Synthesizing Data-structure Manipulations from Storyboards

Rishabh Singh
MIT CSAIL
Massachusetts, USA
rishabh@csail.mit.edu

Armando Solar-Lezama
MIT CSAIL
Massachusetts, USA
asolar@csail.mit.edu

ABSTRACT

We present the *Storyboard Programming* framework, a new synthesis system designed to help programmers write imperative low-level data-structure manipulations. The goal of this system is to bridge the gap between the "boxes-and-arrows" diagrams that programmers often use to think about data-structure manipulation algorithms and the low-level imperative code that implements them. The system takes as input a set of partial input-output examples, as well as a description of the high-level structure of the desired solution. From this information, it is able to synthesize low-level imperative implementations in a matter of minutes.

The framework is based on a new approach for combining constraint-based synthesis and abstract-interpretation-based shape analysis. The approach works by encoding both the synthesis and the abstract interpretation problem as a constraint satisfaction problem whose solution defines the desired low-level implementation. We have used the framework to synthesize several data-structure manipulations involving linked lists and binary search trees, as well as an insertion operation into an And Inverter Graph.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; I.2.2 [**Artificial Intelligence**]: Program Synthesis

Keywords

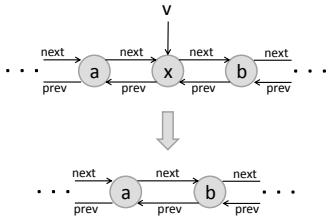
Program Synthesis, Storyboard Programming, Data structure manipulations

1. INTRODUCTION

When implementing complex data-structure manipulations, programmers must bridge the gulf that separates their intuitive understanding of how the manipulation works from the low-level code that actually implements it. The intuition behind these manipulations often takes the form of "boxes-and-arrows" diagrams which programmers might draw on a whiteboard to illustrate the evolution of the data-structure as it is manipulated. Unfortunately the visual intuition reflected in these diagrams is absent from the low-level pointer updates that make up the implementation. This semantic gap between intuition and implementation makes data-structure manipulations difficult to write and maintain.

To illustrate this gap, consider the example in Figure 1. Part (a) of the figure shows a graphical description of the removal of a node from a doubly linked-list. The diagram—we call it a *storyboard*—communicates very clearly the effect of the manipulation. By contrast, the imperative code in Figure 1(b) is short but not self-explanatory; understanding this code essentially requires one to mentally recreate the image from the storyboard in part (a). The system presented in this paper bridges this gap by using constraint-based synthesis technology to automatically implement data-structure manipulations that are provably correct with respect to high-level descriptions like the one illustrated by Figure 1.

a) Graphical intuition:



b) Code:

```
void dllRemove(Node v){
    v.n.p = v.p;
    v.p.n = v.n;
}
```

Figure 1: Doubly linked list deletion example

Our system is not yet a graphical programming system—it lacks a graphical user interface—but it brings closer to reality the promise of an effective graphical programming model for data-structure manipulations. It does so by supporting a form of *Programming by Example* (PBE) [6] where the manipulations are synthesized from partial descriptions of their effects. The main input to the system is a set of descriptions of the state of a data-structure before and after manipulation—essentially text descriptions of storyboards like the one in Figure 1. Two important features, however, distinguish our system from traditional PBE systems. The

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

first is the ability to abstract away those parts of the data-structure that are not relevant to the manipulation, as is done in Figure 1 through the use of ellipsis. This form of abstraction gives our system a lot of expressive power, because it allows a single figure to succinctly describe the behavior of the algorithm on an infinite number of concrete inputs, turning a simple input-output pair into a partial specification. The second difference with PBE is that our system asks the user to provide information about the control-flow structure of the solution; this information reduces the space of possible implementations that the system needs to consider and makes it less likely that the system will produce a solution that only works for the given examples. These two features allow us to synthesize complex data-structure manipulations from relatively simple storyboards.

The system is made possible by a new synthesis algorithm that combines previous work on constraint-based synthesis [24, 29] with abstract interpretation. Our algorithm is not the first to do this [29], but it is the first to scale to the very large and complex abstract domains required to reason about data-structure manipulations. The key idea behind the algorithm is to use quantification to eliminate operations that require complex set-based reasoning, and to use the SKETCH solver [24] to solve these formulas(Section 5). The new synthesis algorithm allows us to combine constraint-based synthesis with a form of shape-analysis loosely based on TVLA [15]. The shape analysis algorithm used by our system is not as powerful as many of those found in the literature [21], but it is powerful enough to reason about most operations involving trees and lists. The strength of our particular form of shape analysis, however, lies in the ease with which we can take the abstractions expressed as part of the storyboard and use them as the basis for an abstract domain that is then used to verify each candidate implementation.

In summary, the key contributions of the paper are:

- Development of a new model of interaction between the synthesizer and the programmer, targeted at the domain of data-structure manipulations (Section 2).
- A novel synthesis algorithm that combines constraint-based synthesis with abstract interpretation to bridge the gap between high-level graphical specifications and their low-level implementation (Section 5).
- The development and evaluation of a tool to automatically synthesize data-structure manipulations involving linked lists and trees from high-level graphical specifications (Section 6).

So far, we have used our system to successfully synthesize several data structure manipulations such as insertion, deletion, search, reversal and rotation operations over singly linked list, doubly linked list and binary search tree data structures. We have also used our framework to synthesize small puzzle problems as well as some manipulations involving a tricky real-world And Inverter Graph [18] data-structure used in the ABC solver [3].

2. OVERVIEW: LINKED LIST REVERSAL

In this section, we present an overview of our framework through a textbook data-structure manipulation example: in-place linked list reversal. Reversing a list with a loop and using only a constant amount of additional memory is non-trivial; in fact, the algorithm for this manipulation is a common question in technical interviews. In the remainder of the section, we describe how the user describes this data-structure manipulation to the framework, and provide a high-level view of how the synthesizer uses this input to derive an implementation.

2.1 Storyboard

In order to synthesize the manipulation, our system takes as input a *storyboard* composed of three elements: a set of *scenarios*, each of which corresponds to an abstract input-output pair; a set of fold and unfold definitions, and a skeleton of the looping structure of the desired algorithm.

A scenario in our system is an input-output pair describing the effect of the manipulation on a potentially abstract data-structure, where abstraction is used to elide details of the data-structure that are not considered relevant. For example, Figure 2 below shows the main scenario describing the effect of reversing a linked list. Like the more informal

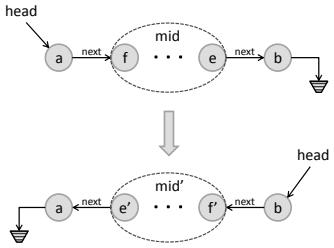


Figure 2: Graphical description of linked-list reverse

example in Figure 1, the scenario uses ellipses to abstract away part of the list. In our notation, however, the ellipses are formalized by using the concept of *summary nodes*. For this example, the scenario uses a summary node mid to represent the middle part of the list, which may vary in size for different runs of the algorithm. Out of all the nodes in the sub-list represented by mid, the first and last node deserve special attention because other nodes outside the sub-list mid may point to them. We call these special nodes *attachment points* of the summary node, and as we shall see, they play an important role in reasoning about scenarios.

