Pointer Program Synthesis as Non-deterministic Planning

Xu Lu and Bin Yu(✉)

ICTT and ISN Lab, Xidian University, Xi'an 710071, People's Republic of China
byu@xidian.edu.cn

Abstract. Program synthesis is the task of automatically constructing programs that satisfy a given high-level formal specification (constraints). In this paper, we concentrate on the synthesis problem of a special category of program, named pointer program that manipulate heaps. Separation logic has been applied successfully in modular reasoning of pointer programs. There are many studies on formal analysis of pointer programs using a form of symbolic execution based on a decidable proof theory of separation logic. Automatic specification checking can be done efficiently by means of symbolic execution. With this basis, we present a novel approach to simulate the symbolic execution process for the sake of synthesizing pointer programs. Concretely, symbolic execution rules are compiled into a non-deterministic planning problem which can be directly solved by existing planners. The reason of using non-deterministic planning is that it enables to generate strong cyclic plans where loop and branch connections (similar to basic program constructs) may appear. We show the preliminary experimental results on synthesizing several programs that work with linked lists.

Keywords: Program synthesis · Non-deterministic planning · Separation logic · Symbolic execution

1 Introduction

Automatic synthesis of program has long been considered as one of the most central problems in computer science. It is the task of automatically finding programs from the underlying language that satisfy user intent expressed in some form of (formal) constraints [15]. Usually, we need to perform certain kind of search over the state space of all potential programs in order to generate one that meets the constraints.

Fruitful studies have achieved a lot of progress for program synthesis in many communities. Beginning in 1957, Alonzo Church defines the problem to synthesize a circuit from mathematical requirements. Reactive synthesis is a special

This research is supported by the National Natural Science Foundation of China under Grant 61806158, China Postdoctoral Science Foundation under Grant 2019T120881 and Grant 2018M643585.

J. Xue et al. (Eds.): SOFL+MSVL 2020, LNCS 12723, pp. 126–141, 2021.
https://doi.org/10.1007/978-3-030-77474-5_9

case of program synthesis that aims to produce a controller that reacts to environment's inputs satisfying a given temporal logic specification [5]. An international competition called the Reactive Synthesis Competition is held annually since 2014.[1] Camacho et al. establish the correspondence of planning problems with temporally extended goals to reactive synthesis problems [8]. Building on this correspondence, synthesis can be realized more efficiently via planning. A pattern-based code synthesis approach is presented to assemble an application from existing components [12]. The code patterns are expressed by planning domain models. Recently, the application of AI techniques especially deep learning methods in program synthesis becomes an active research topic. DeepCoder, developed by Microsoft, is to train a neural network to predict properties of program that generated the outputs from the inputs [1]. Empirically, DeepCoder is able to help generate small programs only containing several lines. Gu et al. propose a deep learning based approach to generate API usage sequences for a given natural language query [14]. The work in [2] transforms a graphical user interface screenshot created by a designer into computer code by deep learning methods. Various codes for three different platforms can be generated with the accuracy over 77%. In addition, other techniques from different perspectives such as inductive programming [16] and genetic programming [20] are also applied in program synthesis. However, synthesizing a program is still a challenging problem due to the large search space.

AI planning, or planning for short, has been successfully applied in many fields. Planning is the problem of finding a sequence of actions that leads from an initial state to a goal state [13]. Classical planning is the problem such that each action has deterministic outcome. If the outcomes of some actions are uncertain, the problem is referred to as a non-deterministic planning problem. The non-deterministic actions give rise to the exponential growth in the search space and hence make the problem more difficult even in the simplest situation where the states of the world are full observable. A plan to a non-deterministic planning problem may have loops or branches which are similar to basic programming language constructs. Inspired by this, we obtain the idea to bridge the gap between program synthesis and non-deterministic planning.

Programs that manipulate heaps are called pointer programs. Pointer operations allow dynamic heap allocation and deallocation, pointer reference and dereference etc. These characteristics make pointer programs more error prone. Separation logic is an extension of Hoare logic addressing the task of reasoning about pointer programs [19]. Its key power lies in the separating conjunction $\Sigma_1 * \Sigma_2$, which asserts that Σ_1 and Σ_2 hold for separate portions of heaps, leading the reasoning in a modular way. Starting from the pioneer work [4], which presents a symbolic execution of a fragment of separation logic formulas called symbolic heaps, many researchers exploit symbolic execution techniques to build formal proofs of pointer programs [7,10,17]. The most famous tool, Infer [6], is a static analyzer developed at Facebook rooting on symbolic execution.

[1] http://www.syntcomp.org/.

This paper focusses on the pointer program synthesis. We propose a compilation based approach to simulate the symbolic execution process of pointer programs by non-deterministic planning. The compilation result is specified in the Planning Domain Definition Language (PDDL) [11] that is a standard input to the state-of-the-art planners. The major contribution of our work is the encoding approach from symbolic execution rules to non-deterministic planning models. To the best of our knowledge, it is unique in using non-deterministic planners as program synthesizers.

The rest of the paper is organized as follows. Section 2 and Sect. 3 review the notations of non-deterministic planning and symbolic heaps; Sect. 4 shows symbolic execution theory of pointer programs with symbolic heaps; Sect. 5 describes how to compile symbolic execution rules into non-deterministic planning problems; Sect. 6 gives the experimental results; the last section concludes our work.

2 FOND Planning

We assume environments are fully observable. Following [13], a *Fully Observable Non-Deterministic* (FOND) planning problem $\mathcal{P}$ is a tuple $(\mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A})$, where $\mathcal{F}$ is a set of *fluents*, $\mathcal{I} \subseteq F$ characterizes what holds initially, $\mathcal{G} \subseteq \mathcal{F}$ characterizes the goal, and $\mathcal{A}$ is the set of actions. The set of literals of $\mathcal{F}$ is $Lits(\mathcal{F}) = \mathcal{F} \cup \{\neg f \mid f \in \mathcal{F}\}$. Each action $a \in \mathcal{A}$ is associated with a pair $(pre(a), \mathit{eff}(a))$, where $pre(a) \subseteq Lits(\mathcal{F})$ is the precondition and $\mathit{eff}(a)$ is a set of outcomes of a. An outcome $o \in \mathit{eff}(a)$ is a set of conditional effects (with, possibly, an empty condition), each of the form $C \rhd l$, where $C \subseteq Lits(\mathcal{F})$ and $l \in Lits(\mathcal{F})$. Briefly speaking, $C \rhd l$ expresses the meaning that after applying a in the current state, l becomes true in the next state if current state satisfies C. A planning state s is a subset of $\mathcal{F}$ that are true.[2] Given $s \subseteq \mathcal{F}$ and $f \in \mathcal{F}$, we say that s satisfies f, denoted $s \models f$ iff $f \in s$. In addition, $s \models \neg f$ iff $f \notin s$, and $s \models L$ for a set of literals L, if $s \models l$ for every $l \in L$. An action a is *applicable* in state s if $s \models pre(a)$. We say s' is a result of applying a in s iff for one outcome o in $\mathit{eff}(a)$, $s' = s \backslash \{f \mid (C \rhd \neg f) \in o, s \models C\} \cup \{f \mid (C \rhd f) \in o, s \models C\}$.

