December 10, 2012

Verification of Software and Hardware using Quantified Boolean Formulas (QBF)

William Klieber

Thesis Proposal

November 2012

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Edmund M. Clarke, chair
Randal E. Bryant
Jeannette M. Wing
João P. Marques-Silva (University College Dublin)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Semiconductor Research Corporation, by a Google Research Award, and by Portugal Fundação para a Ciência e a Tecnologia (FCT) grant ATTEST (CMU-PT/ELE/0009/2009).

December 10, 2012

December 10, 2012

Abstract

Many problems in formal verification of digital hardware circuits and other finite-state systems are naturally expressed in the language of quantified boolean formulas (QBF). The first two parts of this thesis proposal present techniques that advance the state-of-the-art in solving such QBF problems, thereby enabling the verification of more complex hardware designs. The third part proposes a new technique for software verification using a solver for QBF with free variables.

Traditionally, QBF solvers have required that their input formulas be transformed into a special form known as *prenex CNF*. However, although prenex CNF has the benefit of being simple, it is now recognized that transformation to this form can be detrimental to advanced solvers because it obscures features of the input formula that could be useful to the solver. We present two contributions to the development of non-prenex, non-CNF solvers. First, we reformulate *clause/cube learning*, an important technique in prenex solvers, and we extend it to non-prenex instances. Second, we introduce a propagation technique using *ghost literals* that exploits the structure of a non-CNF instance in a manner that is symmetric between the universal and existential variables.

The second part of this thesis proposal discusses an approach to QBF using Counterexample-Guided Abstraction Refinement (CEGAR). The approach recursively solves QBF instances with multiple quantifier alternations. Experimental results show that the CEGAR-based solver outperforms existing types of solvers on many publicly-available benchmark families. In addition, we present a method of combining the CEGAR technique with DPLL-based solvers and show that it improves the DPLL solver in many instances.

The third part of this thesis proposal presents a method for automatically inferring universally quantified loop invariants for programs with dynamically allocated heap data structures. Our technique works by computing an overapproximation of the set of reachable states via a fixed-point procedure. We target a small dynamically typed intermediate language. Sets of states are described by formulas in a fragment of first-order logic augmented with transitive closure; the fragment includes equality, uninterpreted functions, and total order. We introduce an abstraction function that summarizes the heap memory, returning a formula of bounded size. Summarization of memory locations is based, in part, on how they can be reached from the program variables. The inferred invariants can be used to verify the absence of failed assertions and other run-time errors.

December 10, 2012

Contents

Chapter 1

Introduction

Many problems in formal verification (among other areas) are naturally expressed in the language of Quantified Boolean Formulas (QBF). QBF is an extension of propositional logic in which boolean variables can be quantified. Syntactically, we consider QBF formulas described by the following grammar:

$$\Phi \quad ::= \quad \mathsf{True} \;\mid\; \mathsf{False} \;\mid\; x \;\mid\; \neg\Phi \;\mid\; \Phi \vee \Phi \;\mid\; \Phi \wedge \Phi \;\mid\; \exists x\ \Phi \;\mid\; \forall x\ \Phi$$

Additionally, to simplify matters, we disallow formulas such as $\exists x.\, \exists x.\, \Phi$ in which one binding of a variable shadows another.

A *literal* is a variable or its negation. We represent an assignment to boolean variables by a set of literals, as follows: An assignment π assigns x true iff $x \in \pi$, and an assignment π assigns x false iff $\neg x \in \pi$. We write "$\pi(\ell)$" to denote the value (true, false, or undef) that π assigns to ℓ, as follows: $\pi(\ell) = \mathsf{true}$ if $\ell \in \pi$, $\pi(\ell) = \mathsf{false}$ if $\neg\ell \in \pi$, and $\pi(\ell) = \mathsf{undef}$ otherwise. For any variable x, we treat $\neg\neg x$ as equivalent to x. An assignment may not include both a variable and its negation.