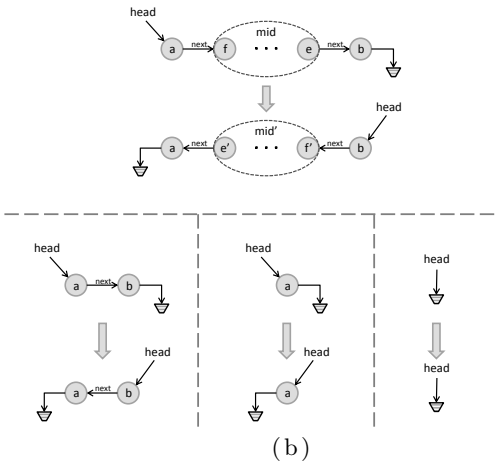
Figure 3 shows the complete set of scenarios needed for this example, including scenarios to describe the behavior of the algorithm on lists of length zero, one and two. Figure 3(a) presents the text notation used by our system to describe one of the scenarios on the right. The storyboard consists of an environment description and a sequence of scenario descriptions. The environment defines the set of variables used in the implementation, the set of fields in the objects that make up the data-structure (which are called *selectors* in the shape analysis literature), and the set of concrete and summary locations. For this example there are only four program variables (head, temp1, temp2 and temp3), one selector corresponding to the field next, and eight concrete locations a, f, e, b, f' and e'. The concrete locations f and e (resp. f' and e') represent the two attachment points of the summary node mid (resp. mid'). The environment description also states a global invariant that the next pointers of locations f and f' point to mid and mid' respectively with a value of 1/2 in a 3-valued logic similar to TVLA [21]. Each scenario description, in turn, uses variable predicates, se-

Environment
vars Node head, temp1, temp2, temp3
selectors (Node,Node) next
locations Node a, f, e, b, f', e'
summary Node (mid,f,e), (mid',f',e')
invariant next(f,mid)=1/2, next(f',mid')=1/2

Scenario 1
input
head(a) //vars
next(a,f),next(e,b),next(b,null) //sels

output
head(b) //vars
next(b,f'), next(e',a), next(a,null) //sels

(a)



(b)

Figure 3: Storyboard for in-place linked list reversal

lector predicates and data constraint predicates to define a pair of input-output state configurations. The variable predicates of the form var(loc) denote that variable var points to location loc, and the selector predicates of the form sel(loc_1, loc_2) denote that the sel field of location loc_1 points to location loc_2. The data constraint predicates define additional constraints involving the data values of locations.

In order to make scenarios precise, it is often necessary to provide additional information about the structure of summary nodes. For example, in Figure 2, the summary node mid represents a set of nodes with a very particular structure; specifically, the scenario only makes sense under the assumption that node e is reachable from node f. Our system allows the user to provide this structural information through fold and unfold predicates. For example, Figure 4 shows the fold and unfold predicates used to describe the recursive structure of the mid summary node. The predicates describe the structure of summary nodes in terms of their attachment points, in a similar way as Fradet *et al.* [7] used context-free graph grammars to describe shape types.

The exact syntax and semantics of the predicates will be discussed in detail in Section 4.1; for now, it is enough to understand that the predicates in Figure 4(a) are precise text representations of the recursive definitions illustrated in Figure 4(b). For example, the unfold rule shows two alternatives for the summary node mid: either the attachment points f and e are actually the same node x', and this is the only node in mid, or f is a node x' whose next pointer points to the attachment point f' of another similar summary node mid'. For this example, fold is an inverse of unfold and could be derived automatically, but Section 4.1 will show other examples where it is useful to define fold and unfold independently as a way to guide the solver to a specific solution.

The scenarios and the fold and unfold definitions together describe the effects of the desired manipulation, but recall that our running example included some non-functional requirements, such as the requirement that the implementation use of a single loop. This requirement is expressed by providing a skeleton of the looping structure of the desired algorithm. Figure 5 shows the skeleton for list reverse, which states that the implementation should contain exactly one while loop with some blocks of code before and after it. The code produced by the synthesizer will have each of these blocks replaced by a series of guarded assignments of the form if(COND) then STMT. The conditional COND corresponds to expressions EXP op EXP, where EXP is either an expression of the form var(.sel?) or null and op ranges over the set of comparison operators. The set of assignment statements STMT corresponds to assignments of the form LHS = EXP where LHS is the same set as EXP but excluding null.

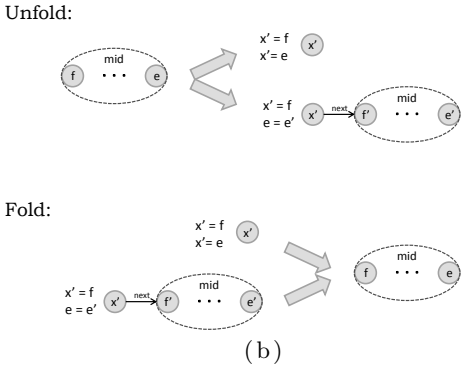
Each of these elements—the scenarios, the loop skeleton and the fold and unfold definitions—comprise a partial specification of the desired function. For example, the storyboard above is a partial specification because the abstraction does not define the relationship between mid in the input and mid' in the output. In this case, asking the synthesizer to produce a solution with a small number of statements is sufficient to ensure the correct answer, but sometimes the user may have to provide the system with additional information. In keeping with the PBE model, this additional information usually takes the form of additional scenarios with concrete examples, but it can also include providing intermediate state configurations, adding predicates in scenarios or providing a more detailed implementation sketch in place of the simple loop skeletons. The strength of our synthesis approach is that it can combine these different constraints into a concrete implementation. Moreover, as shown in Section 6, the constraints imposed by the storyboards are strong enough that in the few cases where the specification has to be strengthened, it only takes a few additional concrete scenarios or an intermediate state configuration to guide the framework to synthesize the correct implementation.

2.2 Synthesizing code from storyboards

Our synthesis strategy is based on constraint-based synthesis [24, 29]. The key idea behind this form of synthesis is to define a space of possible solutions to the synthesis problem and to represent it as a parameterized program. For example, in our case, each unknown assignment can be represented as a switch statement where an unknown parameter determines which assignment is actually performed. This means that the entire solution space can be represented as a program $P(in, c)$ where in is the input to the program and c is a vector of *control* parameters that determines which computations are performed. The goal of the synthesis process

unfold(f, {(f, in, x′), (e, out, x′)}, [true]);
unfold(f, {(f, in, x′), (e, out, e′)}, [x′.next = f′]);
fold(x, {(x, in, f′), (x, out, e′)}, [true]);
fold(x, {(x, in, f′), (e, out, e′)}, [x.next = f]);

(a)



(b)

Figure 4: Unfold and fold predicate definitions for mid summary node

```
void llReverse(Node head){

        /* 1 */

        while(/*2*/){/*3*/}


        /*4*/
        return head;
}
```

Figure 5: Control flow sketch for list reversal. Each number corresponds to an unknown block of code.

is then to find a value c_{sol} such that the program $P(in, c_{sol})$ meets the storyboard specification for all inputs in.

This view of the synthesis problem is common to all constraint based synthesis approaches [24, 29]. In our framework, we derive these constraints from the storyboard by interpreting the input-output states in each scenario of the storyboard as abstract states in a specially crafted abstract domain. This allows us to use the theory of abstract interpretation [5] to frame the correctness condition as a set of equations. The fold/unfold predicates can be understood in the context of abstract interpretation as a way of defining instrumentation predicates for summary nodes. Once the correctness condition has been framed as a set of equations, the translation to constraints follows with only a small amount of effort.

