. Given this library and the following goal query:

```
goal : {res = 0 ∧ flag = false} (v : unit) {res' = 10};
```

backward synthesis starts by generating a partial term (a term with holes) (λ (). (??) : unit; m1 ()) as the specification for m1 satisfies the weakest-precondition check. It then continues trying to fill the hole to find the next such function with (WP(m1 (), {res' = 10}) as an updated postcondition. Now, the specification for m2 satisfies this check and thus a bigger program is created, consequently generating the term (λ ().(??) : unit ; m2(); m1()), before passing the synthesis to the *forward-synthesis* component.

Forward synthesis has two subcomponents (refer Figure 2) that collectively attempt to synthesize a term e', which when unified with the partial term e, synthesized by the backward synthesis, gives the required solution. If it fails to find such a term, it again invokes the Backward component with information about the failed program. Forward synthesis uses strongest post-condition reasoning to synthesize a term in a forward fashion.

2.1.2 Strongest Postcondition Guided Search. Because the body of the function being synthesized may contain top-level conditional branches or match-case statements, search begins by first synthesizing the body of the function as a top-level branching term. Specifically, Cobalt looks for functions with Boolean return types, or arguments in the query-spec that have inductive type (like lists, trees, etc.) and applies proof rules for *if-then-else* and *match* respectively to introduce top-level conditional branching or matching synthesis sub-problems.

For instance, given the specification in Listing 1, and the specification for Boolean-valued library function mem_tbl (refer Figure 2a), Cobalt generates the synthesis sub-queries shown in Figure 5 by applying an *if-then-else* synthesis rule and introducing true and false postconditions from mem_tbl's specification to the preconditions of the term's *true* and *false* branches (shown in red). We now explain how Cobalt proceeds with the synthesis query for the *true* branch. The enumeration procedure is conceptually similar to the backward derivation, but uses the strongest postcondition of the already-built term to guide the choice of the next library component candidate. Intuitively, Cobalt iteratively builds type-correct program terms of increasing length in a depth-first manner until it finds a program that satisfies the hypotheses (the holed terms), and the required specifications. Not surprisingly, the domain of such terms is unbounded since we can generate distinct unrolled looping terms like $t_1; t_2{}^*; t_3$, $t_1; t_2; t_3{}^*$, etc. of arbitrarily large depth. To mitigate this situation, we bound the maximum depth of search to a depth value k on the length of program terms. The value of k can be chosen by the user and can be iteratively incremented. For ease of presentation, assume ($k = 5$) for our running example. Suppose during enumeration we have synthesized a term t of size less than k. The procedure first tries to verify if t is the required solution by performing two checks that ascertain whether: a) t satisfies the hypothesis H ($t \prec H$); and, b) SP (P, t) $\Rightarrow$ Q', where P and Q' are pre- and post-specifications for the query specification. Informally, $t \prec H$ if t is a term that has the same shape as H with every holed term replaced by a type-consistent concrete one.

```
λ (tbl : table, s : string).
 b1 ← mem_tbl (tbl, s);
 if (b1) then
  {∀ h. sel (h, tbl) = Tbl ∧ (mem(Tbl, s)}
     (x1 ← {(??) : float}; y1 ← {(??) : int}
  {size (Tbl') = size (Tbl +1) ∧ mem (Tbl' s)}
 else
  {∀ h. sel (h, tbl) = Tbl ∧ not(mem(Tbl, s)}
     (x1 ← {(??) : float}; y1 ← {(??) : int}
  {size (Tbl') = size (Tbl +1) ∧ mem (Tbl' s)}
```

Fig. 5. A partial program with holes and top-level branching.

If t satisfies these checks, the procedure returns t as the required solution. However, if either of the two checks fail, a search commences to look for a component f that can be sequenced with t. To guide this search, Cobalt uses a strongest postcondition call rule and performs the following check, assuming the library specification for f is $(\overline{x_i : \tau_i}) \rightarrow \{Pre_f\}$ v : t $\{Post_f\}$:

$\forall$ h. (SP (P, t)) h => ([y_i/x_i]Pre_f) h

To illustrate, consider again a similar example scenario discussed in the previous section. Suppose we have a query; goal:

(y_i : int) $\rightarrow$ { dom (h, i) = true $\wedge$ $y_i \geq 5$ $\wedge$ sel (h, i) $\leq$ 20} v : 'a { ... }

Given two functions with signatures:

f : (x : int) $\rightarrow$ { x $\geq$ 5 $\wedge$ sel (h, i) $\leq$ 20} v : int { x == 10 $\wedge$ sel (h', i) == 10 }
g : (x : int) $\rightarrow$ { x < 5 $\wedge$ sel (h, i) $\leq$ 10 } v : int { x == 4 }

the synthesis procedure synthesizes the call f (y_i), discarding the synthesis candidate $g(y_i)$.

Figure 4 presents a pictorial representation of this enumeration process for some of the *true* branch paths of the synthesis query. Each choice made by the search process is given a label (Fi). The edges include black edges representing explored edges as well as gray edges showing discarded choices encountered by the forward call-rule. For instance, edge (`F0 -> F1`) is disallowed since the branch precondition (mem(Tbl, s)) is inconsistent with the precondition for (add s). Similarly, we have other discarded edges like (`F2 -> F3`), etc. The rule thus allows the search procedure to discard multiple incorrect programs early on; for instance, given the initial query above, Cobalt prunes out incorrect paths such as {_ <- add_tbl (tbl, s); ....} thereby significantly reducing the search space of possible candidates.

2.1.3 Conflict-Driven Learning Based Enumeration. While the forward (similarly backward) search process can prune out many incorrect programs quickly, the number of candidate programs (paths with black edges in the figure) is likely to be still very large, making a naïve enumerative search over this space infeasible. A primary reason is the likelihood of repeated exploration of previously seen paths allowed by forward reasoning that do not lead to the goal postcondition. For instance, consider the path (`F0 -> F4 -> F5 -> F6 -> F7`). Here, the corresponding program term is not a solution since the postcondition of the synthesized term:

```
∃ Tbl1, Tbl2, Tbl3. ∀ Tbl, Tbl'.
   Tbl1 = Tbl ∧ size (Tbl2) <= size (Tbl1) ∧
   size (Tbl3) <= size (Tbl2) ∧
   size (Tbl') = 0
```

does not imply the postcondition of the true branch of the current synthesis candidate, which requires that

```
   size (Tbl') = size (Tbl +1) ∧ mem (Tbl', s)
```

Further, since the depth of the path when reaching `F7` (here, 4) is less than the depth-bound (k=5), the search process seeks a component that can satisfy the required postcondition. Unfortunately, no such choice is possible. Thus, the algorithm backtracks and makes a different choice at one of the earlier nodes. We call a node like `F7` that can no longer make progress for a given k as a *k-bound-stuck-node*, inspired by the notion of a *conflict-node* in conflict-driven learning approaches [Biere et al. 2009; Zhang et al. 2001].

Suppose, the algorithm backtracks to node `F5` (over multiple steps); unfortunately, it again faces a choice similar to the choice at the stuck-node `F7` to select/discard the function clear_tbl. Since the term corresponding to path (`F0 -> F4 -> F5 -> F9`) (call it t_1) is syntactically distinct from the term corresponding to (`F0 -> F4 -> F5 -> F6 -> F7`) (call it t_2), the algorithm cannot trivially prune out this choice. But, by choosing `F9`, it must again performs checks similar to those performed under the k-bound-stuck-node `F7`. For instance, the algorithm would again visit discarded terms (gray edges) like (add_tbl(tbl, s), avg_len_tbl (tbl), etc.). Finally, it will eventually find itself at another k-bound-stuck node (`F10-1`) causing it to backtrack to `F5`. Once again, it faces a choice (`F5 -> F10-2`), which is similar to the new stuck node found at (`F10-1`). Figure 4 depicts how the algorithm repeatedly traverses many paths similar to an already visited stuck path, and must therefore eventually backtrack.

Even though concluding the similarity between t_1 and t_2 is not possible syntactically, we observe that this similarity arises because the algorithm is making the same function choice (i.e. clear_tbl) and the strongest postconditions SP (P, t_1) and SP (P, t_2) that guide the search process under `F9` and `F7` are related by (SP (P, t_1) $\Rightarrow$ SP (P, t_2)). In an unbounded depth-first search (k=∞), for any path explored under `F9`, there exists a path explored under `F7`, and hence by knowing that `F7` does not lead to a solution (i.e. is a k-bound-stuck-node), we can conclude that `F9` will also not lead

to a solution. We can, therefore, discard the exploration of the tree rooted at F9 by learning the postconditions at F7 and F9. We call nodes like (F7 and F9) as *equivalent-modulo-stuckness*, and highlight their similarity by depicting them with the same color in Figure 4.

However, since our exploration is bounded by terms of size k, there may be paths that were prematurely truncated under F7 but can make progress under F9. For example, consider a path (F10-1 -> F11-1) under F9. Eagerly discarding F9 would lead us to miss such paths and may result in failure to satisfy a feasible synthesis query under a given k bound. Notice, however, that for each such path under F9, there is a smaller path e.g. (F10-2 -> F11-2) that is also reachable and can lead to a solution if the longer path under F9 can. Thus F7 and F9 can be assumed to be logically equivalent-modulo-stuckness; we can thus safely discard the exploration of the tree rooted at F9, given that there is an equivalent path at F10-2.

Based on the above observations, we equip our search procedure with a conflict-driven learning component called *CDCL-search* that learns *discriminating propositions* (D^k (Fi)) associated with each visited *k-bound-stuck-node* Fi and uses them to discard future exploration of nodes that are logically equivalent to an earlier k-bound-stuck-node, modulo the stuckness property.