Solutions to a FOND planning problem are referred to as *policies*. A policy p is a partial mapping from states to actions. We say a is applicable in s if $p(s) = a$. An *execution* σ of a policy p in state s is a finite sequence $\langle (s_0, a_0), \ldots, (s_{n-1}, a_{n-1}), s_n \rangle$ or an infinite sequence $\langle (s_0, a_0), (s_1, a_1), \ldots \rangle$, where $s_0 = s$, and all of its state-action-state substrings s, a, s' satisfy $p(s) = a$ and s' is a result of applying a in s. Finite executions ending in a state s if $p(s)$ is undefined. A state trace π can be *yielded* from an execution σ by removing all the action symbols from σ.

[2] Fluents in $\mathcal{F} \backslash \mathcal{I}$ are implicitly assumed to be false according to the closed world assumption.

An infinite execution σ is *fair* iff whenever s, a occurs infinitely often within σ, then for every s' that is a result of applying a in s, s, a, s' occurs infinitely often. A solution to $\mathcal{P}$ is *strong cyclic* iff each of its executions in $\mathcal{I}$ is either finite and ends in a state that satisfies $\mathcal{G}$ or is infinite and unfair [9]. Intuitively speaking, the execution fairness of a strong cyclic solution guarantees that a goal state can eventually be reached from every reachable state with no effect that is always ignored. There are also *strong* solutions and *weak* solutions to a FOND planning problem, but we will not need those definitions in this paper.

3 Symbolic Heaps

We assume a set of programs variables Var (ranged over by $x, y, \ldots$), and a set of primed variables Var' (ranged over by $x', y', \ldots$). All variables are restricted as pointer type. The primed variables can only be used within logical formulas. The concrete heap models contain a set of locations Loc and a special notation nil which indicates a null pointer value. Let $Val = Loc \cup \{nil\}$. We then define stack $\mathcal{S}$ and heap $\mathcal{H}$ as:

$$\mathcal{S} : (Var \cup Var') \rightarrow Val \qquad \mathcal{H} : Loc \rightharpoonup Val$$

A heap maps a location to a location or nil representing a heap cell. The syntax of symbolic heap is defined below [4] which is a strict subset of separation logic [19].

$e ::= x \mid x' \mid nil$	expression
$\Pi ::= e_1 = e_2 \mid e_1 \neq e_2 \mid true \mid \Pi_1 \wedge \Pi_2$	pure formula
$\Sigma ::= emp \mid e_1 \mapsto e_2 \mid ls(e_1, e_2) \mid true \mid \Sigma_1 * \Sigma_2$	spatial formula
$P ::= \Pi \wr \Sigma \mid \exists x' : P$	symbolic heap

Note that the assertions are restricted without negations and universal quantifiers. A symbolic heap $\Pi \wr \Sigma$ can be divided into pure part Π (heap independent) and spatial part Σ (heap dependent), where Π is essentially an $\wedge$-separated sequence of pure formulas, and Σ a $*$-separated sequence of spatial formulas. The pure part is straightforward to understand, and the spatial part characterize spatial features of heaps. $e_1 \mapsto e_2$ is read as e_1 points-to e_2. It can hold only in a singleton heap, where e_1 is the only active cell holding the value e_2. $ls(e_1, e_2)$ denotes a linked list segment with head pointer e_1 and e_2 holding in the tail cell. A complete linked list is one that satisfies $ls(e, nil)$.

The semantics of symbolic heaps is given by a relation $\mathcal{S}, \mathcal{H} \models_{\text{SH}} P$. $\mathcal{H} = \mathcal{H}_1 \bullet \mathcal{H}_2$ indicates that the domains of $\mathcal{H}_1$ and $\mathcal{H}_2$ are disjoint, and $\mathcal{H}$ is their union.

$[\![x]\!]s \stackrel{\text{def}}{=} s(x) \qquad [\![x']\!]s \stackrel{\text{def}}{=} s(x') \qquad [\![nil]\!]s \stackrel{\text{def}}{=} nil$

$\mathcal{S}, \mathcal{H} \models_{\text{SH}} e_1 = e_2 \quad \text{iff} \quad [\![e_1]\!]s = [\![e_2]\!]s.$

$\mathcal{S}, \mathcal{H} \models_{\text{SH}} e_1 \neq e_2 \quad \text{iff} \quad [\![e_1]\!]s \neq [\![e_2]\!]s.$

$\mathcal{S}, \mathcal{H} \models_{\text{SH}} true \quad \text{iff} \quad \text{always.}$

$\mathcal{S}, \mathcal{H} \models_{\text{SH}} \Pi_1 \wedge \Pi_2 \quad \text{iff} \quad \mathcal{S}, \mathcal{H} \models_{\text{SH}} \Pi_1 \text{ and } \mathcal{S}, \mathcal{H} \models_{\text{SH}} \Pi_2.$

$\mathcal{S}, \mathcal{H} \models_{\text{SH}} emp \quad \text{iff} \quad \mathcal{H} = \emptyset.$

$\mathcal{S}, \mathcal{H} \models_{\text{SH}} e_1 \mapsto e_2 \quad \text{iff} \quad \mathcal{H} = [([\![e_1]\!]s, [\![e_2]\!]s)].$

$\mathcal{S}, \mathcal{H} \models_{\text{SH}} ls(e_1, e_2) \quad \text{iff} \quad$ there is a nonempty acyclic path from $[\![e_1]\!]s$ to $[\![e_2]\!]s$ in $\mathcal{H}$ and this path contains all heaps cells in $\mathcal{H}$.