2.2.1 From storyboards to equations

Our framework uses a powerset domain as an abstract domain where the state is represented as a set of shapes. Each shape, in turn, is represented as a set of predicates, following a formalism similar to TVLA [15, 21]. Given this abstract domain, the next step is to use the control flow sketch to derive a set of equations relating the abstract states at program points as described by Sharir and Pnueli in [23]. In these equations, each one of the unknown blocks of code in the control flow sketch is represented by a parameterized transition function $F_m(t_i, c)$ that maps an abstract state to an output abstract state for a given control value c. The control flow sketch then defines a set of equations involving the transition functions. For example, the control flow sketch in Figure 5 induces the following equations, where the transition function F_i corresponds to block i in the sketch.

$$t_0 = F_1(in, c); \quad (1)$$
$$t_1 = F_3(t_0 \cup t_2, c); \quad (2)$$
$$t_2 = F_2(t_1, c); \quad (3)$$
$$t_3 = \overline{F_2}(t_0 \cup t_2, c); \quad (4)$$
$$out = F_4(t_3, c); \quad (5)$$

If these equations came from a verification problem, each of the F_i would be a transition function, and we could use an iterative approach to find the least fixed point solution to the equations and check that *out* matched the solution required by the storyboard. But for synthesis, each F_i represents an unknown block of code, and the c values we are interested in finding do not form a lattice, so we cannot use a standard iterative solver find c_{sol}. Instead, we provide an efficient way to encode the problem in a form that an off-the-shelf constraint solver can efficiently solve.

Transition Functions: Each transition function $F_m(t_i, c)$ encodes the behavior of set of possible sequences of guarded assignments that can be used to complete the control flow skeleton. In addition to conditional statements, a transition function can also include two special kinds of statements: unfold var and fold var. The semantics of unfold and fold correspond to no-ops in the concrete domain; their sole purpose is to allow the abstract interpretation to reach a fixed point while allowing for very precise reasoning within the abstract domain. Figure 6 shows the role of the unfold and fold functions in materializing abstract nodes [21] to allow for more precise reasoning, and then collapsing multiple abstract shapes into a single one to allow the abstract interpretation to reach a fixed point. The semantics of fold and unfold will be described in detail in Section 4 after we formalize the overall abstract interpretation framework.

2.2.2 Equations to Constraints

So far, we have described how to encode the problem in terms of the least fixed point solution of a set of equations. Our framework translates these equations into a 2QBF formula, which in principle could be solved by any 2QBF solver, although in practice we have found that only our SKETCH solver scales to large synthesis problems. As we will show in Section 5, the key idea in the translation to constraints is that instead of having the transfer functions produce sets of states, we make the functions non-deterministically produce one element in the set. Therefore, by exploring all

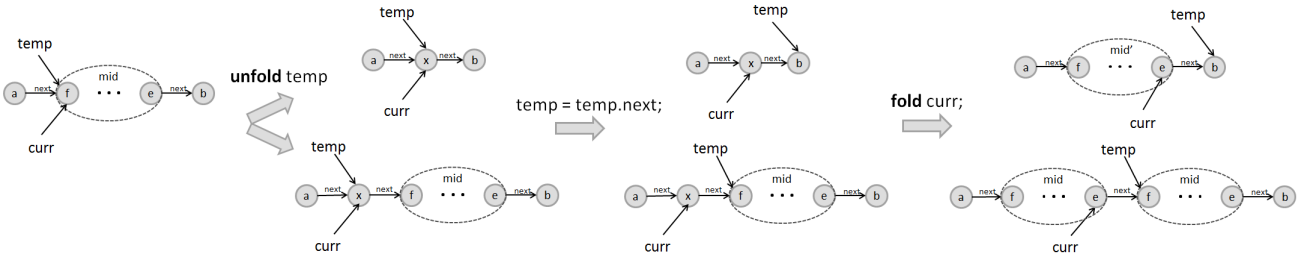


Figure 6: unfold and fold operations in action

```
Node llReverse(Node head){
  Node temp1 = null, temp2 = null;
  Node temp3 = null;

  temp1 = head;
  while(temp1 != null){
    // unfold temp1;
    head = temp1;
    temp1 = temp1.next;
    head.next = head;
    head.next = temp3;
    temp3 = head;
    // fold head;
} }
```

Figure 7: Synthesized list reverse implementation

non-deterministic choices, we guarantee that the entire set is considered.

2.3 Synthesized Implementation

From a satisfying assignment to the constraints, the framework derives the imperative implementation shown in Figure 7 for the linked list reverse manipulation. The conditionals (all true) from the conditional assignment statements have been removed for better readability of the code. It can be noted that the implementation did not use the program variable (temp2) and the loop body includes an extra *dead store* assignment statement (head.next = head). Aside from the additional assignment, the code is an efficient implementation of the desired algorithm, and the entire synthesis process takes only a couple of minutes.

3. DATA STRUCTURE CONFIGURATIONS

In this section we present the formalism used by our framework to encode and reason about the storyboard description. The formalism is based on abstract interpretation, and is similar to that of TVLA; the primary difference is our treatment of summary nodes with attachment points. But before we describe the abstract interpretation, we need to define the concrete domain over which programs operate. In this concrete domain, the state of the program is defined by a fixed set of local variables and a set of memory locations (also called nodes), where each location can have a number of fields pointing to other memory locations. Our framework currently does not support the allocation of memory by a synthesized routine, so the set of nodes that the program has to reason about does not grow as the routine executes.

Let $\mathcal{L}^{\#}$ represent the set of memory locations the synthesized program will operate on. Then, the state of the program is captured by two sets of predicates. First, for every variable v and location $l \in \mathcal{L}^{\#}$ there is a predicate $v(l)$ that indicates whether v points to location l. Then, for every field sel, there is a predicate $sel(l_1, l_2)$ that indicates whether a location l_1 has a field sel that points to location l_2. These two sets of predicates encode a *concrete shape* which defines the instantaneous configuration of the heap at any point in the execution.

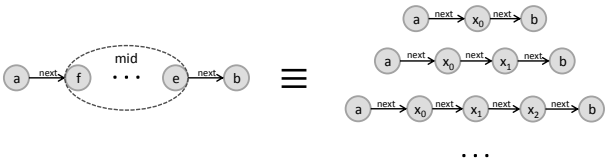


Figure 8: An abstract list representing infinite concrete lists

Shapes in the abstract domain.

The abstract domain consists of sets of *abstract shapes*, where each abstract shape itself represents a set of concrete shapes. The abstract shapes are defined in terms of a set of locations $\mathcal{L}$. Each location $loc \in \mathcal{L}$ can be either a summary location or a concrete location; we use the predicate $sm(loc)$ to indicate that loc is a summary location, so $\neg sm(loc)$ indicates that the location is concrete. As we have stated before, a concrete location loc may serve as an *attachment point* for a summary location u, which we express with the notation $loc \in \mathcal{A}(u)$; we use the predicate $apt(loc)$ to indicate the role of loc as an attachment point.

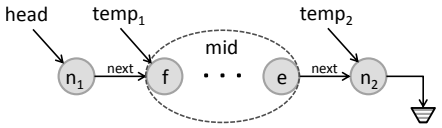


Figure 9: State configuration for a singly linked list

EXAMPLE 1. *The state configuration in Figure 9 is encoded as follows. The set of locations is given by* $\mathcal{L} = \{l_1, \mathsf{mid}, \mathsf{f}, \mathsf{e}, l_2\}$, *with a summary node* mid *and attachment points* $\mathcal{A}(\mathsf{mid}) = \{\mathsf{f}, \mathsf{e}\}$. *The set of program variables are* $\mathcal{V} = \{\mathsf{head}, \mathsf{temp}_1, \mathsf{temp}_2\}$. *The variable predicates* $\mathsf{head}(l_1)$, $\mathsf{temp}_1(\mathsf{f})$ *and* $\mathsf{temp}_2(l_2)$ *are true. The* next *selector predicates* $\mathsf{next}(l_1, \mathsf{f})$, $\mathsf{next}(\mathsf{e}, l_2)$ *and* $\mathsf{next}(l_2, \mathsf{null})$ *are true, and* $\mathsf{next}(\mathsf{f}, \mathsf{mid}) = 1/2$.