We explain the working of the algorithm using our running example. Upon encountering a *k-bound-stuck-node* (e.g. node F7), our CDCL search procedure learns two propositions. First, it learns a proposition S_p, which we call the *stuck-path proposition* that captures the post-state for the k-bound-stuck-node (e.g. F7). Second, it creates a disjunctive formula T_p called *truncated proposition* containing a disjunct for each call truncated prematurely for the k-bound-stuck-node (e.g. F7 -> fresh_str). The idea is to learn information about paths that were taken but were prematurely left unexplored due to the bound k. The *discriminating proposition* for a k-bound-stuck node D^k (k-bound-stuck-node) is given by a tuple (S_p, T_p).

```
{not ( ∀ Tbl'. size (Tbl') = 0 => size
    (Tbl') = 0) } ∨
{(∀ Tbl, Tbl'. ... ∧ size (Tbl') = 0 =>
     true ∧
        (not (∀ Tbl, Tbl'. size (Tbl')
             <= size (Tbl) => true))
             }
```

Fig. 6. Checks derived using the discriminating proposition D^k (F7) in CDCL-search.

D^k (F7) can help us to discard logically equivalent-modulo stuck nodes: the algorithm backtracks with this learned information to the earlier decision node F5; while making the decision at edge (F5 -> F9) with the D^k (F7) information in hand, the algorithm checks if the decision node F9 is logically equivalent-modulo-stuckness with the earlier encounter of clear_tbl(tbl). The algorithm performs the following checks, where t_3 is the term corresponding to path (F0 -> F4 -> F5), and $[\![$ F7 $]\!]$ = clear(), the function invoked at node F7 in Figure 4:

(not ($D^k([\![\text{F7}]\!]).S_p$ => SP (P, t_1))) ∨
(SP (P, t_1) => $D^k([\![\text{F7}]\!]).T_p$ ∧
not (SP (P, t_3) => $D^k([\![\text{F7}]\!]).T_p$))

Intuitively, these two disjuncts check the two observations discussed earlier. The first disjunct captures the fact that any path that can be explored by making this choice was already visited and seen to be leading to a stuck-node under the earlier exploration of clear_tbl(tbl) at F7. The second disjunct verifies that for any node that was truncated prematurely under the stuck-node and that can make progress under the current choice, there is an equivalent path in a tree rooted outside F9. For our running example, this translates and simplifies to checks shown in Figure 6. Since both these disjuncts are false, the CDCL-search algorithm decides that for the current value of k, the two nodes are logically equivalent-modulo-stuckness and it can thus safely discard the exploration of F9.

$c \in$ *Constants*		
$\mathsf{x} \in$ *Variables*		
$\ell \in$ *Locations*		
$\mathsf{v} \in$ *Value*	::=	$c \mid \ell \mid \lambda\,(\mathsf{x}{:}\tau).\ \mathsf{e}_{ip} \mid \mathsf{D_i}\ \overline{x_j}$
$\mathsf{e}_p \in$ *Pure Exp*	::=	$\mathsf{x} \mid \mathsf{v}$
$\mathsf{e}_{ip} \in$ *Impure Exp*	::=	$\mathsf{f}\ (\overline{\mathsf{e}_p}) \mid \mathsf{ref}\ \mathsf{v} \mid \textbf{match}\ \mathsf{e}_p \textbf{with}\ \mathsf{D_i}\ \overline{\mathsf{x}_j} \rightarrow \mathsf{e}_{ip}$
		$\mid \textbf{if}\ \mathsf{e}_p\ \textbf{then}\ \mathsf{e}_{ip}\ \textbf{else}\ \mathsf{e}_{ip} \mid \textbf{return}\ \mathsf{e}_p \mid \mathsf{x} \leftarrow \mathsf{e}_{ip1};\ \mathsf{e}_{ip2} \mid \diamond$
$\mathsf{f} \in$ *Library Function*		
$\diamond \in$ *hole*	::=	$(??) : \tau$
$\mathsf{TN} \in$ *TypeNames*	::=	list, tree, pair, ...
$\mathsf{t} \in$ *Base-Type*	::=	$\mathsf{int} \mid \mathsf{bool} \mid \ldots \mid \mathsf{heap} \mid \mathsf{TN} \mid \mathsf{t}\ \mathsf{ref}$
$\tau \in$ *Type*	::=	$\{\nu : \mathsf{t} \mid \phi\} \mid (\mathsf{x} : \tau) \rightarrow \tau \mid \{\phi_1\}\ \nu : \mathsf{t}\ \{\phi_2\}$
$\phi, P, Q \in$ *Propositions*	::=	$\mathsf{true} \mid \mathsf{false} \mid Q(\overline{x_i})$
		$\mid \neg\,\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \forall\,(\mathsf{x} : \mathsf{t}).\ \phi \mid \exists\,(\mathsf{x} : \mathsf{t}).\ \phi$
$\Gamma \in$ *Type Context*	::=	$\varnothing \mid \Gamma, \mathsf{x} : \tau \mid \Gamma, \phi$
$\Sigma \in$ *Library*	::=	$\varnothing \mid \Sigma, f : (\overline{x_i : \tau_i}) \rightarrow \tau$
		$\mid \Sigma, \mathsf{D_i}\ \overline{x_j : \tau_j} \rightarrow \tau$

Fig. 7. λ_{eff} Expressions and Types

The forward-algorithm continues the exploration with learning until it finds a solution for the given value k. If it fails to find a program, it returns the failed paths of lengths upto k to the backward search again, in a *handshaking* step. At this point, the backward algorithm may need to backtrack and make different choices. By supplying failing information about paths, the backward search can avoid choosing equivalent terms modulo these failures. We depict various components of the synthesis procedure in Figure 1. Applying these mechanisms (backward+forward+cdcl) to the original goal (goal2), Cobalt synthesizes the solution shown in Figure 2b in approximately 7 seconds. (i.e. forward+cdcl) finds a solution in 10 seconds; a forward-no-cdcl synthesis strategy explores many more paths (compared to forward+cdcl) and takes 28 seconds, while a backward-alone synthesis strategy fails to find a solution within a 10 minute time-bound.

3 COBALT SYNTHESIS

We now present a set of *bi-directional* search rules and the CDCL-search algorithm presented in the last section that formalizes our specification-guided synthesis strategy.

Synthesis Language. Our synthesis procedure operates over a core-calculus λ_{eff} [Wadler and Thiemann 2003], an extension of the call-by-value simply-typed λ-calculus tailored to support specification-guided component-based synthesis. The language differentiates between pure and impure expressions, the latter being those whose computation can induce effects. Values are constants of base type, type constructor applications, (closed) lambda expressions, and locations. Pure expressions are values and variables. Impure expressions include calls to effectful library functions, expressions that create references, pattern-matching and conditional expressions whose bodies may introduce effects, a monadic return expression, and and a monadic sequencing expression ($x \leftarrow \mathsf{e}_{ip1}$; e_{ip2}) that evaluates e_{ip_1} and binds its result to a variable x in e_{ip_2}.

As we have seen in our earlier examples, the language also allows typed holed-expressions of a given type τ that takes the form $((??) : \tau)$. Such a term represent an unknown expression in a program that must be constrained by the type τ; our synthesis procedure transforms such expressions by replacing these holes with concrete terms.

Types and Environments. The type language includes support for *base types* such as types for integers, Booleans, strings, etc., as well as a special heap type to denote the type of abstract heap variables like h, h' found in specifications. There are additionally user-defined data types TN, and type constructors used to type references that hold values of some base type. More interestingly, base types can be refined with *propositions*, and effectful computations have types defined in terms of Hoare-style pre- and post-conditions of the form $\{\phi_1\}\ v : \mathsf{t}\ \{\phi_1'\}$ that represent an effectful computation, which when executed in a pre-state satisfying proposition $\{\phi_1\}$, upon termination, returns a value v of base type t along with a post-state satisfying $\{\phi_1'\}$.

Propositions (ϕ) are first-order predicate logic formulae over base-typed variables. Propositions also include a set of *Qualifiers* which are user-defined uninterpreted functions symbols such as mem, size etc. used in our example; qualifiers also include two special interpreted function symbols (sel and update) used to model access and modification to the global heap. The type language also includes dependent-function types since arguments and return values of library functions can be associated with types that are refined by propositions[2].

Propositions in pre- and post-conditions capture *non-spatial* properties of the pre- and post-abstract heaps respectively. These properties capture actions involving accesses and modifications to heap objects associated with a heap variable (sel and update), or describe shallow structural properties of heap objects, e.g., length, head, etc. for a list. Our current implementation currently does *not* allow expression of spatial properties that describe disjointedness of heap fragments. Consequently, we assume that each heap object is always referenced by a unique path (variable x or x.f.y) and that there is no sharing of heap objects. We have found that these assumptions are not particularly onerous in the context of the OCaml libraries we have examined.

There are two environments maintained by Cobalt of particular interest: (1) environment Γ records the types of variables along with a set of propositions relevant to a specific context, and (2) and, environment Σ maps library functions and datatype constructors to their signatures.

3.1 A Cobalt Synthesis Problem

A Cobalt synthesis problem can be described as follows: given a library Σ of functions and data constructors, annotated with a suitable types, a type environment Γ, and a goal specification (Ψ), which is a dependent-function type of the form

$$(\mathsf{x} : \tau) \rightarrow (\{P\}\mathsf{v} : \mathsf{t}\{Q\})$$

where v is a free variable denoting the return value of the program and in which pre- and post-conditions P, Q may contain argument variable x, heap locations, and the return variable v, the synthesis problem seeks to synthesize an expression $e \in e_p \cup e_{ip}$ in λ_{eff} such that

$$\Gamma; \Sigma \vdash e : (\mathsf{x} : \tau) \rightarrow \{P\}\mathsf{v} : \mathsf{t}\{Q\}$$

3.2 Bi-directional Deductive Component-Based Synthesis

Given a Cobalt synthesis problem, the synthesis procedure is a *bi-directional* deductive proof-search [Delaware et al. 2015; Osera and Zdancewic 2015; Polikarpova et al. 2016] over library functions and the given query specification. We next explain each of these modes of the synthesis procedure.

3.2.1 Forward Synthesis. Figure 8a shows our forward synthesis system using synthesis rules of the following form:

$$\Gamma; \Sigma \vdash \tau \twoheadrightarrow e$$

[2]Additional details about the language's type system can be found in an accompanying technical report [Mishra and Jagannathan 2022b].