$\mathcal{S}, \mathcal{H} \models_{\text{SH}} \Sigma_1 * \Sigma_2 \quad \text{iff} \quad \exists \mathcal{H}_1, \mathcal{H}_2 : \mathcal{H} = \mathcal{H}_1 \bullet \mathcal{H}_2 \text{ and } \mathcal{S}, \mathcal{H}_1 \models_{\text{SH}} \Sigma_1 \text{ and } \mathcal{S}, \mathcal{H}_2 \models_{\text{SH}} \Sigma_2.$

$\mathcal{S}, \mathcal{H} \models_{\text{SH}} \Pi \wr \Sigma \quad \text{iff} \quad \mathcal{S}, \mathcal{H} \models_{\text{SH}} \Pi \text{ and } \mathcal{S}, \mathcal{H} \models_{\text{SH}} \Sigma.$

$\mathcal{S}, \mathcal{H} \models_{\text{SH}} \exists x' : P \quad \text{iff} \quad \exists v \in Val : \mathcal{S}, \mathcal{H} \models_{\text{SH}} P(v/x').$

For simplicity, symbolic heap we define only allows to reason about linked lists. The field can be regarded as the next pointer. For other linked shapes such as binary trees, we can extend the points-to assertion as $e_1 \mapsto e_2, e_3$. The semantics of list segment is given informally, saying that it holds of given heap containing at least one heap cell. Therefore it is equivalent to the least predicate satisfying:

$$ls(e_1, e_2) \Leftrightarrow e_1 \mapsto e_2 \vee \exists x' : e_1 \neq e_2 \wedge e_1 \mapsto x' * ls(x', e_2)$$

4 Symbolic Execution

In this section we give symbolic execution rules for a pointer programming language. The grammar of commands is given by:

$b ::= e_1 = e_2 \mid e_1 \neq e_2$ Boolean terms

$c ::= x := e \mid x := [e] \mid [e] := e' \mid new(x) \mid dispose(e)$ Primitive commands

$\mathcal{C} ::= c \mid \mathcal{C}_1; \mathcal{C}_2 \mid while\ (b)\ do\ \{\mathcal{C}\} \mid if\ (b)\ then\ \{\mathcal{C}_1\}\ else\ \{\mathcal{C}_2\}$ Commands

The heap dereferencing operator $[\cdot]$ is similar to symbolic heaps, that refers to the "next" field. $x := e$ is the assignment, $x := [e]$ and $[e] := e'$ are called lookup and mutation respectively, $new(x)$ and $dispose(e)$ are heap allocation and deallocation commands.

Shown in Table 1, the symbolic execution semantics $P, \mathcal{C} \implies P'$ takes a symbolic heap P and a primitive command $\mathcal{C}$ as input, and transforms it into a new symbolic heap P' as an output. The primed variables x', y' are fresh primed variables in these rules.

Table 1. Symbolic execution rules

$\Pi \wr \Sigma$	$new(x)$	$\implies$	$\exists x', y' : (\Pi \wr \Sigma)(x'/x) * x \mapsto y'$
$\Pi \wr \Sigma * e_1 \mapsto e_2$	$dispose(e_1)$	$\implies$	$\Pi \wr \Sigma$
$\Pi \wr \Sigma$	$x := e$	$\implies$	$\exists x' : x = e(x'/x) \wedge (\Pi \wr \Sigma)(x'/x)$
$\Pi \wr \Sigma * e_1 \mapsto e_2$	$[e_1] := e_3$	$\implies$	$\Pi \wr \Sigma * e_1 \mapsto e_3$
$\Pi \wr \Sigma * e_1 \mapsto e_2$	$x := [e_1]$	$\implies$	$\exists x' : x = e_2(x'/x) \wedge (\Pi \wr \Sigma * e_1 \mapsto e_2)(x'/x)$

We use notation $A(e)$ for primitive commands that access heap cell e:

$$A(e) ::= [e] := e' \mid x := [e] \mid dispose(e)$$

When executing $A(e)$, we expect its precondition to be in a particular form $\Pi \wr \Sigma * e \mapsto e'$. That is, the value holds in e should be explicitly exposed in order to fire the rule. Therefore, we have to equivalently rearrange the precondition whenever current symbolic heap do not match the rule.

Rearrangement rules are listed below. The Switch rule simply makes use of equalities to recognize that a dereferencing step is possible. The other two rules correspond to unrolling a list segment. To do so, we need to unroll the list to be a single heap cell (Unroll List2) or more cells (Unroll List1).

Rearrangement Rules

$$\text{Switch} \quad \frac{\Pi_1 \wr \Sigma_1 * e_1 \mapsto e_3, A(e_1) \implies \Pi_2 \wr \Sigma_2}{\Pi_1 \wr \Sigma_1 * e_2 \mapsto e_3, A(e_1) \implies \Pi_2 \wr \Sigma_2} \quad \Pi_1 \vdash e_1 = e_2$$

$$\text{Unroll List1} \quad \frac{\exists x' : e_1 \neq e_2 \wedge \Pi_1 \wr \Sigma_1 * e_1 \mapsto x' * ls(x', e_2), A(e_1) \implies \Pi_2 \wr \Sigma_2}{\Pi_1 \wr \Sigma_1 * ls(e_1, e_2), A(e_1) \implies \Pi_2 \wr \Sigma_2}$$

$$\text{Unroll List2} \quad \frac{\Pi_1 \wedge e_1 \mapsto e_2 \wr \Sigma_1, A(e_1) \implies \Pi_2 \wr \Sigma_2}{\Pi_1 \wr \Sigma_1 * ls(e_1, e_2), A(e_1) \implies \Pi_2 \wr \Sigma_2}$$

Generally, the number of symbolic heaps is infinite since primed variables can be introduced during symbolic execution. For example, in a loop that includes allocation (e.g., *while* (*true*) *do* $\{\ldots; new(x); \ldots\}$). An arbitrary length of symbolic heap can be generated, i.e., $x \mapsto x' * x' \mapsto x'' \cdots$. In order to achieve fixed-point convergence, abstraction rules $\Pi_1 \wr \Sigma_1 \rightsquigarrow \Pi_2 \wr \Sigma_2$ are introduced.