Definition 1 (Reduction). The *reduction* of a formula f under an assignment π, denoted by "$f|\pi$", is constructed from f as follows: For each variable x which is assigned a value by π, we delete the quantifier of x (if any) and replace each occurrence of x with its assigned value. E.g., if $\pi = \{e_1\}$, then $[\exists e_1.\, \forall u_2.\, (e_1 \wedge u_2)]|\pi = [\forall u_2.\, (\mathsf{true} \wedge u_2)]$. Formally:

$$\ell|\pi = \begin{cases} \pi(\ell) & \text{if } \pi(\ell) \neq \mathsf{undef} \\ \ell & \text{if } \pi(\ell) = \mathsf{undef} \end{cases} \qquad (\exists x.f)|\pi = \begin{cases} f|\pi & \text{if } \pi(x) \neq \mathsf{undef} \\ \exists x.(f|\pi) & \text{if } \pi(x) = \mathsf{undef} \end{cases}$$

$$(f_1 \wedge \ldots \wedge f_n)|\pi = (f_1|\pi) \wedge \ldots \wedge (f_n|\pi)$$

$$(f_1 \vee \ldots \vee f_n)|\pi = (f_1|\pi) \vee \ldots \vee (f_n|\pi) \qquad (\forall x.f)|\pi = \begin{cases} f|\pi & \text{if } \pi(x) \neq \mathsf{undef} \\ \forall x.(f|\pi) & \text{if } \pi(x) = \mathsf{undef} \end{cases}$$

Semantically, boolean quantifiers are defined as follows:

- Universal quantifier: $\forall x.\,\Phi \;=\; \Phi|\{x\} \wedge \Phi|\{\neg x\}$
- Existential quantifier: $\exists x.\,\Phi \;=\; \Phi|\{x\} \vee \Phi|\{\neg x\}$

A QBF instance is *closed* iff every occurrence of every variable is bound by a quantifier. In the next two chapter, we will only consider closed instances.

A boolean formula in *conjunctive normal form (CNF)* is a conjunction of *clauses*, where a clause is a disjunction of literals. Whenever convenient, a CNF formula is treated as a set of clauses.

Given two literals x and y, we say that x is *upstream* of y iff the scope of the quantifier of x contains the quantifier of y. If a literal x is upstream of another literal y, then y is *downstream* of x.

For a literal ℓ, $\mathsf{var}(\ell)$ denotes the variable in ℓ, i.e. $\mathsf{var}(\neg x) = \mathsf{var}(x) = x$.

1.1 QBF as a Two-Player Game

It is helpful to view QBF as a game between two players, Player $\exists$ and Player $\forall$. We make the following definitions:

- The existentially quantified variables are *owned* by Player $\exists$.
- The universally quantified variables are *owned* by Player $\forall$.

Informally, the game formulation goes as follows. Throughout the course of the game, the two players assign values to the variables that they own. The order in which the players assign variables is the quantification order of the variables. On each turn of the game, the owner of an outermost-quantified unassigned variable assigns it a value. The goal of Player $\exists$ is to make the formula true, and the goal of Player $\forall$ is to make the formula false.

Definition 2 (Winning under an assignment).

- Player $\exists$ *wins* a formula f under π iff $f|\pi$ is true.
- Player $\forall$ *wins* a formula f under π iff $f|\pi$ is false.

Chapter 2

Game-State Learning and Ghost Literals in QBF

2.1 Introduction

Traditionally, QBF solvers have used conjunctive normal form (CNF). Although CNF works well for SAT solvers, it hinders the work of QBF solvers by impeding the ability to detect and learn from satisfying assignments. In fact, a family of problems that are trivially satisfiable in negation-normal form (NNF) were experimentally found to require exponential time (in the problem size) for existing CNF solvers [40].