$\boxed{\Gamma; \Sigma \vdash \tau \twoheadrightarrow e}$ Forward Synthesis

$$\frac{\begin{array}{c}\Gamma; \Sigma \vdash \{v : \mathbf{TN} \mid \phi\} \twoheadrightarrow e \\ \mathbf{Di}\,(\overline{x_j : \tau_j}) \rightarrow \{v : \mathbf{TN} \mid \phi_i\} \in \Sigma \\ \Gamma_i \equiv \Gamma, \overline{x_j : \tau_j}, \{x'/v\}\phi_i \\ \Gamma, \{x'/v\}\phi, \Gamma_i; \Sigma \vdash \{P\}v : t\{Q\} \twoheadrightarrow e_i\end{array}}{\Gamma; \Sigma \vdash \{P\}v : t\{Q\} \twoheadrightarrow \mathbf{match}\ e\ \mathbf{with}\ \mathbf{Di}\,(\overline{x_j}) \rightarrow e_i} \text{FW_MATCH}$$

$$\frac{x : \tau \in \Gamma}{\Gamma; \Sigma \vdash \tau \twoheadrightarrow x} \text{FW_VAR}$$

$$\frac{\begin{array}{c}\Gamma; \Sigma \vdash \{v : \mathbf{bool} \mid \phi_t \wedge \phi_f\} \twoheadrightarrow e \\ \Gamma, \{\mathrm{true}/v\}\phi_t; \Sigma \vdash \{P\}v : t\{Q\} \twoheadrightarrow e_t \\ \Gamma, \{\mathrm{false}/v\}\phi_f; \Sigma \vdash \{P\}v : t\{Q\} \twoheadrightarrow e_f\end{array}}{\Gamma; \Sigma \vdash \{P\}v : t\{Q\} \twoheadrightarrow \mathbf{if}\ e\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f} \text{FW_IF}$$

$$\frac{\begin{array}{c}f : \overline{x_i : \tau_i} \rightarrow \{P_1\}v : t'\{Q_1\} \in \Sigma \\ \Gamma; \Sigma \vdash \tau_i \twoheadrightarrow y_i \\ \Gamma; \Sigma \vdash P \Rightarrow P_1 \\ Q' \equiv \mathbf{SP}\,(P, f(\overline{y_i})) = P \wedge Q_1 \\ \Gamma, \overline{y_i : \tau_i}; \Sigma \vdash \{Q'\}v : t\{Q\} \twoheadrightarrow e\end{array}}{\Gamma; \Sigma \vdash \{P\}v : t\{Q\} \twoheadrightarrow z \leftarrow (f(\overline{y_i})); (e)} \text{FW_CALL}$$

$$\frac{\begin{array}{c}\Gamma, R; \Sigma \vdash \{P\}v : t\{Q\} \twoheadrightarrow e \\ ((\mathbf{Vars}\,(R)) \cap (\mathbf{EVars}\,(P, Q))) = \varnothing \\ ((\mathbf{Qual}\,(R)) \cap \\ ((\mathbf{Qual}\,(P)) \cup (\mathbf{Qual}\,(Q)))) = (\varnothing)\end{array}}{\Gamma; \Sigma \vdash \{P \wedge (R)\}v : t\{Q \wedge (R)\} \twoheadrightarrow e} \text{FW_FRAME}$$

$$\frac{\begin{array}{c}\Gamma; \Sigma \vdash \{P\}v : t\{Q_1\} \twoheadrightarrow e \\ \Gamma; \Sigma \vdash \mathbf{SP}\,(P, e) \Rightarrow (Q)\end{array}}{\Gamma; \Sigma \vdash \{P\}v : t\{Q\} \twoheadrightarrow e} \text{FW_SUB}$$

(a) Forward Type Synthesis Rules

$\boxed{\begin{array}{c}\Gamma; \Sigma \vdash \{\phi_1\}\diamond\{\phi_2\} \twoheadleftarrow e \\ \Gamma; \Sigma \vdash \tau \twoheadleftarrow e\end{array}}$ Backward Synthesis

$$\frac{\Gamma; \Sigma \vdash \{P\}v : t\{Q\} \twoheadrightarrow e}{\Gamma; \Sigma \vdash \{P\}(??) : t\{Q\} \twoheadleftarrow e} \text{BW_FW}$$

$$\frac{y \notin \mathbf{Dom}\,(\Gamma)}{\Gamma; \Sigma \vdash \tau \twoheadleftarrow y \leftarrow ((??) : \tau); (\mathbf{skip})} \text{BW_HOLE}$$

$$\frac{\begin{array}{c}f : \overline{x_i : \tau_i} \rightarrow \{P_1\}v : t'\{Q_1\} \in \Sigma \\ \Gamma; \Sigma \vdash \tau_i \twoheadleftarrow y_i \\ \Gamma; \Sigma \vdash P_1 \Rightarrow (Q_1 \Rightarrow Q) \\ P' \equiv \mathbf{WP}\,(f(\overline{y_i}), Q) = P_1 \wedge (Q_1 \Rightarrow Q) \\ \Gamma; \Sigma \vdash \{P\}(??) : \tau_i\{P'\} \twoheadleftarrow e_i \\ \overline{y_i \leftarrow (e_i)}; (f(\overline{y_i})) \notin F\end{array}}{\Gamma; \Sigma \vdash \{P\}(??) : t'\{Q\} \twoheadleftarrow \overline{y_i \leftarrow (e_i)}; (f(\overline{y_i}))} \text{BW_CALL}$$

$$\frac{\begin{array}{c}\Gamma; \Sigma \vdash \{P_1\}(??) : t\{Q\} \twoheadleftarrow e \\ (\mathbf{Holes}\,(e)) = (\varnothing) \\ \Gamma; \Sigma \vdash P \Rightarrow (\mathbf{WP}\,(e, Q))\end{array}}{\Gamma; \Sigma \vdash \{P\}(??) : t\{Q\} \twoheadleftarrow e} \text{BW_SUB}$$

$$\frac{\begin{array}{c}P \vdash P_1 \wedge (R) \\ Q_1 \wedge (R) \vdash Q \\ ((\mathbf{Vars}\,(R)) \cap (\mathbf{EVars}\,(P_1, Q_1))) = (\varnothing) \\ ((\mathbf{Qual}\,(R)) \cap \\ ((\mathbf{Qual}\,(P_1)) \cup (\mathbf{Qual}\,(Q_1)))) = (\varnothing) \\ \Gamma; \Sigma \vdash \{P_1\}(??) : t\{Q_1\} \twoheadleftarrow e\end{array}}{\Gamma; \Sigma \vdash \{P\}(??) : t\{Q\} \twoheadleftarrow e} \text{BW_FRAME}$$

(b) Backward Type Synthesis Rules

Fig. 8. Forward and Backward Type Synthesis Rules

Each such rule defines a declarative judgment explaining the generation (along with a proof-derivation) of a conclusion term e in an environment of types (Γ) and libraries (Σ) against a given type τ, using the derivation of other *well-typed* subterms in the rule's premise. Generating a variable (FW_VAR) simply requires choosing the variable of the required type from the environment. To generate a *match* expression (FW_MATCH) the procedure first recursively generates a term e using the (FW_VAR) rule for a datatype TN from the environment. Second, it creates an extended environment Γ_i for each case branch i using constructors ($\mathrm{D_i}\ (\overline{\mathrm{x}_j : \tau_j}) \rightarrow \{\ v : \mathrm{TN} \mid \phi_i\ \}$ for the TN, while replacing the bound variable v in each ϕ_i with an existential match variable x'. Finally, it recursively generates expressions (e_i) for the original synthesis problem specification in each of these extended environments. Thus, the rule allows us to break the original synthesis problem into

i subproblems that can be solved in stronger environments, thereby pushing type information from a constructor's specification ($D_i \in \Sigma$) to the synthesis query.

The generation of a conditional *if-then-else* expression (FW_IF) is similar to match with a few important differences. It first requires the generation of the Boolean test expression e. Since our focus is component-based synthesis, the synthesis procedure only has access to the library specifications (Σ) and the goal specification Ψ at its disposal. Thus, only way to generate a Boolean-typed expression is via function calls. Consequently, the procedure searches for a library function call with Boolean return type using the FW_Call rule described below and postconditions (ϕ_t and ϕ_f) for true and false return values, resp. It then creates extended environments to recursively synthesize the true and false branch by substituting true and false for the bounded variable v, and synthesizes terms for the true and false branches in their extended environments.

The FW_Call rule defines the strongest-post condition forward search procedure. The rule depicts a scenario when a single function-call does not suffice to generate a term for the required goal specification Ψ. It breaks the original synthesis into two sub-synthesis problems: First, it searches for a function f in the library with a type, such that a) the synthesis procedure can synthesize expressions $\overline{y_i}$ as its arguments (see second premise); b) with appropriate mapping for the heap and arguments variables[3], it can satisfy the forward rule for Hoare-style reasoning, i.e. the goal's precondition P implies the required precondition P_1 for f (see third premise). Successfully checking these two conditions implies that a function call expression ($f\ (\overline{y_i})$) in the current environment is well-typed. Second, it calculates the strongest-post condition (**SP** (P, ($f\ (\overline{y_i})$))) for this term, and recursively synthesizes an expression e with this as the new precondition. The expression synthesized for the original specification is a monadic sequencing of the function call ($f\ (\overline{y_i})$) and e.

Framing. The rule (FW_Frame) is concerned with expression synthesis taking into account *frames*, heap/store fragments that do not change during the evaluation of a program expression [Reynolds 2002].

The auxiliary function **Vars**(R) gives the set of reference used in R. The auxiliary function **EVars** takes a list of propositions and returns the set of existential references found in the environment used in these propositions; these existentials are introduced when computing the strongest postcondition in the FW_Call rule. The function **Qual**(R) gives the set of qualifiers (like size, mem, etc.) used in R. The premise in FW_Frame checks that the references found in the frame R are disjoint from the existential references in P and Q and that R's qualifier set is also disjoint from P and Q.