The main effort of abstraction rules is to reduce primed variables. The abstraction rules are reported below. On one hand, we can remove primed variables from the pure parts of formulas (Abs1). On the other hand, we can gobble up primed variables by merging lists, swallowing single cells into lists, and abstracting two cells by a list (Abs2 and Abs3). We use the notation $H(e_1, e_2)$ to stand for a formula in either of the form $e_1 \mapsto e_2$ or $ls(e_1, e_2)$.

Abstraction Rules

Abs1 $e = x' \wedge \Pi \wr \Sigma \rightsquigarrow (\Pi \wr \Sigma)(e/x')$ or $x' = e \wedge \Pi \wr \Sigma \rightsquigarrow (\Pi \wr \Sigma)(e/x')$

$$\text{Abs2}\ \frac{\Pi \vdash e_2 = nil \qquad x' \text{ not in } \{\Pi, \Sigma, e_1, e_2\}}{\Pi \wr \Sigma * H_1(e_1, x') * H_2(x', e_2) \rightsquigarrow \Pi \wr \Sigma * H(e_1, nil)}$$

$$\text{Abs3}\ \frac{\Pi \vdash e_2 = e_3 \qquad x' \text{ not in } \{\Pi, \Sigma, e_1, e_2, e_3, e_4\}}{\Pi \wr \Sigma * H_1(e_1, x') * H_2(x', e_2) * H_3(e_3, e_4) \rightsquigarrow \Pi \wr \Sigma * ls(e_1, e_2) * H_3(e_3, e_4)}$$

The $*$-conjunct $H_3(e_3, e_4)$ cannot be left out by considerations of soundness as Berdine and Calcagno pointed out [3,4]. If we want to abstract $H_1(-,-)$ and $H_2(-,-)$ into one, the end of the second should not point back into the first.

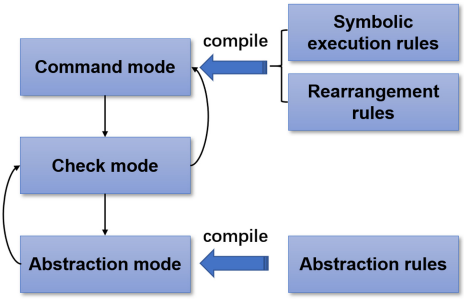

Fig. 1. The encoding approach

5 Compiling Symbolic Execution into FOND Planning

In this section we compile a pointer program synthesis problem into a FOND planning problem. The former is formalized as follows.

Definition 1 (Pointer Program Synthesis). *Given symbolic heaps P_{in} and P_{out} as input and output respectively, the task of pointer program synthesis is to generate a pointer program $\mathcal{C}$ that satisfies P_{in} and P_{out}.*

Figure 1 illustrates the key idea of our approach. There are three modes after compilation, i.e., Command mode, Check mode and Abstraction mode. The order of their executions is reflected by black arrows. The Command mode contains a set of planning actions encoded from primitive command. We do not

encode the rearrangement rules into a separate phase. Instead, the rearrangement step is embedded in the encoding of $A(e)$. The actions in the Check mode are used to check the existence of an abstraction action that can fire. The abstraction rules are compiled into a set of abstraction actions.

Suppose the resulting FOND planning problem is $\mathcal{P} = (\mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A})$, where each component is described as follows. The initial state $\mathcal{I}$ and goal $\mathcal{G}$ are determined by P_{in} and P_{out} of specific synthesis problems.

Fluents: The set of fluents $\mathcal{F}$ is listed in Table 2, where int_0 represents the null value. Moreover, we use the fluents $command(), check(), choose(), abstraction()$ to represent different phases, and $abs_1(), abs_2(), abs_3()$ to denote which abstraction rule can be activated.

Table 2. Fluents of the encoded problem

Fluent	Meaning
$pvar(x)$	x is a program variable
$lvar(x')$	x' is a logical variable
$auxiliary(x')$	x' is a logical variable in use
$pt(x_1, x_2)$	A single heap cell
$ls(x_1, x_2)$	A linked list segment
$equal(x_1, x_2)$	Equality of x_1 and x_2
$var\text{-}num(x, v)$	Value of x is v in $\{int_i \mid 0 \leq i \leq n\}$
$active(v)$	Value v is allocated

Command Mode: Command mode is the first phase that a primitive command action is performed along with a possible rearrangement. Different from symbolic execution, here we do not distinguish rearrangement and command execution, only do rearrangement when it is needed. The encoded actions are shown in Table 3. For instance, there are two dispose actions corresponding to the dispose command, i.e., $dispose_1$ and $dispose_2$. The precondition of the former includes an explicit heap cell $pt(x_2, x_3)$, and that of the latter includes a list $ls(x_2, x_3)$ which needs to be unrolled as non-deterministic effects. The keyword "**oneof**" is used to express the non-deterministic effects in a planning model.