To make the definition of the abstract domain more formal, consider a concrete shape $S^{\#}$ with a set of nodes $\mathcal{L}^{\#}$, a selector predicate $\mathsf{sel}^{\#}$ and a variable predicate $\mathsf{var}^{\#}$, together with an abstract shape S with a set of nodes $\mathcal{L}$, selector sel and variable predicate var. We say that shape $S^{\#}$ is in the concretization of S ($S^{\#} \in \gamma(S)$) when there exists a relation $\mathcal{M} : \mathcal{L}^{\#} \times \mathcal{L}$ that satisfies the following conditions.

- Every node in $S^{\#}$ maps to some node in S and vice versa: *i.e.* $\forall l_1 \in \mathcal{L}^{\#}\ \exists n_1 \in \mathcal{L}\ s.t.\ \mathcal{M}(l_1, n_1)$ and $\forall n_1 \in \mathcal{L}\ \exists l_1 \in \mathcal{L}^{\#}\ s.t.\ \mathcal{M}(l_1, n_1)$
- Nodes that do not map to summary nodes map to a single concrete node: *i.e.* for any $l_1 \in \mathcal{L}^{\#}$, if $\neg\exists n_2\ s.t.\ sm(n_2) \wedge \mathcal{M}(l_1, n_2)$ then $\mathcal{M}(l_1, n_a) \wedge \mathcal{M}(l_1, n_b) \Rightarrow n_a = n_b$.
- Summary nodes do not overlap: *i.e.* $sm(n_a) \wedge \mathcal{M}(l_1, n_a) \wedge \mathcal{M}(l_1, n_b) \Rightarrow (n_a = n_b \vee n_b \in \mathcal{A}(n_a))$.
- Edges between concrete nodes are preserved: *i.e.* given $l_1, l_2 \in \mathcal{L}^{\#}$ and $n_1, n_2 \in \mathcal{L}$ where $\neg sm(n_1)$ and $\neg sm(n_2)$, let $\mathcal{M}(l_1, n_1)$ and $\mathcal{M}(l_2, n_2)$; then, $\mathsf{sel}^{\#}(l_1, l_2) \Leftrightarrow \mathsf{sel}(n_1, n_2)$ and $\mathsf{var}^{\#}(l_1) \Leftrightarrow \mathsf{var}(n_1)$.
- Summary nodes own their associated attachment points: *i.e.* if $\mathcal{M}(l_1, n_2)$ and $n_2 \in \mathcal{A}(u)$ then $\mathcal{M}(l_1, u)$
- Any edge pointing to a summary node from the outside must point to one of its attachment points: *i.e.* let $\mathsf{sel}^{\#}(l_1, l_2) \wedge \mathcal{M}(l_2, n_b) \wedge sm(n_b)$, then either $\mathcal{M}(l_1, n_b)$ or $\exists n_a \in \mathcal{A}(n_b) s.t. \mathcal{M}(l_2, n_a)$, and for variables, $\mathsf{var}^{\#}(l_2) \wedge \mathcal{M}(l_2, n_b) \wedge sm(n_b)$, then $\exists n_a \in \mathcal{A}(n_b) s.t. \mathcal{M}(l_2, n_a)$
- Selector edges for summary nodes not originating in an attachment point are ignored: *i.e.* if $sm(n_a) \vee sm(n_b)$ then $\mathsf{sel}(n_a, n_b) = 0 \vee n_a \in \mathcal{A}(n_b)$.
- Selector edges from an attachment point to its enclosing summary node will have value 1/2: *i.e.* if $n_a \in \mathcal{A}(n_b)$ and $\exists l_1, l_2 \in \mathcal{L}^{\#}\ s.t.\ \mathsf{sel}^{\#}(l_1, l_2) \wedge \mathcal{M}(l_1, n_a) \wedge \mathcal{M}(l_2, n_b)$, and $\neg\exists n_c\ s.t.\ \mathcal{M}(l_2, n_c) \wedge \neg sm(n_c)$, then $\mathsf{sel}(n_a, n_b) = 1/2$.

In shape analysis it is common to use 3-valued logic to represent the values of selector and variable predicates in abstract shapes. However, notice that the rules above specifically require us to ignore most selector edges involving summary nodes. The restrictions imply that the only selector edges that will potentially have value equal to 1/2 are edges from an attachment point to its corresponding summary node; that is why we have $\mathsf{next}(\mathsf{f}, \mathsf{mid}) = 1/2$ in the earlier example. As we shall see in the next section, the transition rules in the abstract semantics are defined in such a way that if the algorithm under analysis ever tries to dereference a field corresponding to one of these half edges, it will transition into an error state. This makes the analysis simpler at the expense of added imprecision, but our analysis compensates for this imprecision by relying on the unfold predicates to *materialize*[21] summary nodes.

4. ABSTRACT INTERPRETATION

Having defined the structure of the abstract domain, we now describe the transition rules used to model statements and conditionals, focusing on those aspects unique to our framework. Figures 10 and 11 show respectively the statements and conditionals considered by the synthesizer. The figures also show the formal definitions of the transition rules associated with each construct. The transition rules relate the state before the transition—the pre-state represented with non-primed predicates—with the post-state represented with primed predicates. It is assumed that the values of all other predicates not mentioned in the transition rule remain unchanged.

Statement	Transition rule
x = null	$\forall\, l \in \mathcal{L} :\ x'(l) = 0$
x = t	$\forall\, l \in \mathcal{L} :\ x'(l) = t(l)$
x = t.sel	assert $\neg\exists l_1, l_2 \in \mathcal{L} : t(l_1) \wedge sel(l_1, l_2) \wedge sm(l_2)$ $\forall\, l \in \mathcal{L} :\ x'(l) = \exists l_1\ t(l_1) \wedge sel(l_1, l)$
x.sel = null	assert $\neg\exists l_1, l_2 \in \mathcal{L} : x(l_1) \wedge sel(l_1, l_2) \wedge sm(l_2)$ $\forall\, l_1, l_2 \in \mathcal{L} :\ sel'(l_1, l_2) = \neg x(l_1) \wedge sel(l_1, l_2)$
x.sel = t	$\forall\, l_1, l_2 \in \mathcal{L} :\ sel'(l_1, l_2) = sel(l_1, l_2) \vee (x(l_1) \wedge t(l_2)))$
unfold x	unfoldPred(E, M, C) $\wedge\, x$(E) $\Longrightarrow$ $((\forall\, l_1 \xrightarrow{\text{in}} l_2 \in \mathsf{M} : \mathsf{fresh}(l_2) \wedge$ $\forall\, v \in \mathcal{V} :\ v'(l_2) = v(l_1)\ \wedge v'(l_1) = 0\ \wedge$ $\forall\, l \in \mathcal{L} :\ sel'(l, l_2) = sel(l, l_1)\ \wedge sel'(l, l_1) = 0) \wedge$ $(\forall\, l_1 \xrightarrow{\text{out}} l_2 \in \mathsf{M} : \mathsf{fresh}(l_2) \wedge$ $\forall\, l \in \mathcal{L} :\ sel'(l_2, l) = sel(l_1, l)\ \wedge sel'(l_1, l) = 0)\ \wedge$C)
fold x	foldPred(E, M, C) $\wedge\, (x$(E) $\wedge$ C) $\Longrightarrow$ $((\forall\, l_1 \xrightarrow{\text{in}} l_2 \in \mathsf{M} : \mathsf{fresh}(l_2) \wedge$ $\forall\, v \in \mathcal{V} :\ v'(l_2) = v(l_1)\ \wedge v'(l_1) = 0\ \wedge$ $\forall\, l \in \mathcal{L} :\ sel'(l, l_2) = sel(l, l_1)\ \wedge sel'(l, l_1) = 0) \wedge$ $(\forall\, l_1 \xrightarrow{\text{out}} l_2 \in \mathsf{M} : \mathsf{fresh}(l_2) \wedge$ $\forall\, l \in \mathcal{L} :\ sel'(l_2, l) = sel(l_1, l)\ \wedge sel'(l_1, l) = 0))$