The subtype rule (FW_Sub) defines the condition for the successful termination of the forward proof search process using the standard strongest postcondition-based verification condition check.

3.2.2 Backward Synthesis. Figure 8b presents the backward synthesis inference rules whose judgments are either of the form:

$$\Gamma; \Sigma \vdash \tau \twoheadleftarrow e$$

for introducing a holed subterm into, or

$$\Gamma; \Sigma \vdash \{\phi_1\} \diamond \{\phi_2\} \twoheadleftarrow e$$

for eliminating a holed subterm from, the term being synthesized.

Backward synthesis can invoke forward-synthesis non-deterministically at any time (see rule BW_FW). In practice, we invoke the forward rule when the backward synthesis cannot make any progress, i.e. when no backward rule applies.

The backward hole rule (BW_hole) generates a holed expression bound to a fresh variable y for an arbitrary synthesis query. This is the introduction rule for a holed expression that allows

[3]We drop variable substitutions in propositions to reduce clutter in rules.

```
Synthesize(⟨Γ,Σ, Ψ⟩, F)
(1)  t := BW_Rules (Γ, Σ ,Ψ, F)
(2)  if (t ≠ (⊥, _)) then
(3)      return t;
     else
(4)      if (t = (⊥, ⟨ H, e_b, Ψ′⟩)) then
(5)          t′ := CDCL (⟨Γ, Σ, Ψ′⟩, H)
(6)          if (t′ = e_f) then
(7)              return (e_f; e_b);
(8)          else if t′ = (⊥, F′) then
(9)              Synthesize (Γ, Σ, Ψ, (F ∪ F′));

CDCL(⟨Γ,Σ, Ψ⟩, H)
(10) D := ∀ c_i ∈ Σ. D (c_i) = (true, false)
(11) F := ∅, p_i := ⊥
(12) while true do
(13)     (Γ, D, c_i) := ℛ_Choice (Γ, Σ, D, Ψ, H, p_i)
(14)     if c_i = ⊥ then
(15)         if (| p_i | > 0) then
(16)             F := F ∪ {(p_i)} ;
(17)             (p_i, D) := ℛ_Learn (Γ, Σ, ⟨ D, p_i ⟩);
(18)         else return (⊥, F) ;
         else
(19)         e := FW_SUB (Γ, Σ, (p_i;c_i), Ψ);
(20)         if (e ≠ ⊥) then return e ;
(21)         else p_i := (p_i;c_i) ;
```

Algorithm 1: The Synthesis Algorithm

the procedure to create hypotheses when backward synthesis cannot find a required term in the context.

The main rule for backward enumeration is BW_Call. The rule requires searching for a function f in the library with a return type matching the hole. Note the difference from the FW_Call rule, where we looked for any allowed function call; here, we use goal directed search instead. Once such a function is found, the rule generates arguments y_i for f by either introducing holed terms for each argument, effectively yielding new synthesis sub-queries, or finding suitable variables (using BW_Var, similar to FW_Var, not shown here) in the environment of the required type. This is an instance where the effect of having an incomplete view of the context becomes apparent during the backward search. The rule ensures that the function call can be soundly made using the weakest precondition check. This check verifies that, assuming the precondition for the function (P_1) in the given environment (Γ) holds, that the postcondition for the function (Q_1) implies the goal postcondition (Q) with appropriate substitution for heap variables and arguments[4] If this check succeeds, it further checks that the resulting term is not already seen as a failed program using the set of learned failed programs F. If successful, the weakest precondition predicate ($\mathbf{WP}((f\ (\overline{y_i})), Q)$) for the function call using Q and the function's argument and specifications is used. Finally, it creates new subproblems using this weakest precondition as the postcondition and the types of the function's arguments as the hole types.

The backward frame rule (BW_Frame) identifies a frame R using the consequence judgments in the premise, applies frame rule checks on the disjointness of variables and qualifiers, and establishes a synthesis query on the framed pre- and postconditions (P_1 and Q_1).

4 SYNTHESIS ALGORITHM

Algorithm 1 outlines the top-level synthesis algorithm and can be understood as pseudo-code for the overview of our approach given in Figure 1. The input to the algorithm is a Cobalt synthesis problem (a triple $\langle \Gamma, \Sigma, \Psi \rangle$) along with a set of explored *stuck-paths* F, initially empty. The algorithm first makes a call to the backward synthesis procedure using a function BW_Rules, a deterministic

[4]We elide variable substitutions in the rules for perspicuity.

implementation of the backward synthesis rules given in Figure 8b). In case backward synthesis does not succeed in producing a complete solution (line 4), it returns a partial solution e_b, a hypothesis H, and a new specification Ψ', which is calculated by substituting the weakest precondition for Ψ's postcondition and the partial solution e_b. The algorithm invokes the CDCL routine (line 5) with the hypothesis H, the updated specification Ψ'. The CDCL routine if successful, returns a solution e_f for Ψ', which is then sequenced (using a monadic-sequencing expression) with the partial backward solution e_b, to give the required solution for the original problem (line 8). Otherwise, the synthesis routine is recursively called (line 9) with an updated stuck-paths set ($\mathsf{F} \cup \mathsf{F}'$).

4.1 Conflict Driven Learning Based Enumeration

The CDCL routine takes as input a synthesis problem as well as a hypothesis H and returns either a λ_{eff} expression satisfying Ψ (line 20) or $\perp$ (line 18) if it cannot find such an expression. It maintains three data-structures: 1) a *Discriminating Propositions* map D that maps components c_i to a pair of stuck-path and truncated-path propositions as discussed in the last section; 2) a sequence of components p_i (a path) representing the partially synthesized expression; and, 3) a set of already explored *stuck-paths* F. The algorithm begins by initializing D by mapping each component in Σ with trivial propositions and the sequence p_i as an empty sequence. The search is performed by the main loop (lines 12-21) that iterates until it finds a correct expression (line 20) or has exhausted path exploration (line 18), updating D and p_i in each iteration.

The algorithm makes a choice of the next component for a given p_i and D, using a function $\mathcal{R}$_Choice (line 13), a deterministic implementation of the CDCL_CHOICE rule given in Figure 9. If the procedure is unable to find a new component (line 14), it learns new discriminating propositions for the stuck-node associated with p_i and backtracks to the previous path using a function $\mathcal{R}$_Learn (line 17), a deterministic implementation of the CDCL_LEARN rule in Figure 9, or it has exhausted all paths and terminates the loop (line 18). If a candidate component has been found, a call to the FW_Sub function (corresponding to the rule FW_SUB in Figure 8a) is performed (line 19); this call checks if the type for the expression corresponding to path (p_i;c_i) is a subtype of the original synthesis query Ψ, in which case it returns this expression. If not, the algorithm continues with an updated path (p_i;c_i).

Learning Discriminating Propositions. We introduce discriminating propositions for a k-bound-stuck-node c_i using the CDCL_LEARN rule.[5] Given a stuck-path p_i, typing (Γ) and library (Σ) environments, and an incoming discriminating propositions Map D, the rule generates a new set of discriminating propositions for the stuck-node c_i, updating D in the process, and returning a smaller path to be explored next. The learned proposition has two components. The first is a *stuck-path proposition* ϕ_s that captures the strongest postcondition for t_{p_i}, the expression corresponding to p_i for the given goal precondition ϕ.[6] The second component, a *truncated proposition* ϕ_t, is a disjunction over the preconditions of those components c_j that can in principle be invoked using the FW_call rule but which cannot due to the bound k and which are thus prematurely truncated. This is ensured by the implication ($\Sigma, \Gamma\ \phi_s \Rightarrow \phi_{c_j}$).

The CDCL_CHOICE rule uses the discriminating propositions introduced by the learning rule to prune away equivalent-modulo-stuckness paths. It returns a new function component c_i that can be used to construct a bigger λ_{eff} expression, provided an existing path p_i, a hypothesis H, typing and library environments, a discriminating propositions map D, and the goal specification Ψ. The rule first searches for a component c_i using the forward synthesis rules, and performs two

[5] A formal definition for k-bound-stuck nodes can be found in an accompanying technical report [Mishra and Jagannathan 2022b].

[6] In the following, we abuse the use of p_i to serve as both the path and the term t_{p_i} it represents for perspicuity.

$$\boxed{\begin{array}{c} H; D; \Gamma; \Sigma \vdash (\Psi, p_i) \hookrightarrow (p_i; c_i) \\ \Gamma; \Sigma \vdash (p_i, D) \hookrightarrow (p_i', D') \end{array}} \quad \text{CDCL Rules}$$

$$\frac{\begin{array}{c} \Psi \equiv \{\phi\} v : t\{\phi'\} \quad p_i \equiv c_1; c_2; ...; c_i \qquad D(f_i) = \langle \phi_{s_{f_i}}, \phi_{t_{f_i}} \rangle \qquad \phi_s \equiv \mathbf{SP}(\phi, \mathbf{p_i}) \\ \phi_t \equiv \{\bigvee_j . \phi_{c_j} \mid (c_j : (\overline{x_i : \tau_i}) \to \{\phi_{c_j}\} v : t'\{\phi'_{c_j}\}) \in \Sigma \wedge (\Gamma, \phi_s \Rightarrow \phi_{c_j})\} \\ D' = D[c_i \mapsto \langle (\phi_{s_{f_i}} \wedge \phi_s), (\phi_{t_{f_i}} \vee \phi_t) \rangle] \end{array}}{\Gamma; \Sigma; \vdash (p_i, D) \hookrightarrow ((c_1; c_2; ...c_{i-1}), D')} \text{CDCL_LEARN}$$

$$\frac{\begin{array}{c} \Psi \equiv \{\phi\} v : t\{\phi'\} \qquad \Gamma; \Sigma \vdash \{\mathrm{SP}(\phi, p_i)\} \mathrm{v} : \mathrm{t}\{\phi'\} \twoheadrightarrow c_i \\ (p_i; c_i) \prec H \quad D(c_i) = \langle \phi_s, \phi_t \rangle \quad \mid (p_i; c_i) \mid \leq k \qquad X \equiv \{\neg(\phi_s \Rightarrow \mathbf{SP}(\phi, (\mathbf{p_i}; \mathbf{c_i}))\} \vee \\ \{(\mathbf{SP}(\phi, (\mathbf{p_i}; \mathbf{c_i})) \Rightarrow \phi_t) \wedge \neg(\mathbf{SP}(\phi, (\mathbf{p_i})) \Rightarrow \phi_t)\} \\ ([\Gamma] \models X) \end{array}}{H; D; \Gamma; \Sigma; \vdash (\Psi, p_i) \hookrightarrow (p_i; c_i)} \text{CDCL_CHOICE}$$