Table 3. Compilation of primitive commands

Action	Preconditions	Effects
$new(x_1, x_2, v_1, v_2)$	$command()$, $pvar(x_1)$, $lvar(x_2)$, $\neg auxiliary(x_2)$, $var\text{-}num(x_1, v_1)$, $var\text{-}num(x_2, int_0)$, $\neg active(v_2)$	$\neg command()$, $check()$, $pt(x_1, nil)$, $active(v_2)$, $var\text{-}num(x_1, v_2)$, $\neg var\text{-}num(x_1, v_1)$, $\forall y : \{equal(x_1, y)\} \rhd \{auxiliary(x_2), \neg equal(x_1, y), \neg equal(y, x_1), equal(x_2, y), equal(y, x_2), \neg var\text{-}num(x_2, int_0), var\text{-}num(x_2, v_1)\}$, $\forall y : \{pt(x_1, y)\} \rhd \{auxiliary(x_2), \neg pt(x_1, y), pt(x_2, y), \neg var\text{-}num(x_2, int_0), var\text{-}num(x_2, v_1)\}$, $\forall y : \{pt(y, x_1)\} \rhd \{auxiliary(x_2), \neg pt(y, x_1), pt(y, x_2), \neg var\text{-}num(x_2, int_0), var\text{-}num(x_2, v_1)\}$, $\forall y : \{ls(x_1, y)\} \rhd \{auxiliary(x_2), \neg ls(x_1, y), ls(x_2, y), \neg var\text{-}num(x_2, int_0), var\text{-}num(x_2, v_1)\}$, $\forall y : \{ls(y, x_1)\} \rhd \{auxiliary(x_2), \neg ls(y, x_1), ls(y, x_2), \neg var\text{-}num(x_2, int_0), var\text{-}num(x_2, v_1)\}$
$dispose_1(x_1, x_2, x_3, v)$	$command()$, $pvar(x_1)$, $active(v)$, $var\text{-}num(x_1, v)$, $var\text{-}num(x_2, v)$, $pt(x_2, x_3)$	$\neg command()$, $check()$, $\neg pt(x_2, x_3)$, $\neg active(v)$, $\forall y : \{var\text{-}num(y, v)\} \rhd \{\neg var\text{-}num(y, v), var\text{-}num(y, int_0)\}$
$dispose_2(x_1, x_2, x_3, x_4, v_1, v_2)$	$command()$, $pvar(x_1)$, $lvar(x_4)$, $active(v_1)$, $\neg active(v_2)$, $var\text{-}num(x_1, v_1)$, $var\text{-}num(x_2, v_1)$, $ls(x_2, x_3)$, $\neg auxiliary(x_4)$, $var\text{-}num(x_4, int_0)$	$\neg command()$, $check()$, $\neg ls(x_2, x_3)$, $\neg active(v_1)$, $\forall y : \{var\text{-}num(y, v_1)\} \rhd \{\neg var\text{-}num(y, v_1), var\text{-}num(y, int_0)\}$, $\mathbf{oneof}(\emptyset, \{auxiliary(x_4), ls(x_4, x_3), active(v_2), \neg var\text{-}num(x_4, int_0), var\text{-}num(x_4, v_2)\})$
$assign(x_1, x_2, x_3, v_1, v_2)$	$command()$, $pvar(x_1)$, $pvar(x_2)$, $lvar(x_3)$, $var\text{-}num(x_1, int_1)$, $var\text{-}num(x_2, int_2)$, $\neg auxiliary(x_3)$, $var\text{-}num(x_3, int_0)$	$\neg command()$, $check()$, $equal(x_1, x_2)$, $equal(x_2, x_1)$, $\{v_1 \neq v_2\} \rhd \{\neg var\text{-}num(x_1, v_1), var\text{-}num(x_1, v_2)\}$, $\forall y : \{equal(x_1, y)\} \rhd \{auxiliary(x_3), \neg equal(x_1, y), \neg equal(y, x_1), equal(x_3, y), equal(y, x_3), \neg var\text{-}num(x_3, int_0), var\text{-}num(x_3, v_1)\}$, $\forall y : \{pt(x_1, y)\} \rhd \{auxiliary(x_3), \neg pt(x_1, y), pt(x_3, y), \neg var\text{-}num(x_3, int_0), var\text{-}num(x_3, v_1)\}$, $\forall y : \{pt(y, x_1)\} \rhd \{auxiliary(x_3), \neg pt(y, x_1), pt(y, x_3), \neg var\text{-}num(x_3, int_0), var\text{-}num(x_3, v_1)\}$, $\forall y : \{ls(x_1, y)\} \rhd \{auxiliary(x_3), \neg ls(x_1, y), ls(x_3, y), \neg var\text{-}num(x_3, int_0), var\text{-}num(x_3, v_1)\}$, $\forall y : \{ls(y, x_1)\} \rhd \{auxiliary(x_3), \neg ls(y, x_1), ls(y, x_3), \neg var\text{-}num(x_3, int_0), var\text{-}num(x_3, v_1)\}$
$mutation_1(x_1, x_2, x_3, x_4, v)$	$command()$, $pvar(x_1)$, $pvar(x_4)$, $var\text{-}num(x_1, v)$, $var\text{-}num(x_2, v)$, $pt(x_2, x_3)$	$\neg command()$, $check()$, $\neg pt(x_2, x_3)$, $pt(x_2, x_4)$
$mutation_2(x_1, x_2, x_3, x_4, x_5, v_1, v_2)$	$command()$, $pvar(x_1)$, $pvar(x_4)$, $lvar(x_5)$, $\neg active(v_2)$, $var\text{-}num(x_1, v_1)$, $var\text{-}num(x_2, v_1)$, $ls(x_2, x_3)$, $\neg auxiliary(x_5)$, $var\text{-}num(x_5, int_0)$	$\neg command()$, $check()$, $\neg ls(x_2, x_3)$, $\mathbf{oneof}(\{pt(x_2, x_4)\}, \{auxiliary(x_5), pt(x_2, x_4), ls(x_5, x_3), active(v_2), \neg var\text{-}num(x_5, int_0), var\text{-}num(x_5, v_2)\})$

(*continued*)

Table 3. (*continued*)