Figure 10: Abstract semantics of statements manipulating pointers and pointer-valued fields

The rules follow a convention from shape analysis to assume that every statement of the form $\mathsf{exp} = \mathsf{t}$ is preceded by a statement of the form $\mathsf{exp} = \mathsf{null}$, where exp is either x or x.sel. This assumption simplifies the rules because the transition rule for $\mathsf{exp} = \mathsf{t}$ does not have to worry about destroying the value previously stored at exp [21]. The other important observation about the rules is the use of assertions for the two rules that do field dereferences. These assertions ensure that the system will transition into an error state when it tries to dereference a selector that points to a summary node, which in turn guarantees that 1/2 values corresponding to these selector predicates will not propagate through the representation.

The abstract semantics for the class of conditionals is shown in Figure 11. It can be noted that although the assignment statements in our target language of programs ignore data fields of the data-structure (.data as opposed to .sel), the conditionals can reason about the data constraints. We store data predicates using gt, gte, eq etc., which encode data constraints over the data values of locations. We do not consider conditionals involving selector dereferencing of variables, e.g. of the form x.next == null, as they can be reduced into a conditional of the form y == null where the variable y is first assigned by the statement y = x.next.

4.1 Fold/Unfold semantics

The unfold operation is described with a triple unfoldPred (E, M, C). The first argument E is called the enabling node, and it represents the summary node that is being expanded. The transition rule for the unfold x statement performs the unfold operation only if the variable x points to the enabling node E. The second argument M is the location mapping $\mathsf{M} : \mathsf{loc} \rightarrow \mathsf{loc}$, which describes how nodes before expansion relate to nodes after expansion. There are two kinds of loca-

tion mappings in M: $\overset{\text{in}}{\rightarrow}$ mappings and $\overset{\text{out}}{\rightarrow}$ mappings. An $\overset{\text{in}}{\rightarrow}$ mapping maps a location loc_1 to loc_2 such that all variables and selector edges pointing to loc_1 in the pre-state should point to loc_2 in the post-state. An $\overset{\text{out}}{\rightarrow}$ mapping maps a location loc_1 to loc_2 such that all outgoing selector edges from loc_1 in the pre-state emanate from loc_2 in the post-state.

Finally, the description of unfold also includes a set of constraints C. These constraints describe how the new nodes will be connected together, and are asserted to hold in the post-state after unfolding. The transition rule for fold x statement works similarly to the unfold rule. The difference is that the fold operations are enabled only if the constraints C are also satisfied by the state configuration in addition to the requirement of x pointing to the enabling node E. The unfold and fold predicate definitions on the summary node mid are shown in Figure 4.

Another set of fold-unfold examples for the binary search tree (bst) case studies is shown in Figure 12. The goal of bst search is to search for a value x in the tree where r represents its root. The bst search (contains) manipulation assumes that the value x always exists in the tree. The three cases of bst search (contains) unfold are: i) $x < y.val$, ii) $x = y.val$ and iii) $x > y.val$ as shown in Fig 12(a), where y denotes the root node of the subtree being unfolded. The unfold definition for the more general case of bst search is shown in Figure 12(b). The tree summary nodes labeled *stuff* are given without any unfold rules, which means they cannot be materialized, so the verifier will not be able to reason about any implementation that tries to visit them. In this way, the unfold rule is providing algorithmic insights, telling the synthesizer that a given region of the tree should not be visited or manipulated.

One important thing to note about unfold is that a given shape can be expanded in many different ways, as illustrated in Figure 4. This is expressed by having multiple unfoldPred triples with the same enabling node. As a consequence, every abstract shape in the pre-state of an unfold operation may be expanded into a *set* of abstract shapes. This expansion allows the analysis to maintain precision, but having to represent sets of abstract shapes in the abstract interpreter will pose an interesting challenge when we turn the problem into a constraint satisfaction problem.

Another very important aspect about unfold is that the presence of unfold changes the concretization relation γ between abstract and concrete shapes. In the absence of unfold, any arbitrary set of concrete nodes can be mapped to a summary node by the relation $\mathcal{M}$ described in the previous section, but unfold has the effect of placing some structural constraints on the set of nodes that can be mapped to a summary node. This is a result of the requirement that unfold correspond to skip in the concrete domain; this means that if a given abstract shape S can be transformed by unfold into any shape in the set $\{S_i\}$, then the set of concrete shapes $\gamma(S)$ should equal $\bigcup_i \gamma(S_i)$. So we can refine the earlier definition of the concretization function to say that $S^{\#} \in \gamma(S)$ if it satisfies the requirements stated before *and* if $S^{\#} \in \bigcup_i \gamma(S_i)$, where S_i are all the shapes that can be derived from S through the application of unfold. The next section elaborates on how this definition relates to the instrumentation predicates used by TVLA to describe the structure of summary nodes.

Conditional	Transition rule
x == null	$\forall\, l \in \mathcal{L} :\ \neg x(l)$
x != null	$\exists\, l \in \mathcal{L} :\ x(l)$
x == t	$\forall\, l \in \mathcal{L} :\ x(l) \iff t(l)$
x.data > t.data	$\forall\, l_1, l_2 \in \mathcal{L} : x(l_1) \wedge t(l_2) \implies \mathsf{gt}(\mathsf{data}, l_1, l_2)$
x.data ≥ t.data	$\forall\, l_1, l_2 \in \mathcal{L} : x(l_1) \wedge t(l_2) \implies \mathsf{gte}(\mathsf{data}, l_1, l_2)$
x.data == t.data	$\forall\, l_1, l_2 \in \mathcal{L} : x(l_1) \wedge t(l_2) \implies \mathsf{eq}(\mathsf{data}, l_1, l_2)$
x.data != t.data	$\forall\, l_1, l_2 \in \mathcal{L} : x(l_1) \wedge t(l_2) \implies \neg\mathsf{eq}(\mathsf{data}, l_1, l_2)$

Figure 11: Abstract semantics of the conditionals involving the pointer variables

4.2 Relationship with TVLA

In order to understand some of the more subtle aspects of our formalism, it is useful to understand how it relates to the formalism in TVLA [15]. The two most apparent differences between the two formalisms are the use of attachment points as part of summary nodes and the use of fold and unfold.

In our system, the unfold definition serves two purposes: it provides a mechanism to convey structural properties of summary nodes, and it is also used to *materialize* summary nodes, *i.e.* to produce a set of more refined shapes that together represent the same set of concrete configurations as the original configuration. In TVLA, by contrast, structural properties are described through *instrumentation predicates*. These predicates are also used for materialization, but not directly; instead, a *focus* operation first expands a summary node into a set of possible shapes, and then a *coerce* operation uses the instrumentation predicates to refine the new shapes and to remove those that do not satisfy the required structural properties.