Various techniques have been proposed for avoiding the drawbacks of a CNF encoding. Zhang et al. have investigated dual CNF-DNF representations in which a boolean formula is transformed into a combination of an equi-satisfiable CNF formula and an equi-tautological DNF [40]. Sabharwal et al. have developed a QBF modeling approach based a game-theoretic view of QBF [34]. Ansotegui et al. have investigated the use of *indicator variables* [1]. These approaches all help to alleviate the problems of a pure CNF encoding, but we argue that a fully non-clausal approach can lead to even greater improvements, especially for instances produced from deeply-nested circuits.

In addition to combined CNF-DNF techniques, fully non-clausal techniques have recently been investigated. A prenex circuit-based DPLL solver with "don't care" reasoning and clause/cube learning has been developed by Goultiaeva et al. [17]. A non-prenex NNF-based DPLL solver with dependency-directed (non-chronological) backtracking, but without learning, was developed by Egly, Seidl, and Woltran [11]. Non-clausal techniques using symbolic quantifier expansion (rather than DPLL) have been developed by Lonsing and Biere [28] and by Pigorsch and Scholl [31].

Giunchiglia et al. have developed a technique for mini-scoping quantifiers (pushing quantifiers inward so as to minimize their scope) [15]. Non-clausal representations have also been investigated in the context of SAT solvers [13, 18, 39].

Most existing DPLL-based QBF solvers perform clause/cube learning. However, traditional clause/cube learning was designed for prenex QBF instances, and it is not optimal for (or even directly applicable to) non-prenex QBF instances. We reformulate clause/cube learning and extend it to the non-prenex case. Additionally, we develop a new propagation technique using *ghost literals*. Experimental results indicate that our approach can beat other state-of-the-art solvers on fixed-point computation instances of the type found in the `tipfixpoint` benchmark family.

2.2 Preliminaries

In this chapter, we assume that existentially quantified variables have the form e_i and universally quantified have the form u_i, where i is a positive integer. We label each conjunction and disjunction of the input QBF with a *gate variable* of the form g_i, as illustrated in Figure 2.1. The conjunction/disjunction labelled g_i, together with its quantifier prefix (if any), is labelled with the primed gate variable g_i', as illustrated in Figure 2.1. As indicated in the abstract grammar, each labelled conjunction/disjunction may have any number of conjuncts/disjuncts.

$$\exists e_{10}\ \Big[\underbrace{\big[\exists e_{11}\ \forall u_{21}\ \overbrace{(e_{10} \wedge e_{11} \wedge u_{21})}^{g_1}\big]}_{g_1'} \wedge \underbrace{\big[\forall u_{22}\ \exists e_{30}\ \overbrace{(e_{10} \wedge u_{22} \wedge e_{30})}^{g_2}\big]}_{g_2'}\Big]$$

Figure 2.1: Example QBF instance with gate labels.

The term "gate variable" arises from the circuit representation of a propositional formula, in which a gate variable labels a logic gate.

Let "Φ_{in}" denote the formula that the QBF solver is given as input. We impose the following restriction on Φ_{in}: Every variable in Φ_{in} must be quantified exactly once, and no variable may occur free (i.e., outside the scope of its quantifier). The variables that occur in Φ_{in} are said to be *input variables*. An *input assignment* is an assignment in which every assigned variable is an input variable (as opposed to a gate variable). We say that a gate literal g is *upstream* of an input literal y iff every variable that occurs in the subformula g is upstream of y.

For non-prenex instances, we say that each quantifier-prefixed subformula (e.g., g_1' and g_2' in Figure 2.1) is a *subgame*. It may happen that two or more variables are quantified outermost; e.g., in Figure 2.1 on page 4, after e_{10} is assigned a value, both e_{11} and u_{22} are quantified outermost. In this case, two subgames have become independent of each other; they may be played in parallel.

2.3 Symbolic Game States

In this section, we introduce *game-state learning*, a reformulation of clause/cube learning. For prenex instances, the game-state formulation is isomorphic to clause/cube learning; the differences are merely cosmetic. However, the game-state formulation is more convenient to extend to the non-prenex case.