Fig. 9. Rules for constructing and using discriminating propositions.

additional checks for the new potential path $p_{i+1} = (p_i;c_i)$: (1) that p_{i+1} satisfies the shape given by the hypothesis H, and (2) that p_{i+1} is not equivalent-modulo-stuckness to some earlier visited stuck-path. The check first generates the strongest postconditions for the expressions corresponding to paths p_i and p_{i+1} respectively. It extracts the discriminating proposition pair (ϕ_s, ϕ_t) for the component c_i and generates a check with two disjuncts. Failure of the first disjunct intuitively implies that any path (and hence the corresponding term) that can be explored by choosing c_i was explored earlier without leading to a solution and hence the exploration of c_i and all the following paths can be safely skipped without effecting the completeness of the search process. In such a case, we should choose c_i only if we had prematurely truncated some path earlier that can also be taken along p_{i+1} (checked using conjunct #1 in the second disjunct) and which cannot be explored without exploring p_{i+1} (checked using conjunct #2 in the second disjunct).

4.2 Soundness

Programs synthesized by Cobalt are correct with respect to the provided query specification Ψ assuming the validity of each library function against their specifications.[7]

Theorem 4.1 (Soundness). *Iff* Synthesize $(\langle \Gamma, \Sigma, \Psi \rangle, \varnothing) = e$ *then* $\Gamma; \Sigma \vdash e : \Psi$.

Since the CDCL routine (refer Algorithm 1) can possibly discard a correct program if it can ensure that there exists another program satisfying the given query-spec of smaller size, the completeness argument is relative to a query spec.

Theorem 4.2 (Completeness). $\forall\, k$. *If* Synthesize $(\langle \Gamma, \Sigma, \Psi \rangle, \varnothing) = \bot$ *then* $\nexists\, e. \mid e \mid \leq k$ *and* $\Gamma; \Sigma \vdash e : \Psi$.

5 IMPLEMENTATION AND EVALUATION

Cobalt is implemented in approximately 7300 lines of OCaml. We rely on OCaml lexing and parsing libraries OCamllex [Leroy et al. 2022] for handling the front end of our query specification language and use Z3 [de Moura and Bjørner 2008] to discharge SMT queries. The input to Cobalt is a specification file containing a library of functions and data constructors, along with their specifications, followed by a goal query specification.

[7]Proofs can be found in the accompanying technical report [Mishra and Jagannathan 2022b].

We evaluate Cobalt by synthesizing programs from several domains and consider its effectiveness with respect to the following questions:

- RQ_1 Is Cobalt effective in synthesizing programs from available verified libraries?
- RQ_2 How does Cobalt's integration of forward, backward, and CDCL search compare against each technique applied individually?
- RQ_3 How sensitive is Cobalt synthesis to the complexity of library specifications and queries?
- RQ_4 How does Cobalt compare against other state-of-the art component-based synthesis techniques when applied to specification-rich libraries?

5.1 Benchmarks

We consider a number of synthesis problems for applications drawn from three different domains; a detailed characterization of the queries can be found in [Mishra and Jagannathan 2022b]. The results of applying Cobalt to these problem domains are shown in Figures 10a, 10b and 10c. In these figures, synthesis problems for *database* applications are prefixed with "D" and are adopted from [Itzhaky et al. 2017]. These queries (D1-D11) are defined over two database applications. The first is a *Newsletter* database with a single table NS with attributes *newsletter, user, subscribed, articles, code, etc.* and effectful library functions such as *subscribe, unsubscribe, add, etc.* An example query (D5) encodes the following problem: *given a newsletter n and a user u, return the list of articles available to u in n, and then unsubscribe u from n*; the solution must take care to first check that the user is subscribed to the newsletter before unsubscribing. The second is a network firewall database that has two tables, a table of devices and a table storing sender-receiver links; its library functions include *add_device, add_connection, delete_device etc.* For example, query D7 encodes the following problem: *insert new devices d and x in the device table and create a connection between them.* Synthesizing programs from queries of this kind must take into account appropriate preconditions that reflect the effectful behavior of the library; e.g., to establish a connection to a device that is not currently in the device table requires that the device first be added.

The second domain consists of *parser* benchmarks prefixed with "P" and include stateful combinator style parsers for simplified grammars for a PNG image format and C-language declaration syntax. The libraries and specifications are constructed using the grammars of stateful parsers [Jim et al. 2010]. The libraries include subparsers and standard basic parsers for alphabet, identifier, number, etc. The synthesis queries describe the specification for bigger parsers that can be constructed using these libraries. E.g. benchmark P1 encodes the following data-dependent property: *synthesize a parser for a png-chunk using subparsers for length, typespec, content, etc. such that the combined size of typespec and content of the output chunk is always equal to the parsed length value.* Synthesizing programs that satisfy these kinds of properties must take into account the effects of upstream parsers on the length value when considering parsing candidates downstream in a parsing pipeline.

The third domain considers *imperative data structure* libraries that implement tables, queues written in OCaml; in the figures, these benchmarks are prefixed with "I". The Table library described in Figure 2a and its specifications are adopted from [Sekiyama and Igarashi 2017], while libraries for Queue are adopted from the development of mutable data structures given in Software Foundations [Appel et al. 2021]. The queries we consider involve multiple insertions, deletions, conditional insertions/deletions etc. on tables and queues, maintaining library usage protocols. For instance, I4 encodes the following query : *Given a queue of unique integers, and an integer, synthesize a program which increments the size of the queue.* The result must take into account if the given integer is present in the queue or not and then appropriately insert either the given or a new integer.

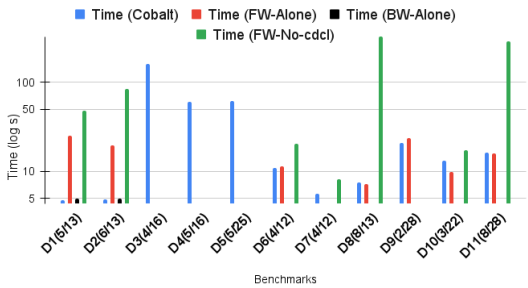
This domain also includes other OCaml data structure libraries imported from works attempting mechanical verification for OCaml libraries [Charguéraud et al. 2017]. These benchmarks are prefixed with the appropriate data-structure name, for instance "V" for OCaml Vector library, "HT" for Hash Tables, etc. The queries again include standard textbook examples of the usages of these libraries. E.g. benchmark HT3 encodes the following query: *given a hash table, and a key-value pair, add the pair in the table, create a new hash table and transfer the contents of the current table to the new table.*

In total, these libraries span 48 files and contain a total of 251 functions, a size that makes memorization of their signatures and specifications by clients impractical. The imperative data-structure Libraries contribute 105 functions, the parser library contributes 32, the database library contributes 40, with the remaining functions include constructors (e.g. Pair and Triple, etc.) and pure functions from OCaml libraries like the OCaml Core [Leroy et al. 2022] (or functions translated from the Haskell Core libraries used by other purely functional component-based synthesis approach [James et al. 2020a,b]). Note that Cobalt works on this complete library set; the alternative could be to find the minimal set of functions required for each domain. Finding such a library set *a priori* is not feasible as these functions can be called across multiple domains; e.g., a database domain benchmark may use methods from a List library defined in the imperative data-structure domain; a Queue benchmark may use a pure Pair creation function, etc. Indeed, we found that 28% of our synthesized solutions used at least one function from outside its domain.

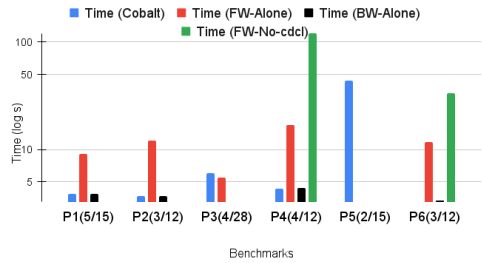
5.2 Library Specification Annotations

All the benchmarks in our evaluation were taken from verified libraries whose specifications were provided by the library authors. Fortunately there are multiple such projects currently available across a number of different domains [Charguéraud et al. 2017; Jim et al. 2010; Sekiyama and Igarashi 2017]. We adopted these specifications to the Cobalt specification language, a straightforward mechanical task for most of these benchmarks; four of the libraries defined specifications that capture richer properties than what Cobalt currenty supports and their specifications had to be slightly rewritten. For example, the specification for the Vector library in the VOCal suite leverages the algebraic theory of lists which cannot be handled using our SMT-based synthesis procedure. In these cases, we modified these specification to use more abstract notions like list membership, ordering, etc. These modified libraries are used in benchmarks V1-V3, Q1-Q3, RB1, RB2, and ZL1-ZL3. Queries were chosen to ensure that every library method for each application class would be used in at least one solution, that no two solutions would be identical, and that each solution would entail some combination of non-trivial control-flow (e.g., pattern-matching over type constructors) with library calls, and non-trivial synthesis of function call arguments.