One can understand the unfold rules in our framework as a specialized way of describing instrumentation predicates. For example, from the unfold rule for mid we can derive the following instrumentation predicate:

$$\mathsf{isMid}(f, e) = (f = e) \vee \exists f'(f.next = f' \wedge \mathsf{isMid}(f', e)) \quad (6)$$

The predicate isMid encodes that every node in mid is reachable from the front attachment point, and therefore that the sub-list between f and e is acyclic. When we say that a summary node satisfies this predicate, it means that the summary node can only represent sets of nodes where we can find two nodes f and e that satisfy the predicate. The unfold operation induces this predicate because we want the effect of unfold in the abstract domain to be equivalent to the effect of skip in the concrete domain, and this will only be true if the summary nodes satisfy this predicate. It is interesting to see, however, that the structure of the predicate is very close to the structure of the unfold rules, with the attachment points serving as convenient parameters to the predicate.

Given such an instrumentation predicate, we can map our summary nodes with attachment points to a shape in TVLA; for example, Figure 13 shows how mid would look like as a shape in TVLA. Our unfold operation is equivalent to first applying materialization to partially concretize the summary node and then coerce to remove invalid shapes obtained after materialization.

The use of unfold in place of instrumentation predicates and the use of attachment points both have a number of advantages for the purpose of our framework. The first important advantage is that it simplifies the transition rules, because it eliminates the need to track instrumentation pred-

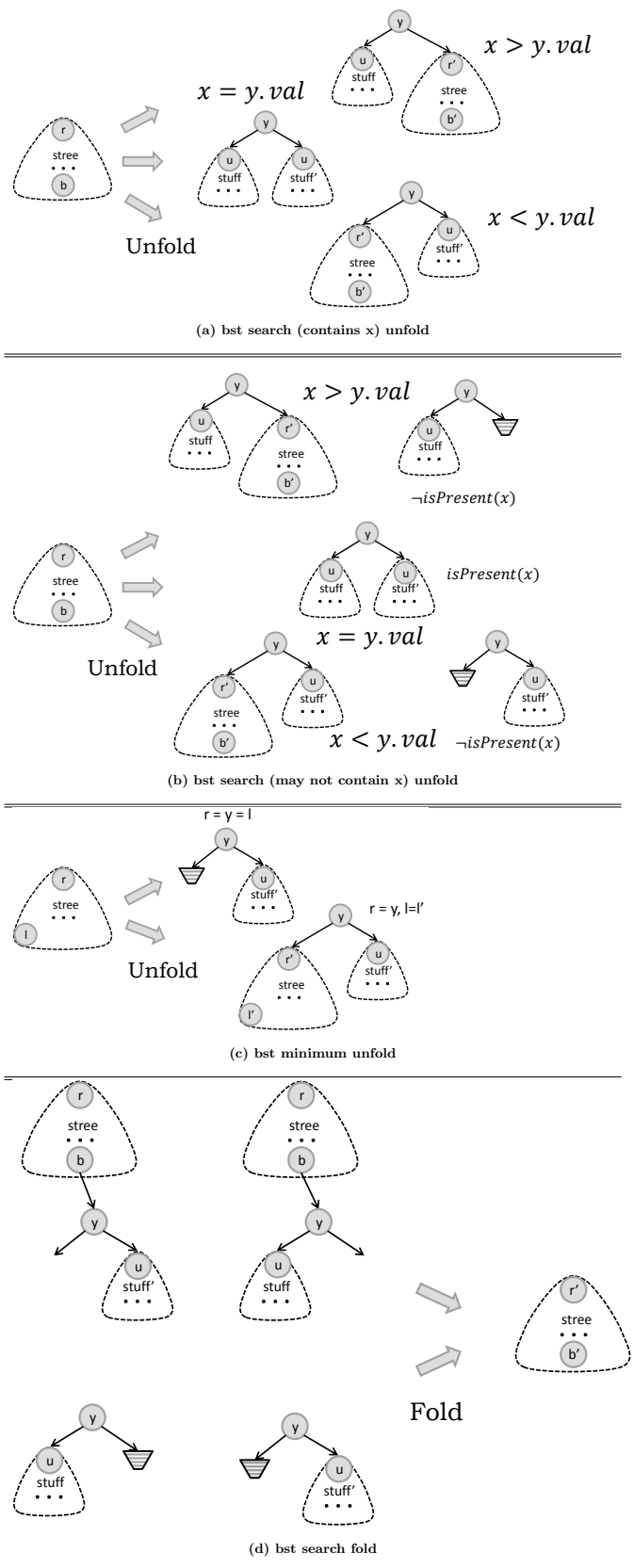


Figure 12: Unfold and fold operations for different data structure manipulations

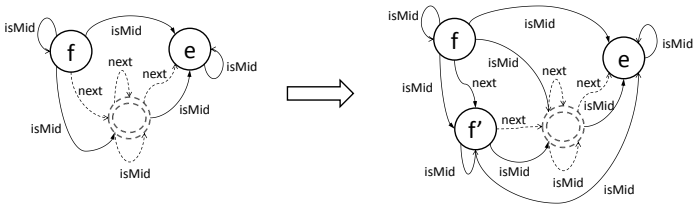


Figure 13: unfold in 3-valued shape analysis

icates. Another important benefit of using summary nodes with attachment points is that it simplifies the graphical representations, as can be readily appreciated by comparing Figure 13 with Figure 4. One clear difference between the two representations is that Figure 13 includes a number of selector edges with value 1/2 which are not present in the diagrams in Figure 4. This is partly by convention, since we omit from our representation the selector edges within summary nodes, and partly because the assumptions in Section 3 ensure that references from concrete to summary nodes always point to their attachment points.

Compared to TVLA, out formalism allows for a simpler analysis and more concise graphical representations. The downside, of course, is reduced expressive power. If one were trying to verify arbitrary programs, the shortcuts taken by our system would make the analysis impractical—it would be too easy for the user to write a program that could not be verified because it violated one of our assumptions. On the other hand, as Vechev *et al.* have pointed out [31], the combination of synthesis with abstract interpretation means that the synthesizer can work around the limitations of the abstract interpreter by producing programs that are easy to verify. In our case, the way the synthesizer works around the limitations imposed by our assumption is by using unfold statements to materialize nodes at the right time and ensure that the assumption is never violated.

5. CONSTRAINT BASED SYNTHESIS

Having defined our basic abstract interpretation framework, we can now describe how we frame the synthesis problem as a set of constraints whose solution will describe the implementation we are looking for. The starting point for this process is the control flow sketch together with the storyboard.

The control flow sketch Sk constitutes the control flow graph of the implementation; unlike a CFG, however, the vertices are not just basic blocks because they can contain conditional assignments. More formally, we represent the control flow sketch Sk as a directed graph $Sk = G(V, E)$ where V represents a set of blocks of code which can either be a sequence of conditional assignment statements or a conditional (for loop exit conditions); E in turn represents potential transitions between these blocks of code. By convention, we say that vertex v_0 is the entry point of the graph node and v_N is the exit point; neither the entry nor the exit blocks contain any code. As for scenarios, each of them is represented as a pair of input and output states $S_i = (In_i, Out_i)$.

We use standard techniques to encode each block of unknown statements as a parameterized function $F_i(in, c_i)$, where the parameter c_i selects which block of code out of

the set of possible blocks of code F_i will represent. The inputs and outputs of F_i are elements of the abstract domain, which happen to be sets of shapes. Now, the set of possible sequences of conditional assignments is infinite, but bounding the maximum length of the statement sequence makes the set finite as there are a finite number of assignment statements and conditionals of the form shown in Figure 10 and Figure 11 respectively.