To motivate the notation of game-state learning, we start by reviewing certain aspects of clause learning. Suppose the input formula Φ_{in} is a prenex CNF QBF whose first clause is $(e_1 \vee e_3 \vee u_4 \vee e_5)$. Under an assignment π, if all the literals in the clause are false, then clearly $\Phi_{\text{in}}|\pi$ is false. Moreover, if, under π, all the clause's existential literals are assigned false and none of the clause's universal literals are assigned true (i.e., they may either be assigned false or be unassigned), then $\Phi_{\text{in}}|\pi$ is false, since the universal player can win by making all the universal literals in the clause false.

As shown in [41], when the QBF clause learning algorithm is applied to

$$\exists e_1 \exists e_3 \forall u_4 \exists e_5 \exists e_7.\, (e_1 \vee e_3 \vee u_4 \vee e_5) \wedge (e_1 \vee \neg e_3 \vee \neg u_4 \vee e_7) \wedge ...$$

it can yield the tautological learned clause $(e_1 \vee u_4 \vee \neg u_4 \vee e_5 \vee e_7)$. Although counter-intuitive, this learned clause can be interpreted in the same way as a non-tautological clause: Under an assignment π, if all the clause's existential literals are assigned false and none of the clause's universal literals are assigned true, then $\Phi_{\text{in}}|\pi$ is false.

Learned cubes are similar: Under an assignment π, if all the cube's universal literals are assigned true and none of the cube's existential literals are assigned false, then $\Phi_{\text{in}}|\pi$ is true. With game-state learning, we explicitly separate the "must be true" literals from the "may be either true or unassigned" literals. (For non-prenex instances, the division is more complicated than just existential-vs-universal.) Instead of writing a cube $(e_1 \vee u_2 \vee \neg e_3)$, we will write a game-state sequent $\langle \{u_2\}, \{e_1, \neg e_3\} \rangle \models (\exists \text{ wins } \Phi_{\text{in}})$.

Definition 3. A *symbolic game state* is a tuple $\langle L^{\text{now}}, L^{\text{fut}} \rangle$, where L^{now} is a set of literals and L^{fut} is a set of input literals. $\langle L^{\text{now}}, L^{\text{fut}} \rangle$ symbolically represents (or *matches*) exactly those input assignments under which:

1. every literal in L^{now} reduces to `true`, and
2. no literal in L^{fut} is assigned `false` — i.e., for every literal ℓ in L^{fut}, either ℓ is already true or ℓ has not yet been assigned a value (and therefore may become true in the future).

For example, consider again the QBF instance in Figure 2.1 on page 4. The assignment $\{\neg e_{10}\}$ matches both $\langle \{\neg g'_1\}, \varnothing \rangle$ and $\langle \{\neg g'_1\}, \{u_{21}, \neg u_{21}\} \rangle$ (because $\neg e_{10}$ implies $\neg g'_1$), but not $\langle \{\neg g'_1\}, \{e_{10}\} \rangle$. No assignment matches $\langle \{\neg e_{10}\}, \{e_{10}\} \rangle$.

Definition 4 (Winning under a game state). We say that player P *wins* a formula f under a game state GS, written "$GS \models (P \text{ wins } f)$", iff P wins f under all assignments that match GS. Additionally, we say that P *loses* f under GS, written "$GS \models (P \text{ loses } f)$", iff the opponent of P wins f under GS.

For example, for the QBF instance in Figure 2.1:

- Neither player wins g'_1 under the game state $\langle \varnothing, \varnothing \rangle$, because Player $\forall$ loses under the matching assignment $\{e_{10}, e_{11}, u_{21}\}$ and Player $\exists$ loses under the matching assignment $\{\neg e_{10}\}$.
- Player $\forall$ wins g'_1 under $\langle \varnothing, \{\neg u_{21}\} \rangle$. For example, under the assignment $\pi = \{e_{11}\}$, $g'_1|\pi$ is $[\forall u_{21}\, (e_{10} \wedge \texttt{true} \wedge u_{21})]$, which evaluates to `false`.
- Player $\exists$ wins g'_1 under $\langle \{u_{21}\}, \{e_{10}, e_{11}\} \rangle$.