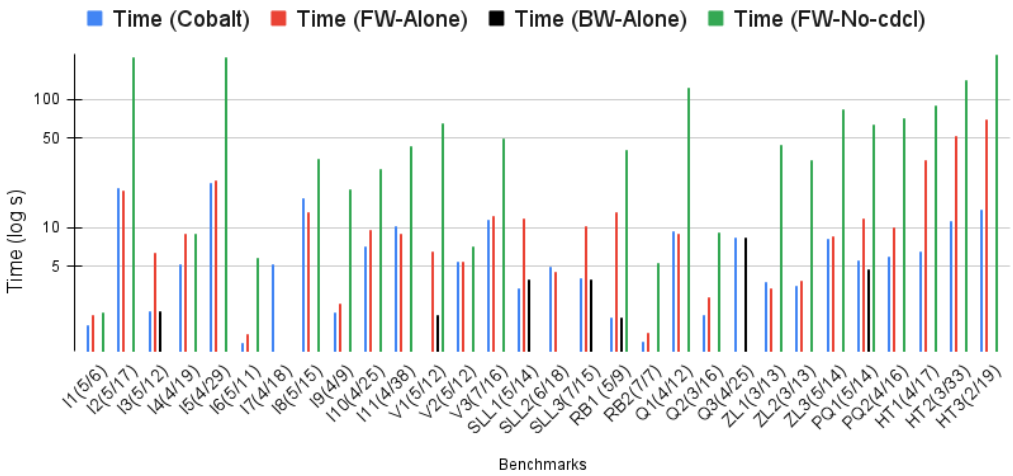
To actually define queries over these annotated libraries, we adopted a mix of methodologies: For some benchmarks, we directly use the verification task defined by the authors and translate it to its synthesis dual. For example, Figure 2 and benchmarks I10 and I11 are direct verification queries given in [Sekiyama and Igarashi 2017]. Similarly, the *Firewall* example (D6) for deleting network devices is translated directly from the verification queries provided in [Itzhaky et al. 2017]. Additionally, we also manually defined queries using real-life scenarios and textbook examples, e.g. extracting read articles from a *Newsletter*, while ensuring that the library protocol is followed (D5), replacing one device with another in a firewall as a central device (D8), etc. We also created several such real-life scenarios for databases and textbook examples over imperative data-structure libraries including inserting multiple elements in a hashtable, adding elements in a queue maintaining uniqueness, etc. For parser examples, we relied directly upon specifications associated with known data-dependent grammars for parsers, e.g. a PNG chunk that must satisfy a length-payload dependence, is given as specification query P1.



(a) Database Benchmarks



(b) Parsing Benchmarks



(c) Imperative Data Structure Benchmarks

Fig. 10. Synthesis time in seconds for Cobalt (T Cobalt) and uni-directional approaches (Time (BW-alone)) and (Time (FW-alone)) and a naive forward synthesis (Time (no-cdcl)). The horizontal axis enumerates different synthesis queries. Benchmarks for which a bar does not appear for a given approach indicate that the synthesis problem was not solvable within a 10 minute time-bound. Graphs are given in log-scale. The ratio of size of query specification, to the size of synthesized expressions in terms of the number of AST nodes is given within parentheses on the labels of the x-axis.

5.3 Results

The figures show synthesis times in seconds (the y-axis is in log scale) executing on a standard Intel laptop with 16GB RAM. All queries were executed with a time-out limit of 10 mins and a bound k=5. The timings are for four different instantiations: the blue bar shows timings for Cobalt (with bidirectional synthesis + CDCL-learning); the red bar (FW-alone) shows times with backward synthesis disabled, but with CDCL-leaning enabled; c) the black bar (BW-alone) shows times for just backward synthesis, with forward synthesis and CDCL disabled; and, d) the green bar shows the synthesis time for a naive forward alone synthesis without the CDCL learning component. Benchmarks with no corresponding bar indicate that the particular instantiation could not find a solution (i.e., it either timed-out or got stuck). Each benchmark label along the horizontal axis has an associated numeric value in parenthesis indicating the size of the synthesized result for that query in terms of number of AST nodes, e.g. D1 (13).

5.3.1 RQ_1 *and* RQ_2*: Effectiveness and impact of design decisions.* Our results show that Cobalt was successfully able to synthesize component-based programs for all the benchmarks considered.[8].) Overall synthesis times for all benchmarks take less than one minute, with approximately 32/47 completing in less than 10 seconds. The variance in synthesis times is primarily due to the number of quantified variables that must be instantiated in queries supplied to Z3. The complexity of these generated formulae are in turn dependent on the complexity of method specifications and synthesis queries, and the specificity of the expected return type. More significantly, the chart also reveals that bi-directional synthesis can solve queries that are not solvable using just the FW/BW-alone synthesis approaches.

Five of the benchmarks we consider (D3-D5, I7, P5) were unable to be solved using either forward- or backward synthesis within the given time bound. Using just a backward synthesis (BW-alone) method fails to find solutions for 34/47 queries, while disabling goal-directed search (FW-alone) fails to find a solution in 6/47 queries. Finally, the naive forward alone synthesis (FW-no-cdcl) was unable to find a solution for 14/47 queries. For the 41 queries FW-alone is able to solve, Cobalt is on average 2x faster, justifying the benefits of our bi-directional synthesis strategy over a unidirectional synthesis with CDCL.

There are 8 queries for which FW-alone can find a solution but a naïve synthesis without the learning component (FW-no-cdcl) failed to find a solution. For the remaining queries where both succeed in finding a solution, the FW-no-cdcl is on-average 6x slower than the CDCL version, justifying the benefits of using a CDCL mechanism as part of a forward search procedure.

```
1  λ (d : device) (x : device).
2    b1 ← is_device x;
3    if (b1)
4      b2 ← is_central x;
5      if (b2) then
6        _ ← delete_device d x;
7        ret ()
8      else
9        _ ← make_central x;
10       _ ← delete_device d x;
11       ret ()
12   else
13     _ ← add_device x;
14     _ ← make_central x;
15     _ ← delete_device d x;
16     ret ()
```

Fig. 11. Synthesized Program for query the D11

Synthesized Programs. The size of synthesized programs (given in parentheses along with the benchmark name on the x-axis) range between 6 to 38 AST nodes. These programs include function calls, conditional control-flow, constructors applications, etc. The number of components (continuous chain lengths) across synthesized programs, range from 2 to 7, comparable to other component-based-synthesis systems such as [Feng et al. 2017b] (Fig. 8) or [Guria et al. 2021] (Table-1)].

As an example of the output Cobalt generates, Figure 11 presents the synthesis result for query D11, which asks to synthesize a program that, given a globally shared *Firewall* database, and two devices: d, a *central* device, and x, deletes d and makes x as the central device. The conditional branches (lines 3, 12) distinguish cases when we need to add x to the database before deleting d. Similarly, the nested conditional (5, 8) distinguishes cases when we can directly delete d (if x is *central*) or when we need to first make x a *central* device.

Utility and Specification Efforts. Each benchmark label along the horizontal axis also has an associated ratio (p/q) in parenthesis, where p is the size of the query specifications in terms of the number of conjuncts in the specification, and q is the size of the synthesized result for that query in terms of the number of AST nodes. E.g., the label D1 (5/13) implies that Cobalt given query D1, a table insertion query whose specification has five conjuncts, produces a synthesized program with 13 AST nodes.

[8]The synthesis time using Cobalt for V1 (Figure 10c) is 1.1 seconds and is thus not visible on the presented log scale.

These ratios highlight that for simple programs, the size of the synthesized programs is comparable to the size of the specifications. However, for programs with intricate control flows found in some of the database queries (e.g., D9 and D10) or conditional queries found in some of the imperative data structure benchmarks (e.g., Q3 and HT2), queries are simpler because their preconditions are weaker. At the same time, the synthesized programs generated are more complex, especially highlighting the power of Cobalt's efficient enumerative search with limited specification.

Although Cobalt performs well on this traditional metric, we found that writing such programs from scratch (even in OCaml), without the use of libraries would typically involve non-trivial complexity with intricate control flows, loops, recursion, etc. For instance, in the absence of component-based synthesis support, synthesizing a program for the query in Figure 2 would need to synthesize code/auxiliary functions for tasks like table insertion, checking membership, taking the average etc. This makes it challenging to apply state-of-the-art deductive synthesis techniques directly to our queries, given that synthesizing auxiliary functions with these complex features in an effectful setting remains very much an open problem [Polikarpova and Sergey 2019]. Thus, a more reasonable and precise assessment of Cobalt's capabilities would involve comparing the complexity of defining queries with the complexity of the overall function synthesized, i.e. the combined size of the synthesized code plus the size of each library function used in the code.

In summary, these results support our two main claims: (1) a bi-directional synthesis strategy is beneficial to reason over effectful libraries - unlike Cobalt, neither FW-alone nor BW-alone could successfully discharge all the synthesis problems in our benchmark suite; note that at least one benchmark in each application class failed to be solved by either uni-directional method, indicating that our technique is not specialized to a particular application class. And, (2) CDCL learning in this setting is demonstrably useful - since FW-alone is also equipped with CDCL, its execution times are competitive with Cobalt for the benchmarks it completes. We note that disabling CDCL in FW-alone causes at least an order of magnitude increase in synthesis times while more than doubling the number of failing benchmarks.

5.3.2 RQ_3 *: Sensitivity to specification complexity and library size.* Synthesis complexity (and hence synthesis times) is dominated by the complexity of the queries discharged to Z3. Synthesis time increases as function specifications and queries become more complex, where complexity of specifications is directly correlated with the number of uninterpreted functions and variables in the query and number of conjuncts in propositional formulas.

Case Study. To understand the impact of specification complexity on synthesis capability, we compared the synthesis times for queries D1-D11 using its provided specifications, comparing it against the synthesis times taken when additional qualifiers are added to these specifications. For instance, the *Newsletter* benchmark has three qualifiers in its original specification *viz.* nlmem (a membership qualifier), subscribed (a Boolean-valued subscription function) and confirmed (a Boolean-valued function, indicating if the user has confirmed an action). To these, we additionally include the following four new qualifiers in a new variant of the benchmark: activenl (a Boolean-valued function that is true if a newsletter has at least one active subscription), activeuser (a Boolean-valued function capturing if a user has at least one active subscription), subsize (an integer-valued function that gives the number of newsletters a user is subscribed to) and nlreach (the number of users which are subscribed to a newsletter).