Action	Preconditions	Effects
$lookup_1(x_1, x_2, x_3, x_4, x_5, v_1, v_2, v_3)$	$command()$, $pvar(x_1)$, $pvar(x_2)$, $lvar(x_5)$, $var\text{-}num(x_1, v_1)$, $var\text{-}num(x_4, v_2)$, $var\text{-}num(x_2, v_3)$, $var\text{-}num(x_3, v_3)$, $pt(x_3, x_4)$, $\neg auxiliary(x_5)$, $var\text{-}num(x_5, int_0)$	$\neg command()$, $check()$, $equal(x_1, x_4)$, $equal(x_4, x_1)$, $\{v_1 \neq v_2\} \triangleright \{\neg var\text{-}num(x_1, v_1), var\text{-}num(x_1, v_2)\}$, $\forall y : \{equal(x_1, y)\} \triangleright \{auxiliary(x_5), \neg equal(x_1, y)$, $\neg equal(y, x_1), equal(x_5, y), equal(y, x_5)$, $\neg var\text{-}num(x_5, int_0), var\text{-}num(x_5, v_1)\}$, $\forall y : \{pt(x_1, y)\} \triangleright \{auxiliary(x_5), \neg pt(x_1, y), pt(x_5, y)$, $\neg var\text{-}num(x_5, int_0), var\text{-}num(x_5, v_1)\}$, $\forall y : \{pt(y, x_1)\} \triangleright \{auxiliary(x_5), \neg pt(y, x_1), pt(y, x_5)$, $\neg var\text{-}num(x_5, int_0), var\text{-}num(x_5, v_1)\}$, $\forall y : \{ls(x_1, y)\} \triangleright \{auxiliary(x_5), \neg ls(x_1, y), ls(x_5, y)$, $\neg var\text{-}num(x_5, int_0), var\text{-}num(x_5, v_1)\}$, $\forall y : \{ls(y, x_1)\} \triangleright \{auxiliary(x_5), \neg ls(y, x_1), ls(y, x_5)$, $\neg var\text{-}num(x_5, int_0), var\text{-}num(x_5, v_1)\}$
$lookup_2(x_1, x_2, x_3, x_4, x_5, x_6, v_1, v_2, v_3, v_4)$	$command()$, $pvar(x_1)$, $pvar(x_2)$, $lvar(x_5)$, $lvar(x_6)$, $var\text{-}num(x_1, v_1)$, $var\text{-}num(x_4, v_2)$, $var\text{-}num(x_2, v_3)$, $var\text{-}num(x_3, v_3)$, $ls(x_3, x_4)$, $\neg active(v_4)$, $\neg auxiliary(x_5)$, $var\text{-}num(x_5, int_0)$, $\neg auxiliary(x_6)$, $var\text{-}num(x_6, int_0)$	$\neg command()$, $check()$, $\neg ls(x_3, x_4)$, $\{v_1 \neq v_5\} \triangleright \{\neg var\text{-}num(x_1, v_1), var\text{-}num(x_1, v_5)\}$, $\forall y : \{equal(x_1, y)\} \triangleright \{auxiliary(x_5), \neg equal(x_1, y)$, $\neg equal(y, x_1), equal(x_5, y), equal(x_5, y)$, $\neg var\text{-}num(x_5, int_0), var\text{-}num(x_5, v_1)\}$, $\forall y : \{pt(x_1, y)\} \triangleright \{auxiliary(x_5), \neg pt(x_1, y), pt(x_5, y)$, $\neg var\text{-}num(x_5, int_0), var\text{-}num(x_5, v_1)\}$, $\forall y : \{pt(y, x_1)\} \triangleright \{auxiliary(x_5), \neg pt(y, x_1), pt(y, x_5)$, $\neg var\text{-}num(x_5, int_0), var\text{-}num(x_5, v_1)\}$, $\forall y : \{ls(x_1, y)\} \triangleright \{auxiliary(x_5), \neg ls(x_1, y), ls(x_5, y)$, $\neg var\text{-}num(x_5, int_0), var\text{-}num(x_5, v_1)\}$, $\forall y : \{ls(y, x_1)\} \triangleright \{auxiliary(x_5), \neg ls(y, x_1), ls(y, x_5)$, $\neg var\text{-}num(x_5, int_0), var\text{-}num(x_5, v_1)\}$, **oneof**$(\{pt(x_3, x_4), equal(x_1, x_4), equal(x_4, x_1)$, $\neg var\text{-}num(x_1, v_1), var\text{-}num(x_1, v_2)\}, \{auxiliary(x_6)$, $active(v_4), equal(x_1, x_6), equal(x_6, x_1)$, $\neg var\text{-}num(x_1, v_1), var\text{-}num(x_1, v_4), pt(x_3, x_6)$, $ls(x_6, x_4), \neg var\text{-}num(x_6, int_0), var\text{-}num(x_6, v_4)\})$

Check Mode: The definition actions in Check mode are shown in Table 4. Action *check-act* alters the truth value of flags $abs_i(), i = 1, 2, 3$ according to the current state whenever a corresponding abstraction rule is enabled. Then the *choose-act* will be executed to determine the next phase to be switched with respect to $abs_i()$. When no abstraction rules can be applied, the next phase is the command mode, otherwise it is still turned to the abstraction mode.

Table 4. Actions in Check mode

Action	Preconditions	Effects
check-act()	*check*()	$\neg check(), choose(),$ $\{\exists x_1, x_2 : pvar(x_1) \wedge lvar(x_2) \wedge equal(x_1, x_2))\} \rhd \{abs_1()\},$ $\forall x_1, x_2, x_3 : \{pt(x_1, x_2), pt(x_2, x_3), auxiliary(x_2),$ $\exists v : var\text{-}num(x_1, v) \wedge \neg var\text{-}num(x_3, v), \forall y : \neg equal(y, x_2) \wedge$ $\neg ls(y, x_2) \wedge \neg ls(x_2, y), \forall y : y \neq x_1 \rightarrow \neg pt(y, x_2),$ $\forall y : y \neq x_3 \rightarrow \neg pt(x_2, y)\} \rhd \{abs_2()\},$ $\forall x_1, x_2, x_3 : \{pt(x_1, x_2), ls(x_2, x_3), auxiliary(x_2),$ $\exists v : var\text{-}num(x_1, v) \wedge \neg var\text{-}num(x_3, v), \forall y : \neg equal(y, x_2) \wedge$ $\neg ls(y, x_2) \wedge \neg pt(x_2, y), \forall y : y \neq x_1 \rightarrow \neg pt(y, x_2),$ $\forall y : y \neq x_3 \rightarrow \neg ls(x_2, y)\} \rhd \{abs_2()\},$ $\forall x_1, x_2, x_3 : \{ls(x_1, x_2), pt(x_2, x_3), auxiliary(x_2),$ $\exists v : var\text{-}num(x_1, v) \wedge \neg var\text{-}num(x_3, v), \forall y : \neg equal(y, x_2) \wedge$ $\neg pt(y, x_2) \wedge \neg ls(x_2, y), \forall y : y \neq x_1 \rightarrow \neg ls(y, x_2),$ $\forall y : y \neq x_3 \rightarrow \neg pt(x_2, y)\} \rhd \{abs_2()\},$ $\forall x_1, x_2, x_3 : \{ls(x_1, x_2), ls(x_2, x_3), auxiliary(x_2),$ $\exists v : var\text{-}num(x_1, v) \wedge \neg var\text{-}num(x_3, v), \forall y : \neg equal(y, x_2) \wedge$ $\neg ls(y, x_2) \wedge \neg ls(x_2, y), \forall y : y \neq x_1 \rightarrow \neg ls(y, x_2),$ $\forall y : y \neq x_3 \rightarrow \neg ls(x_2, y)\} \rhd \{abs_2()\},$ $\{\exists x : auxiliary(x) \wedge \forall y : \neg equal(x, y)) \wedge \neg pt(x, y) \wedge$ $\neg pt(y, x) \wedge \neg ls(x, y) \wedge \neg ls(y, x)\} \rhd \{abs_3()\},$
choose-act()	*choose*()	$\neg choose(), \neg abs_1(), \neg abs_2(), \neg abs_3(),$ $\{abs_1(), abs_2(), abs_3()\} \rhd \{command()\}$ $\{\neg abs_1(), \neg abs_2(), \neg abs_3()\} \rhd \{abstraction()\}$

Abstraction Mode: Table 5 shows the set of abstraction actions encoded from abstraction rules. The first two actions correspond to the abstraction rules. The last rule is used to free a logical variable in use, and make it available. When after applying an abstraction rule, we will go back to the Check mode until no abstraction rules can fire.