In our solver, instead of learning clauses or cubes, we maintain a game-state database with *sequents* of the form $GS \models (P \text{ wins } g'_i)$. It turns out that whenever we learn a new game-state sequent for a prenex instance, the literals owned by the winner all go in L^{fut}, and the literals owned by the loser and the gate literals go in L^{now}. The relationship between learned game-state sequents and learned clauses/cubes (for prenex instances) is as follows. $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\forall \text{ wins } \Phi_{\text{in}})$ is equivalent to the learned clause $[\neg\ell_1 \vee ... \vee \neg\ell_n]$ where $\{\ell_1, ..., \ell_n\} = L^{\text{now}} \cup L^{\text{fut}}$ (where L^{now} contains the loser/gate literals and L^{fut} contains the winner literals). This equivalence is easily verified using the interpretation of learned clauses developed on the previous page. Likewise, $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\exists \text{ wins } \Phi_{\text{in}})$ is equivalent to the learned cube $[\ell_1 \wedge ... \wedge \ell_n]$ where $\{\ell_1, ..., \ell_n\} = L^{\text{now}} \cup L^{\text{fut}}$.

Proposition 1. If $\langle L^{\text{now}} \cup \{\ell\},\, L^{\text{fut}} \rangle \models (P \text{ wins } f)$, and ℓ is owned by Player P and the quantifier of ℓ is inside f, then $\langle L^{\text{now}},\, L^{\text{fut}} \cup \{\ell\} \rangle \models (P \text{ wins } f)$, provided that $\neg\ell \notin L^{\text{fut}}$.

For example, consider the QBF instance $\forall u_1.\, \exists e_2.\, (u_1 \oplus e_2)$, where "$u_1 \oplus e_2$" means "$(u_1 \wedge \neg e_2) \vee (\neg u_1 \wedge e_2)$". If Player $\exists$ wins under $\langle \{u_1, \neg e_2\}, \varnothing \rangle$, then Proposition 1 tells us that Player $\exists$ wins under $\langle \{u_1\}, \{\neg e_2\} \rangle$.

2.4 Algorithm

An overview of the top-level solver algorithm is provided in Figure 2.2. Initially, the current assignment *CurAsgn* is empty. For non-prenex instances, we may temporarily target in on a subgame of the input formula Φ_{in} and ignore the rest; the subgame being targetted is recorded in the *TargFmla* global variable. On each iteration of the main loop, we first test to see if we know who wins *TargFmla* under the current assignment. There are two cases:

- If the winner of *TargFmla* is unknown, then we call `DecideLit`, which picks an unassigned input variable (from the first available quantifier block in the prefix of *TargFmla*) and assigns it a value in *CurAsgn*. If there are no more unassigned variables in the quantifier prefix of the current *TargFmla*, then we pick a new *TargFmla* from among the unassigned immediate subformulas of *TargFmla* and try again. After adding a new literal to *CurAsgn*, we call `Propagate` to perform boolean constraint propagation (BCP).

- If the winner is known, then we call `LearnNewGS` to learn a new game-state sequent, adding it to the database. If the new game-state sequent reveals that Φ_{in} evaluates to a value v under the empty assignment, then we return v as our final answer. Otherwise, we backtrack. We follow the well-known non-chronological backtracking technique, with the addition that we must also undo changes to *TargFmla* as appropriate. (That is, if we backtrack to the beginning of the k^{th} decision level, then we must restore *TargFmla* to the value that it held at the beginning of the k^{th} decision level. For this purpose, we maintain an array `UndoTarg` that maps each decision level to the value of *TargFmla* to be restored.) After backtracking, the newly-learned game-state sequent will force a literal, so we call `Propagate` to perform BCP. (Is a literal forced even when we leave a subgame b by restoring an old value of *TargFmla* during backtracking? Yes; ghosts of b are forced.)

```
func Solve() {
    CurAsgn = ∅;
    TargFmla = Φ_in;
    while (true) {
        while (the winner of TargFmla under CurAsgn is unknown) {
            DecideLit();   // Picks new TargFmla if necessary.
            Propagate();
        }
        GS = LearnNewGS();
        if (TargFmla == Φ_in and ∅ matches GS) return winner;
        Backtrack to the earliest point at which GS will force a literal;
        Propagate();
        }
    }
}
```

Figure 2.2: Overview of top-level solver algorithm.