Figure 12 shows the specification for a library function subscribe, which takes a newsletter n and a user u and sets the subscription of the user for the newsletter to true and a synthesis query (goal) to synthesize a program which returns the list of articles read by u in n and then unsubscribes the user from the newsletter. The original specification and the query is shown in **black**; the modified

```
subscribe : (n : nl)→ (u :user) →
    { nlmem (D , n , u) = true ∧
      confirmed (D, n, u) = true ∧
      subscribed (D, n, u) = false
    } v : unit
    {
      nlmem (D', n, u) = true ∧
      subscribed (D', n, u) = true ∧
      confirmed (D', n, u) = false ∧
     subsize (D', u) == subsize (D, u) +
          1 ∧
          nlreach (D', n) == nlreach (D,
          n) + 1
    }
```

```
goal : (n : nl)→ (u :user) →
    {
      nlmem (D , n , u) = true ∧
      subscribed (D, n, u) = true ∧
      confirmed (D, n, u) = false ∧
      activenl (D, n) = true ∧activeuser
          (D, u) = true ∧subsize (D, u)
          > 0 ∧nlreach (D, n) > 0
    } v : [string]
    {
      v = articles (D') ∧ nlmem (D', n, u) = false
          ∧
      activenl (D, n) = true ∧subsize
          (D', u) == subsize (D, u) - 1
          ∧nlreach (D, n) == nlreach (D,
          n) - 1)
    }
```

Fig. 12. Effectful specifications for a Newsletter library function and a synthesis query goal. Shaded specifications are additional properties that were added to the original to assess the Cobalt's sensitivity to specification complexity and size.

variant includes the original formulas plus the new conjuncts (shown in gray). In a similar fashion we also define revised specifications for the *Firewall* libraries and its associated queries (D6-D11).

Figure 13 shows two line graphs comparing the time for the original run (Time (original)) compared to the time taken to synthesize a result when these new qualifiers are added to specifications and queries (Time (double qualifiers)). Synthesis times increase from 0% to a maximum of 26% (case D7); for most other cases, the increase is less than 20%, an indication that Cobalt's synthesis strategy scales reasonably well against specification complexity.

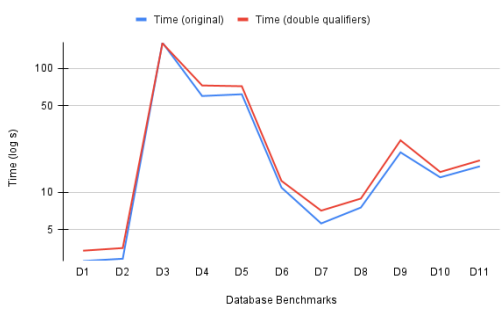


Fig. 13. Running time comparison between original Database benchmarks against doubling the number of qualifiers in specifications.

5.3.3 RQ_4 : *Comparison to other enumerative and deductive component-based synthesis techniques.*

Comparison with other type-directed, component-based enumerative synthesis approaches. To address how Cobalt compares against other systems, we consider the effectiveness of type and example-based synthesis approaches [Feng et al. 2017a,b; James et al. 2020b] in solving effectful queries, using example demonstrations instead of specifications to guide the synthesis procedure. For instance, the query,

(l : int list) → (i : int) → { true } (v : int list) { size (v') = size (l) + 1 }

can be synthesized using type-and-examples by giving the type, (int list → int list), and a set of input-output examples, e.g., i) (l=[1; 2], i=3 , output=[1;2;3]) ii) (l=[1], i=1, output=[1,1]).

TYGAR [Guo et al. 2019] is a type-directed component-based tool that operates over polymorphic Haskell data-types and components. We conducted an experiment on an extension of TYGAR, named Hoogle+ [James et al. 2020b] that allows using examples to further guide the TYGAR

synthesis process. To perform our comparison, we modeled the Table datatype used in our running example as a functional list over an abstract type, erasing effect annotations from each of the libraries, and making sure to include suitable libraries that were available in Hoogle+.

To simplify things further, we also modified the original query to just return a new table (rather than the original Pair value) as follows:

```
goal: (tbl : table) → (s : a) → {True} v : table
    {sel (h, tbl) = Tbl ∧ sel (h', tbl) = Tbl' ∧ mem (Tbl', s) ∧ size (Tbl') = size (Tbl) + 1};
```

We translated the Cobalt query above to the following Hoogle+ query:

```
goal : (tbl : table) → (s : a) → table
```

Running Hoogle+ on this query returns the following synthesized term:

```
goal = λ (tbl : table) (s : a) . (add_tbl s)
```

This program is unsound given the original interface of the add_tbl function since it can violate the uniqueness invariant of the table, a property enforced by the library via the precondition (not (mem (Tbl, s)) of the add_tbl function.

To refine this result, we next supplied input-output examples to Hoogle+ to help guide it to find the required (sound) solution. Some of the examples provided included:

```
1  Input : tbl = [], s = 'b' ; Output : ['b']
2  Input : tbl = ['b'], s = 'b' ; Output : ['b';'c']
3  Input : tbl = ['a'], s = 'b' ; Output : ['a';'b']
4  Input : tbl = ['a';'c'], s = 'b' ; Output : ['a';'c';'b']
5  Input : tbl = ['a';'b'], s = 'b' ; Output : ['a';'b';'d']
```

Unfortunately, these examples were ineffective in helping Hoogle+ to find a solution. This is because, although examples are effective at capturing structural properties like ordering, size or reformatting of inputs, they are not very useful in defining logical cumulative properties like membership or its negation. The input-output pairs at lines 2-5 above try to capture such a property, but fail to do so as the synthesizer has no way of knowing that the new elements inserted (i.e. 'c', 'd') are related to the input table by the property *not a member* or are intended to be just another character. This example illustrates the difficulty in relating the shape and contents of an input-output example to a provided logical specification, especially when these specifications capture effectful behavior.

Comparison with specification-guided heap manipulating program synthesis. A direct comparison with other heap- and effect-aware synthesis tools [Itzhaky et al. 2021b] like Suslik [Polikarpova and Sergey 2019] or Cypress [Itzhaky et al. 2021a] is not feasible because of fundamental differences in approaches and goals. For example, Suslik supports queries over separation-logic formulas with limited support for component-based synthesis, and limited expressiveness to specify effectful but non-separation specifications. Conversely, Cobalt defines a specification language for reasoning over components with non-trivial effectful semantics and rich qualifiers, but does not support separation logic formulas that capture fine-grained sharing and aliasing properties of the heap.

These differing capabilities are in service of differing goals: Suslik aims to synthesize recursive, pointer-manipulating programs from inductive specifications using the shape properties expressed in these specifications. Cobalt, on the other hand, uses pre/post specifications of effectful libraries to guide a component-based synthesis procedure for synthesizing non-recursive (albeit conditional) programs for complex, effectful (albeit non-separation) specifications that do not appeal to sophisticated shape properties.

These differences pose major technical challenges in running such tools on our benchmarks. For example, in theory, our queries correspond to non-spatial specifications in Suslik. However, both our queries and specifications allow rich formulas with qualifiers/method-predicates like mem, size,

etc. Unfortunately, such specifications are beyond what is currently supported for pure (non-spatial) formulas in Suslik, which only supports qualifiers over a simple theory of linear arithmetic. This limitation is discussed by the authors in follow-up work [Itzhaky et al. 2021b], Sec 4.2.

To attempt to better quantify these differences, we translated the Cobalt synthesis problem (Sec. 3.1) to separation-logic (spatial) formulas in Suslik because spatial formulas do allow method predicates; each of these, however, must be given a logical interpretation. We then ran Suslik on this translated problem.

For example, we translated the Cobalt synthesis problem given in Figure 2b to a Suslik query as follows. We first define a unique list (ulist) that is a singly-linked list with unique elements to model the table data structure.

```
predicate ulist(loc x, set s) {
    | x == 0 => { s == {} ; emp }
    | not (x == 0) => { s == {v} ++ s1 ∧ not (v in s1); [x, 2] ** x :→ v ** (x + 1) :→ nxt ** ulist(nxt, s1) }
}
```

We then define two qualifiers sll_mem and sll_len over the table as inductive separation-logic formulas:

```
predicate sll_len(loc x, int len) {
    | x == 0 => { len == 0 ; emp }
    | not (x == 0) =>
        { len == len1 + 1; [x, 2] ** x :→ v **
          (x + 1) :→ nxt ** sll_len(nxt, len1) }
}
```

```
predicate sll_mem(loc x, int str, set s, bool mem){
    | x == 0 => { s == {} && false; emp }
    | not (x == 0) =>
        { s == {v} ++ s1 && (str == v || mem1);
          [x, 2] ** x :→ v ** (x + 1) :→ nxt
          ** sll_mem(nxt, str, s1, mem1) }
}
```

We next define a library of functions using ulist and these qualifiers in terms of separation formulas. Finally, we took Cobalt queries and translated these to Suslik queries; for example, the functional query specification shown in Figure 2b can be written as follows:

```
void goal (loc r, loc ret)
{ r :→ x ** ret :→ val ** ulist (x, s) ** sll_len (x, n) }
{(mem == true) ∧ n1 == n + 1; r :→ y ** ulist (y, s1) ** sll_mem (y, val, s1, mem) ** sll_len (y, n1)}
```

Table 1. Results of running selected Cobalt queries on Suslik.

Benchmark	Cobalt	Suslik
I1	✓	err
I4	✓	t/o
I5	✓	t/o
I6	✓	err
I9	✓	t/o
I10	✓	t/o
I11	✓	t/o
V1	✓	err

A formula in Suslik has two components $\{\ \phi;\ P\}$; a non-separation formula (authors call it a *pure* formula) ϕ and an *impure* component P possibly containing separation formulas. P contains *points-to* specification (written as x :-> y) and separating conjuncts (written as ($P1$ ** $P2$)).

The above query's post-condition thus requires that the size of the output ulist (represented by y), which is pointed-to by r, has size one greater than the input ulist (represented by x), and the returned value (represented by val) is present in y.