Table 5. Compilation of abstraction rules

Action	Preconditions	Effects
$abstract_1(x_1, x_2)$	$abstraction()$, $pvar(x_1)$, $auxiliary(x_2)$, $equal(x_1, x_2)$	$\neg abstraction(), check()$, $\neg equal(x_1, x_2), \neg equal(x_2, x_1)$, $\forall y : (equal(x_2, y) \wedge x_1 \neq y) \triangleright \{\neg equal(x_2, y), \neg equal(y, x_2), equal(x_1, y), equal(y, x_1)\}$, $\forall y : pt(x_2, y) \triangleright \{\neg pt(x_2, y), pt(x_1, y)\}$, $\forall y : pt(y, x_2) \triangleright \{\neg pt(y, x_2), pt(y, x_1)\}$, $\forall y : ls(x_2, y) \triangleright \{\neg ls(x_2, y), ls(x_1, y)\}$, $\forall y : ls(y, x_2) \triangleright \{\neg ls(y, x_2), ls(y, x_1)\}$
$abstract_2(x_1, x_2, x_3, v)$	$abstraction()$, $auxiliary(x_2)$, $var\text{-}num(x_1, v)$, $\neg var\text{-}num(x_3, v)$, $equal(x_3, nil) \vee x_3 = nil$, $(pt(x_1, x_2) \wedge pt(x_2, x_3)) \vee (pt(x_1, x_2) \wedge ls(x_2, x_3)) \vee (ls(x_1, x_2) \wedge pt(x_2, x_3)) \vee (ls(x_1, x_2) \wedge ls(x_2, x_3))$, $\forall y : \neg equal(y, x_2)$, $\forall y : (y \neq x_1 \wedge y \neq x_2 \wedge y \neq x_3) \rightarrow (\neg pt(x_2, y) \wedge \neg pt(y, x_2) \wedge \neg ls(x_2, y) \wedge \neg ls(y, x_2))$,	$\neg abstraction(), check()$, $ls(x_1, x_3)$, $pt(x_1, x_2) \triangleright \neg pt(x_1, x_2)$, $ls(x_1, x_2) \triangleright \neg ls(x_1, x_2)$, $pt(x_2, x_3) \triangleright \neg pt(x_2, x_3)$, $ls(x_2, x_3) \triangleright \neg ls(x_2, x_3)$
$abstract_3(x, v)$	$abstraction()$, $auxiliary(x)$, $var\text{-}num(x, v)$, $\forall y : \neg equal(x, y) \wedge \neg pt(x, y) \wedge \neg pt(y, x) \wedge \neg ls(x, y) \wedge \neg ls(y, x)$	$\neg abstraction(), check()$, $\neg auxiliary(x)$, $v \neq int_0 \triangleright var\text{-}num(x, int_0)$

6 Case Study and Experiment

In this section, we conduct a series of experiments to evaluate our approach. Further, we illustrate an example on synthesis of a disposal program which aims to dispose a linked list. The following code is the disposal program.

```
while (x ≠ nil) do {
  y := x;
  x := [x];
  dispose(y)
}
```

The initial state of this program is $ls(x, nil)$. Using the symbolic execution rules, the symbolic execution process is shown in Fig. 2. Note that the loop arrow from bottom to up means that $ls(x, nil)$ is the invariant of the program. At the beginning, we know x does not equal to nil since $ls(x, nil)$ at least contains one cell. Hence the first assignment command in the loop body is executed. When executing the second command, we do not know what holds in address x according to the spatial part $ls(x, nil)$ of the symbolic heap. Therefore a rearrangement step should be applied to distinguish the symbolic heap into a couple of situations. After executing $x := [x]$, we need to try to abstract the obtained symbolic heap since some primed variables are introduced. Then the last deallocation command is executed. In one situation, the loop exits. In the other, we find an invariant that is the same to the initial state. At this time the process terminates.

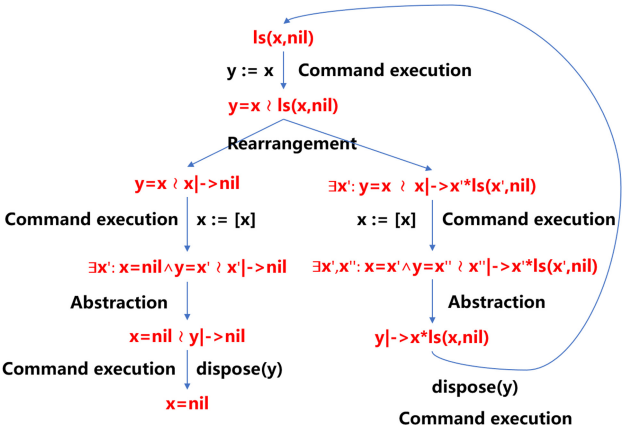

Fig. 2. Symbolic execution process for disposal program

In practice, we encode the program into standard PDDL which can be accepted by existing planners. The details are omitted here. We evaluate our approach on several list manipulated program. The non-deterministic planner PRP [18] is used as the FOND planner. Experiments are conducted on a laptop running Ubuntu 16.04 on an Intel® Core™ i7-8550U CPU 1.80 GHz and 8 GB of RAM. The results are shown in Table 6. "Insert" is to synthesize a program that inserts a cell before the head, "Remove" is the list disposal program, "Traverse" is to travel a list, "Append" is to append two lists into one. Sometimes we cannot synthesize a list program by our approach. For instance, reversing a list is impossible to be synthesized since the initial state and the goal are both an abstract list. Whether the list is reversed or not finally is never known. The second column is the search time in seconds. The third column is length of a policy. Note that the length includes additional actions (check actions and abstraction actions) except primitive command actions.