2.4.1 Ghost Literals

Goultiaeva et al. [17] introduce a powerful propagation technique for QBF that significantly improves on existing QBF solvers on a variety of benchmarks. With their technique, if the solver notices that a gate literal g must be true in order for the existential player to win, then g becomes forced. However, this technique is asymmetric between the existential and universal players. A gate literal g is forced if it is needed for the existential player to win, but not if it is needed for the universal player to win. We adapt this technique so that the universal variables benefit from the same propagation technique as do the existential variables and so that the learning procedure for satisfying assignments is just as powerful as for falsifying assignments.

In a prenex solver, for each gate variable g, we would introduce two *ghost* variables, $g\langle\forall\rangle$ for Player $\forall$ and $g\langle\exists\rangle$ for Player $\exists$. A ghost literal $g\langle P\rangle$ would be forced whenever we detect that Player P cannot win unless g is made true.

For our non-prenex solver, we need to consider subgames (quantifier-prefixed subformulas, such as g'_1 and g'_2 in Figure 2.1). We introduce ghost variables of the form $g\langle\forall, b\rangle$ and $g\langle\exists, b\rangle$ where b is a subgame which contains g as a subformula. A ghost literal $g\langle P, b\rangle$ becomes forced when we detect that Player P cannot win subgame b without g being true. For example, consider the below QBF instance (where g_1 is some propositional formula involving e_1, u_2, and e_3):

$$\exists e_1\, \forall u_2\, \exists e_3\, \forall u_4.\ [\underbrace{[\forall u_5.\ g_1 \vee u_5]}_{g_2'} \wedge u_4] \vee \underbrace{[\forall u_6.\ \neg g_1 \vee u_6]}_{g_3'}$$

Under the empty assignment, $g_1\langle \exists, g_2' \rangle$ is forced (because Player $\exists$ cannot win g_2' under $\varnothing$ unless g_1 is true) and likewise $\neg g_1\langle \exists, g_3' \rangle$ is forced.

2.4.2 Propagation and Learning

The algorithms for propagation and learning are adapted from the existing techniques for DPLL solvers (e.g., [42]). Details are presented in [24].

2.5 Experimental Results

We implemented the ideas in this chapter in a solver which we call *GhostQ*. In our experimental results, GhostQ always did at least as well as CirQit and it outperformed Qube on the `k`, `tipdiam`, and `tipfixpoint` families.

We ran GhostQ on the non-CNF instances from QBFLIB on 2.66 GHz machine with a timeout of 300 seconds. For comparison we show the results for CirQit published in [17] (which were conducted on a 2.8 GHz machine with a timeout of 1200 seconds). (CirQit is not publicly available.) As shown in Table 2.1, GhostQ performs better CirQit on every benchmark family except `consistency`. The `ring` and `semaphore` families consist of prenex instances. The other families are non-prenex, so our solver took advantage of its ability to perform non-prenex game-state learning. During testing of our solver, it was noted that non-prenex learning was especially helpful on the `dme` family.[1]

We compared GhostQ to the state-of-the-art solvers Qube 6.6 [15], Quantor 3.0 [4], and sKizzo 0.8.2 [3]. We ran these solvers on the QBFLIB QBFEVAL 2007 benchmarks [30] on a 2.66 GHz machine, with a time limit of 60 seconds and a memory limit of 1 GB. The results are shown in Tables 2.2 and 2.3. We also show the results for AIGsolve published in [31], but these numbers are not directly comparable because they were obtained on a different machine and with a timeout of 600 s.