On this query, the Suslik tool timed out with a timeout of 30000 seconds[9] (approx. 8 hours 20 minutes). We ran a similar experiment on several other Cobalt benchmarks wherever it was possible for us to define an inductive separation-logic predicate corresponding to the qualifiers in our specifications for the data-structure used in the benchmarks. Table 1 shows the results for these benchmakrs. For each of these experiments, Suslik either timed-out (t/o) or

[9]This is the timeout value set in the online version of the tool.

generated a program with an error expression, a result indicating an inconsistent internal state encountered while solving the query.

These experiments give anecdotal evidence about our claim of differing capabilities and goals. We conjecture the timeout happens because Suslik hides complexities of the function call rule into a call-abduction routine. This routine prepares the current heap (say H) to take it to another heap (say H') such that the precondition of a function f holds in H'. Thus, Suslik uses an abduction (forward search) procedure for resolving multiple function calls. While this may be sufficient in the case of a single recursive function call, its generalization to handle multiple library functions requires a multi-abduction decision procedure [Albarghouthi et al. 2016] to resolve, a challenging problem in the presence of expressive data structures of the kind used in our benchmarks [Zhou et al. 2021].

5.4 Limitations

Cobalt relies on automated verification of the programs for its forward, backward as well as the CDCL search procedures. This reliance requires Cobalt to make fundamental assumptions about library behavior that (a) each heap object (OCaml reference) is always referenced by a unique path (e.g, a variable x or a field access x.f.y) and, (b) that there is no sharing of heap objects. This allows us to reason locally and automatically about effectful behavior of library functions without the need for reasoning over separation-logic formulas. In practice, this restriction prevents us from synthesizing programs from libraries that do require heap sharing, for example, those that implement cyclic data structures, graphs, etc. We leave incorporating approaches such as [Piskac et al. 2013; Qiu et al. 2013] that enable some degree of automated verification for programs specified using separation formulas into our synthesis pipeline as a topic for future work.

6 RELATED WORK

Deductive Program Synthesis. Related deductive synthesis approaches to ours include Suslik [Polikarpova and Sergey 2019] and its follow-up work Cypress [Itzhaky et al. 2021a], both of which take Hoare-triple style specifications and have synthesis rules for function calls. As we have described in the previous section, an important difference between Cobalt and Suslik specifications stems from the form of Suslik pre- and post-conditions, $\{\phi; P\} \rightarrow \{\phi' ; Q\}$, that are expressed using two components - a non-separation part ϕ defining constraints on the logical data structure associated with the actual mutable data structure, and a *spatial* part defining assertions related to the shape of the heap. The pre- and post-formulas in Cobalt specifications are analogous to the non-separation part of Suslik specifications; our specification framework has no corresponding analog to Suslik's spatial component. For non-spatial properties, however, our queries, and consequently our specifications, enable rich formulas with qualifiers, expressivity that is beyond what is currently supported for pure formulas in Suslik, which only supports a simple theory of linear arithmetic. As our experimental comparison results show, these significant differences lead to fundamentally different synthesis strategies making any kind of direct comparison infeasible. We note that Suslik does allows libraries of functions through their (Abduce Call) rule which is a naïve call-abduction routine that is equivalent to Cobalt's *no-cdcl* approach in theory.

Other deductive synthesis [Delaware et al. 2015, 2019; Kneuss et al. 2013; Polikarpova et al. 2016] and proof-search guided synthesis efforts [Frankle et al. 2016; Osera and Zdancewic 2015] use deductive proof rules for synthesizing pure terms. In contrast, our synthesis rules operate over heap manipulating expressions (*viz.* effectful function calls and sequencing). Synquid [Polikarpova et al. 2016] uses a bi-directional typing calculus to synthesize functional programs. However their notion of bi-directionality is related to bi-directional typing [Dunfield and Krishnaswami 2021],

which is unrelated to the notion of bi-directionality used in Cobalt that is defined with respect to forward and backward proof search over programs with effectful specifications [Nanevski et al. 2006, 2008; Swamy et al. 2013]. Viser [Wang et al. 2019] uses a light-weight bidirectional abstract interpretation coupled tightly to table transformation tasks that can reason specifically about simple table inclusion constraints; however, it cannot handle general effectful specifications of the kind intended to be used in Cobalt.

Component-based Synthesis and Learning. There is a long line of work on the use of component-based synthesis in the context of domain-specific languages [Feng et al. 2017a; Jha et al. 2010] as well as general-purpose programming domains [Feng et al. 2018, 2017b; Guo et al. 2019; Guria et al. 2021; Shi et al. 2019; Wang et al. 2019]. Cobalt is distinguished from these other systems in the form of the query specifications that we consider (formal query specifications expressed as Hoare triples vs. input-output or informal specifications as found in [Feng et al. 2018, 2017b; Guria et al. 2021; Shi et al. 2019]); and, in our expectation that the libraries from which synthesized programs are constructed are effectful, in contrast to the assumptions made in [Feng et al. 2018; Guo et al. 2019]. Although Sypet [Feng et al. 2017b] does support effectful libraries, it does not use library protocol specifications to guide synthesis and consequently cannot enforce associated library protocols or programs that require conditional control-flow.

Our CDCL-learning based enumeration is similar in spirit to the conflict-driven learning based synthesis of pure components [Feng et al. 2018], and is inspired by conflict-driven learning based enumeration techniques found in modern SAT solvers [Biere et al. 2009]. The discriminating propositions learned in Cobalt, however, must include the path-sensitive, effectful semantics of failed programs in terms of the strongest postconditions associated with these programs, in contrast to the simpler, global set of Boolean propositional formulas associated with the partial failed programs found in [Feng et al. 2018], such a global formula in presence of effectful libraries with path-sensitive information will grow too large and overwhelm the solver. Further, we must also account for program failures due to bounded exploration of an unbounded search space without losing completeness, a challenge [Feng et al. 2018] does not face.

7 CONCLUSIONS

We present a new specification-guided synthesis framework capable of synthesizing effectful programs from a library of effectful components. We capture the behavior of these components using a rich specification language that capture effectful behavior in terms of Hoare-style pre- and post-conditions. The synthesis procedure itself combines forward and backward proof search with respect to these specifications, integrating a CDCL-style learning framework to enable scalability. Experimental results on a tool (Cobalt) that integrates these ideas are promising, demonstrating Cobalt's ability to efficiently synthesize programs over complex effectful queries, guaranteed to be consistent with component specifications.

8 DATA AVAILABILITY STATEMENT

The Cobalt tool and the artifact used in the paper is available in [Mishra and Jagannathan 2022a]

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their detailed comments and suggestions. Funding for this work material is supported in part by DARPA, under the Safe Documents (SafeDocs) program.

REFERENCES

Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal Specification Synthesis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 789–801. https://doi.org/10.1145/2837614.2837628

Andrew Appel, Lennart Beringer, and Qinxiang Cao. 2021. . Software Foundations, Vol. 5. Electronic textbook, Chapter Verifiable C. Version 1.1.1, `http://softwarefoundations.cis.upenn.edu`.

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.

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

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.

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 689–700. https://doi.org/10.1145/2676726.2677006

Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: Correct-by-Construction Derivation of Decoders and Encoders from Binary Formats. *Proc. ACM Program. Lang.* 3, ICFP, Article 82 (July 2019), 29 pages. https://doi.org/10.1145/3341686

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5, Article 98 (may 2021), 38 pages. https://doi.org/10.1145/3450952

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

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017a. 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

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017b. 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

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-Directed Synthesis: A Type-Theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 802–815. https://doi.org/10.1145/2837614.2837629

Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2004. DPLL(T): Fast Decision Procedures. In *Computer Aided Verification*, Rajeev Alur and Doron A. Peled (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 175–188.

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

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

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

Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. 2021a. Cyclic 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, 944–959. https://doi.org/10.1145/3453483.3454087

Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. 2021b. Deductive Synthesis of Programs with Pointers: Techniques, Challenges, Opportunities. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12759)*, Alexandra

Silva and K. Rustan M. Leino (Eds.). Springer, 110–134. https://doi.org/10.1007/978-3-030-81685-8_5

Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. 2020a. 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

Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. 2020b. 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

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

Trevor Jim, Yitzhak Mandelbaum, and David Walker. 2010. Semantics and Algorithms for Data-Dependent Grammars. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 417–430. https://doi.org/10.1145/1706299.1706347

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo Recursive Functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 407–426. https://doi.org/10.1145/2509136.2509555

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

J. McCarthy. 1993. *Towards a Mathematical Science of Computation*. Springer Netherlands, Dordrecht, 35–56. https://doi.org/10.1007/978-94-011-1793-7_2

Ashish Mishra and Suresh Jagannathan. 2022a. *Cobalt (OOPSLA 2022 Artifact): Code and Benchmarks*. https://doi.org/10.5281/zenodo.7065694

Ashish Mishra and Suresh Jagannathan. 2022b. Specification-Guided Component-Based Synthesis from Effectful Libraries. https://doi.org/10.48550/ARXIV.2209.02752

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2006. Polymorphism and Separation in Hoare Type Theory. (2006), 62–73. https://doi.org/10.1145/1159803.1159812

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: Dependent Types for Imperative Programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 229–240. https://doi.org/10.1145/1411204.1411237

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2737924.2738007

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating Separation Logic Using SMT. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 773–789.

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

Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 72 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290385

Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and Parthasarathy Madhusudan. 2013. Natural Proofs for Structure, Data, and Separation. 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, 231–242. https://doi.org/10.1145/2491956.2462169

J. C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74.

Taro Sekiyama and Atsushi Igarashi. 2017. Stateful Manifest Contracts. 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, 530–544. https://doi.org/10.1145/3009837.3009875

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

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

Philip Wadler and Peter Thiemann. 2003. The Marriage of Effects and Monads. *ACM Trans. Comput. Logic* 4, 1 (Jan. 2003), 1–32. https://doi.org/10.1145/601775.601776

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

Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. 2001. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design* (San Jose, California) *(ICCAD '01)*. IEEE Press, 279–285.

Zhe Zhou, Robert Dickerson, Benjamin Delaware, and Suresh Jagannathan. 2021. Data-driven Abductive Inference of Library Specifications. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29. https://doi.org/10.1145/3485493

Received 2022-04-15; accepted 2022-09-01