The goal of the synthesis process is to find values of c_i such that for each scenario S_k, the least fixed point solution to the following equation satisfies $t_N = Out_k$:

$$t_0 = In_k \quad \wedge \quad \forall v_i \in (V \setminus v_0) \;\; t_i = F_i(\bigcup_{j \in pred(v_i)} t_j, c_i) \quad (7)$$

The function $pred(v_i)$ in the equation above indicates the set of predecessors of node v_i. The equation above is fairly simple, but two challenges prevent us from solving it directly with an SMT solver. First, the equation above requires us to find not just any solution, but the least fixed point solution. Additionally, the t_i in the equation above are elements in the abstract domain, which is composed of sets of shapes. Such sets can get quite big given the nature of our domain, so representing them naïvely in an SMT solver is infeasible. In the rest of this section, we describe how our framework addresses both of these problems.

5.1 Computing Least Fixed Points

In order to find a least fixed point solution to Equation (7), we start from the assumption that an iterative method can reach a least fixed point after visiting each vertex in Sk at most K times. Now, let P^K be the set of all paths in Sk that visit each vertex at most K times.For each path $\mathsf{p}_i \in \mathsf{P}^K$, we can define a path transformer $\mathsf{p}_i(in, \vec{\mathsf{C}})$ which is just the composition of all the transfer functions $F_t(in, c_t)$ of all the vertices v_t in the path, where $\vec{\mathsf{C}} = [c_0, \ldots, c_N]$. Then, the least fixed point solution to the value of t_N in Equation (7) will be given by

$$\bigcup_{\mathsf{p}_i \in \mathsf{P}^K} \mathsf{p}_i(In_k, \vec{\mathsf{C}})$$

The equivalence follows from the distributivity of F (*i.e.* the fact that $F(a \cup b) = F(a) \cup F(b)$). Given a solution to the equation above, it is easy to check the assumption of K convergence by simply checking the solution against Equation 5.

5.2 Dealing with sets of abstract shapes

As we said before, in order to feed the constraints into a solver, we would like to avoid having to reason about sets of abstract shapes. Our strategy will be as follows. First, from a transfer function F, we can define a function F^j that returns a singleton set containing the j^{th} element of the set returned by F, or an empty set if there is no j^{th} element. Thus, $F(a) = \cup_j F^j(a)$, where each F^j produces either singleton or empty sets.

The strategy even works when composing functions thanks to the distributivity of F. Because of this property, if we have a function $F(a)$ and a function $T(a)$, then the composition $F(T(a))$ can be computed as $\cup_{i,j} F^j(T^i(a))$.

In the case of our transfer functions $F_i(in, c)$, it is relatively easy to derive the functions $F_i^j(in, c)$. For example, one of the statements that can produce multiple shapes from a single one is unfold, so if we want a function to return only the j^{th} shape produced by unfold, we only use the j^{th} unfoldPred instead of using all of them. Composing the transfer functions for each block into path expressions, we get a path expression $\mathsf{p}_i^{\vec{j}}(In_k, \vec{\mathsf{C}})$, where instead of composing functions $F_i(in, c_i)$ in the path, we compose functions $F_i^{j_i}(in, c_i)$. With this transformation, the constraint we need to solve becomes:

$$\exists \vec{\mathsf{C}} \;\; (Out_k = \bigcup_{\vec{j}} \bigcup_{p_i \in \mathsf{P}} \mathsf{p}_i^{\vec{j}}(In_k, \vec{\mathsf{C}}))$$

The set unions in the equation above can be turned into universal quantifiers to produce the following equation:

$$\exists \vec{\mathsf{C}} \;\; (\forall_{\vec{j}} \forall_{p_i \in \mathsf{P}} \; \mathsf{p}_i^{\vec{j}}(In_k, \vec{\mathsf{C}}) \in Out_k \wedge \exists_{\vec{j}} \exists_{p_i \in \mathsf{P}} \; \mathsf{p}_i^{\vec{j}}(In_k, \vec{\mathsf{C}}) = Out_k)$$

The universally quantified part of the equation forces the union of the path transformers to be a subset of Out_k, while the existentially quantified part of the constraint ensures that the singleton Out_k is a subset of the union of the path transformers. The equation above no longer has to reason about sets with more than one element, but in exchange for that, it has to cope with $\exists\forall$ quantifier alternation. However, the SKETCH system is very effective in dealing with such doubly quantified formulas, so our system actually translates the above equation into a sketch and uses the SKETCH solver to find a solution to all the unknowns.

5.3 Termination

Equation 5 only ensures partial correctness, so there is no termination guarantee for the synthesized implementation. However, we have found that adding a few additional constraints was enough to guarantee terminating solutions for all the examples we examined. The additional constraint was to require that for every state reachable inside any loop in the program it is possible to satisfy the loop exit condition in an additional K loop unrollings. If the unfold and fold predicates satisfy well-formedness constraints [19] and with the restriction of using only one unfold and fold operation per loop, the framework can guarantee termination of the synthesized implementation using a reasoning similar to [4]. This restriction works for data-structure manipulations that perform a single pass over the data-structure.

6. EXPERIMENTS

In our experiments with the Storyboard framework, the key questions we explored were: i) how does it scale for synthesizing reasonably complex data-structure manipulations, ii) how much additional information is required to be provided in the storyboards other than just the input-output scenarios, iii) how much having abstraction in the scenarios help and iv) can we use it to synthesize user-defined data-structure manipulations.

Table 1 presents the experimental results of the case studies that we performed with the framework. The experiments were run on an Intel Core-i7 1.87GHz CPU with 4GB of RAM. The first column in the table shows the name of the manipulation where ll refers to singly linked-list, dll refers

to doubly linked-list and bst refers to binary search tree. The table presents the details about number of scenarios used in storyboard, the total time it took to synthesize the implementation, the number of clauses in the SAT translation of the constraints and the memory used. Even though this is not the most efficient encoding of the constraints, we were able to synthesize all the manipulations in less than 6 minutes using less than 1 GB of memory.

For reducing the search space in our experiments, we had to restrict the usage of unfold and fold statements to at most once at the beginning and end locations inside the loop respectively; which works well for single-pass algorithms. The case studies with the Interm column marked yes in Table 1 required some additional intermediate state configuration, *e.g.* in the storyboard for linked list insertion, we also had to provide intermediate state configuration after the loop body in the skeleton for helping the synthesizer to converge faster. These intermediate configurations present a natural interface for providing hints about the manipulation. Some case studies also required composition of storyboards (marked with a *), *e.g.* the bst-deletion storyboard required composition of bst-search and bst-find-min storyboards.

In some cases like bst-deletion, we found that the abstract input-output specification was too weak and allowed many undesired solutions; but it was easily fixed by providing a couple of concrete input-output bst instances. We also performed an experiment where we only provided concrete examples for these manipulations, the synthesizer either generated an undesired solution or got timed out and never converged for most of these case studies. This experiment shows the ability of abstract input-output examples to prune a big search space of undesired programs.

We have used our framework to synthesize manipulations for a complicated real-world AIG data structure. AIG is a DAG that encodes the structural implementation of the logical functionality of a circuit [18] using two-input AND gates and inverters. Each internal node of AIG represents an *and* gate and has two parents corresponding to the two inputs of the gate. The child list information for each node is overlayed inside the node itself by keeping a pointer to the first child and pointers to the sibling nodes. Even though our framework currently can not synthesize arbitrary graph manipulations, we exploit the listness property of the child lists of AIG nodes for synthesizing its manipulations.