Table 6. Experimental results on list program

Program	Time(s)	Length
Insert	1.94	49
Remove	25.18	31
Traverse	16.56	25
Append	258.18	86

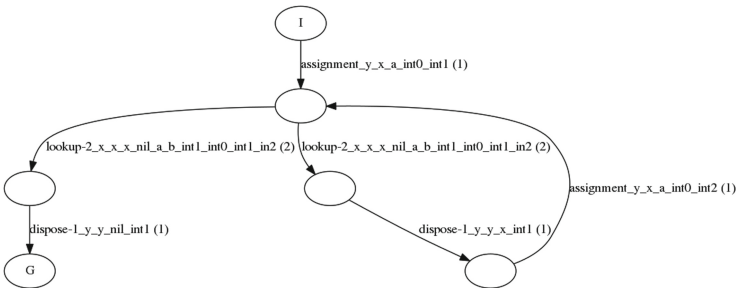

Fig. 3. Policy for synthesizing the disposal program

PRP generates a sequential plan for "Insert" and constructs loops for the other three programs. Consider the performance, we can see that synthesizing loop programs is not easy since the search time is much more than synthesizing sequential ones. More efforts needs to be made in order to find a loop for a planner. The essential reason is the quantifiers in the encoded actions especially in the effects of abstractions and preconditions of check actions. In practice, the quantifiers are expanded that will result in the explosion of the state space.

The solution generated for disposal program is shown in Fig. 3. We preserve the command actions and remove rest. Therefore, we obtain a compact policy much closer to a pointer program. Obviously, the policy is similar to the disposal program mentioned before. The boolean condition at the start of the loop must be specified manually.

7 Conclusion

Synthesizing programs is a difficult and central problem in computer science. In this paper, we propose an automated planning based method for pointer program synthesis. Inspired by symbolic execution of separation logic, we compile this process into a FOND planning problem, mainly including primitive command compilation, rearrangement compilation and abstraction compilation. In future work, we plan to synthesize larger scale programs. This is feasible from the theoretical point of view because of the modular reasoning feature of separation logic. Furthermore, we have to find a way that can reduce the number of quantifiers in the encoded actions to improve the performance of planners.

References

1. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: Deepcoder: learning to write programs. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, 24–26 April 2017, Conference Track Proceedings. OpenReview.net (2017). https://openreview.net/forum?id=ByldLrqlx
2. Beltramelli, T.: pix2code: generating code from a graphical user interface screenshot. In: Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS 2018, Paris, France, 19–22 June 2018. pp. 3:1–3:6. ACM (2018). https://doi.org/10.1145/3220134.3220135
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, 16–18 December 2004, Proceedings, pp. 97–109 (2004). https://doi.org/10.1007/978-3-540-30538-5_9
4. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, 2–5 November 2005, Proceedings, pp. 52–68 (2005). https://doi.org/10.1007/11575467_5
5. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. J. Comput. Syst. Sci. **78**(3), 911–938 (2012). https://doi.org/10.1016/j.jcss.2011.08.007
6. Calcagno, C., et al.: Moving fast with software verification. In: Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, 27–29 April 2015, Proceedings. Lecture Notes in Computer Science, vol. 9058, pp. 3–11. Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
7. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26:1–26:66 (2011). https://doi.org/10.1145/2049697.2049700
8. Camacho, A., Bienvenu, M., McIlraith, S.A.: Towards a unified view of AI planning and reactive synthesis. In: Benton, J., Lipovetzky, N., Onaindia, E., Smith, D.E., Srivastava, S. (eds.) Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, 11–15 July 2019, pp. 58–67. AAAI Press (2019). https://aaai.org/ojs/index.php/ICAPS/article/view/3460
9. Cimatti, A., Pistore, M., Roveri, M., Traverso, P.: Weak, strong, and strong cyclic planning via symbolic model checking. Artif. Intell. **147**(1–2), 35–84 (2003). https://doi.org/10.1016/S0004-3702(02)00374-0
10. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings. Lecture Notes in Computer Science, vol. 3920, pp. 287–302. Springer (2006). https://doi.org/10.1007/11691372_19
11. Fox, M., Long, D.: PDDL2.1: an extension to PDDL for expressing temporal planning domains. J. Artif. Intell. Res. **20**, 61–124 (2003). https://doi.org/10.1613/jair.1129

12. Fu, J., Bastani, F.B., Yen, I.: Automated AI planning and code pattern based code synthesis. In: 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2006), 13–15 November 2006, Arlington, VA, USA, pp. 540–546. IEEE Computer Society (2006). https://doi.org/10.1109/ICTAI.2006.37
13. Ghallab, M., Nau, D.S., Traverso, P.: Automated Planning - Theory and Practice. Elsevier (2004)
14. Gu, X., Zhang, H., Zhang, D., Kim, S.: Deep API learning. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, 13–18 November 2016, pp. 631–642. ACM (2016). https://doi.org/10.1145/2950290.2950334
15. Gulwani, S., Polozov, O., Singh, R.: Program synthesis. Found. Trends Program. Lang. **4**(1–2), 1–119 (2017). https://doi.org/10.1561/2500000010
16. Kitzelmann, E.: Inductive programming: A survey of program synthesis techniques. In: Schmid, U., Kitzelmann, E., Plasmeijer, R. (eds.) Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009, Edinburgh, UK, 4 September 2009. Revised Papers. Lecture Notes in Computer Science, vol. 5812, pp. 50–73. Springer (2009). https://doi.org/10.1007/978-3-642-11931-6_3
17. Magill, S., Nanevski, A., Clarke, E., Lee, P.: Inferring invariants in separation logic for imperative list-processing programs. SPACE **1**(1), 5–7 (2006)
18. Muise, C.J., McIlraith, S.A., Beck, J.C.: Improved non-deterministic planning by exploiting state relevance. In: McCluskey, L., Williams, B.C., Silva, J.R., Bonet, B. (eds.) Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, 25–19 June 2012. AAAI (2012). http://www.aaai.org/ocs/index.php/ICAPS/ICAPS12/paper/view/4718
19. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22–25 July 2002, Copenhagen, Denmark, Proceedings, pp. 55–74 (2002). https://doi.org/10.1109/LICS.2002.1029817
20. Vanneschi, L., Poli, R.: Genetic programming - introduction, applications, theory and open issues. In: Rozenberg, G., Bäck, T., Kok, J.N. (eds.) Handbook of Natural Computing, pp. 709–739. Springer (2012). https://doi.org/10.1007/978-3-540-92910-9_24