Even for complicated data-structures like red-black trees, where it is difficult to draw a simple storyboard expressing the complex invariants about the data-structure, we found the storyboard framework helpful for synthesizing fragments of low-level code of different cases individually and then manually composing the synthesized code to obtain a complete implementation. Figure 14 shows the storyboard for red-black tree `fixInvariant` method (part of the `insertion` procedure) that we obtained from an online lecture note [1]. We used the framework to synthesize low-level code for the four cases, which were then easily composed manually inside the complete algorithm. Our tool and more details about the case studies can be found at the storyboard website [2].

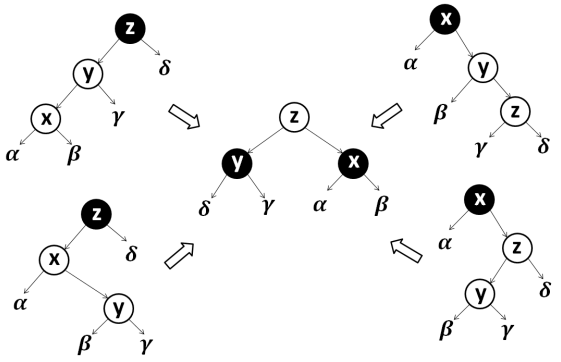


Figure 14: Red-black tree `fixInvariant` storyboard

Manipulation	#Scens	Time	#Clauses	Memory	Loops	Interm
ll-insertion	4	2m9s	1.99M	0.75GB	1	Yes
ll-deletion	4	1m48s	1.88M	0.54GB	1	Yes
ll-reversal	4	1m49s	1.3M	0.35GB	1	No
ll-find-last	4	0m56s	1.02M	0.29GB	1	No
ll-swap-first-last	4	4m18s	1.08M	0.31GB	1	Yes
dll-traversal	4	1m58s	1.72M	0.88GB	1	No
dll-reversal	4	3m47s	2.04M	0.49GB	1	No
bst-search(contains)	1	1m02s	0.62M	0.37GB	1	No
bst-search	1	6m07s	0.77M	0.45GB	1	No
bst-find-min	1	0m58s	0.63M	0.18GB	1	No
bst-find-max	1	0m23s	0.57M	0.16GB	1	No
bst-left-rotate	3	3m18s	1.41M	0.50GB	0	No
bst-right-rotate	3	3m15s	1.47M	0.43GB	0	No
bst-insertion*	3	1m52s	1.04M	0.46GB	1	Yes
bst-deletion*	6	3m13s	0.63M	0.62GB	2	Yes
aig-insertion*	4	1m04s	0.17M	0.31GB	1	Yes

Table 1: Experimental results for case studies

7. RELATED WORK

Software synthesis has been an active research area at least since the early 80s when Waldinger and Manna [17, 16] did seminal work on deductive synthesis. A more algorithmic approach to synthesis was pioneered by Pnueli and Rosner in the context of finite state controllers [20]. More recently, some of the ideas from the field of controller synthesis have been applied to software, for example, to synthesize program repairs [14]. For a recent survey, see [8].

The idea of using abstract interpretation for synthesis was recently introduced by Vechev, Yahav and Yorsh [31], as a follow up to earlier work on synthesis of concurrent data-structures [30]. Their system is designed to synthesize efficient synchronization for concurrent programs, and is very different from ours, both in its scope and in the algorithms it uses. Unlike their system, our synthesizer is based on a more general constraint-based approach that allows us to handle extremely large search spaces with no apparent structure.

The idea of using a constraint-based approach for abstract interpretation was previously introduced by Gulwani *et al.* [10]. Recently, their group has used similar techniques to synthesize invariants [11] and even complete programs [29]. Some important distinctions between their work and ours are the use of storyboards to capture insights, as well as our path-based representation of the constraints to support a very large and complex abstract domain.

The idea of using a sketch to define the structure of the implementation was adapted from the original work on sketch based synthesis [25]. The idea was originally applied to the domain of bit-stream manipulations [27], such as ciphers and error correction codes, and has been applied more recently to scientific programs [25] and concurrent data-structures [26]. Although SKETCH can synthesize some of the data-structure manipulations, it requires the programmer to provide detailed sketches and only provides bounded guarantees for the

synthesized implementation. Additionally, writing specifications for data-structure manipulations tend to be harder, because they have to be written as tricky test harnesses.

Recent work in data representation synthesis [12] automatically synthesizes efficient data-structure representations for a given set of data usage patterns. The representations are built from a library of data-structure building-blocks and support only a fixed set of common interface methods. Our framework supports the implementation of more general data-structure manipulations, such as our list reverse example. The price of the generality is a more involved interaction model compared with the push-button interface provided by that system.

PINS [28] introduced the idea of focusing on individual paths when generating constraints. Their approach does not build upon abstract interpretation as ours and our framework lets the synthesizer select interesting paths automatically using the CEGIS algorithm unlike a heuristic technique used by PINS. Gulwani *et al.* [9, 13] have proposed component-based synthesis techniques for synthesizing tricky (but loop-free) code snippets from a given multi-set of components. These techniques are not applicable in our setting as we deal with loopy programs.

8. ACKNOWLEDGMENT

We would like to thank Rastislav Bodik, Zhilei Xu and the anonymous reviewers for their valuable comments and suggestions. This research was supported by the National Science Foundation grant CCF-1049406 and by MIT's Computer Science and Artificial Intelligence Lab (CSAIL).

9. REFERENCES

[1] http://cs.wellesley.edu/~cs231/fall01/red-black.pdf.

[2] http://people.csail.mit.edu/rishabh/storyboard/.

[3] Robert K. Brayton and Alan Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In *CAV*, pages 24–40, 2010.

[4] James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *POPL*, pages 101–112, 2008.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[6] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[7] Pascal Fradet and Daniel Le Metayer. Shape types. In *POPL*, pages 27–39. ACM Press, 1997.

[8] Sumit Gulwani. Dimensions in program synthesis. In *PPDP*, pages 13–24, 2010.

[9] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.

[10] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292, 2008.

[11] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI*, 2009.

[12] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data representation synthesis. In *PLDI*, pages 38–49. ACM, 2011.

[13] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224, 2010.

[14] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *CAV*, pages 226–238, 2005.

[15] Tal Lev-Ami and Shmuel Sagiv. Tvla: A system for implementing static analyses. In *SAS*, 2000.

[16] Z. Manna and R. Waldinger. Synthesis: Dreams => programs. *IEEE Transactions on Software Engineering*, 5(4):294–328, 1979.

[17] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

[18] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, EECS, UC Berkeley, 2005.

[19] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei ngan Chin. Automated verification of shape and size properties via separation logic. In *In VMCAI*. Springer, 2007.

[20] Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. In *ICALP*, pages 652–671, London, UK, 1989. Springer-Verlag.

[21] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, pages 105–118. ACM, 1999.

[22] Horst Samulowitz and Fahiem Bacchus. Binary clause reasoning in qbf. In *SAT*, pages 353–367, 2006.

[23] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *S. Muchnick, N. Jones (Eds.), Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

[24] Armando Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS, UC Berkeley, 2008.

[25] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.

[26] Armando Solar-Lezama, Chris Jones, Gilad Arnold, and Rastislav Bodík. Sketching concurrent datastructures. In *PLDI*, 2008.

[27] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodik, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[28] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. Path-based inductive synthesis for program inversion. In *PLDI*, pages 492–503. ACM, 2011.

[29] Saurabh Srivastava, Sumit Gulwani, and Jeffrey Foster. From program verification to program synthesis. *POPL*, 2010.

[30] Martin Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. *SIGPLAN Not.*, 43(6):125–135, 2008.

[31] Martin Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, New York, NY, USA, 2010. ACM.