[1]The `dme` family instances were originally given in prenex form, but we pushed the quantifiers inward as a preprocessing step. The unprenexing time was about 0.8 seconds per instance and is included in our solver's total time shown in the table.

Table 2.1: Comparison between GhostQ and CirQit.

Family	inst.	GhostQ		CirQit	
Seidl	*150*	**150**	(1606 s)	147	(2281 s)
assertion	*120*	**12**	(141 s)	3	(1 s)
consistency	*10*	0	(0 s)	0	(0 s)
counter	*45*	**40**	(370 s)	39	(1315 s)
dme	*11*	**11**	(13 s)	10	(15 s)
possibility	*120*	**14**	(274 s)	10	(1707 s)
ring	*20*	**18**	(28 s)	15	(60 s)
semaphore	*16*	**16**	(4 s)	16	(7 s)
Total	*492*	261	(2435 s)	240	(5389 s)

Table 2.2: Comparison between GhostQ and Qube.

Family	inst.	GhostQ		Qube	
bbox-01x	*450*	171	(133 s)	**341**	(1192 s)
bbox_design	*28*	19	(256 s)	**28**	(15 s)
bmc	*132*	43	(266 s)	**49**	(239 s)
k	*61*	**42**	(355 s)	13	(55 s)
s	*10*	10	(1 s)	10	(5 s)
tipdiam	*85*	**72**	(143 s)	60	(235 s)
tipfixpoint	*196*	**165**	(503 s)	100	(543 s)
sort_net	*53*	0	(0 s)	**19**	(176 s)
all other	*121*	9	(38 s)	**23**	(227 s)
Total	*1136*	531	(1695 s)	643	(2687 s)

Table 2.3: Comparison between GhostQ and Non-DPLL Solvers.

		Timeout 60 s			Timeout 600 s	
Family	inst.	GhostQ	Quantor	sKizzo	GhostQ	AIGsolve
bbox-01x	*450*	171	130	166	178	173
bbox_design	*28*	19	0	0	22	23
bmc	*132*	43	106	83	51	30
k	*61*	42	37	47	51	56
s	*10*	10	8	8	10	10
tipdiam	*85*	72	23	35	72	77
tipfixpoint	*196*	165	8	25	170	133
sort_net	*53*	0	27	1	0	0
all other	*121*	9	49	31	17	35
Total	*1136*	531	388	396	571	537

In Tables 1–2, we give the number of instances solved and the time needed to solve them. (Times shown do not include time spent trying to solve instances where the solver timed out.) In Table 3, we give the number of instances solved.

For the CNF benchmarks, we wrote a script to reverse-engineer the QDIMACS file to circuit form and convert it to our solver's input format. (This is similar to the technique in [31], but we also looked for "if-then-else" gates of the form $g = (x\,?\,y : z)$.) Of the four other solvers shown in Tables 2.2 and 2.3, Qube is the only other DPLL-based solver, so it is most similar to our solver. Our experimental results show that GhostQ does better than Qube on the `tipdiam` and `tipfixpoint` families (which concern diameter and fixpoint calculations for model checking problems on the TIP benchmarks) and on the `k` family.

The use of ghost literals can help GhostQ in two ways: (1) By treating the gate literals specially instead of treating them as belonging to the existential player, we can more readily detect satisfactions and we can learn more powerful cubes; (2) By using universal ghost literals, we have a more powerful propagation procedure for the universal input literals. (We did not perform unprenexing on any of the originally-CNF benchmarks, so our use of game-state learning doesn't improve performance here.) To further investigate, we turned off downward propagation of universal ghost literals; on most families the effect was negligible, but on `tipfixpoint` we solved only 149 instances instead of 165.

2.6 Conclusion

In this chapter, we have made two contributions. First, we have introduced the concept of *symbolic game states* and used this concept to reformulate clause/cube learning and extend it to the non-prenex case. Using game states, we have also been able to reformulate the techniques for conflict/satisfaction analysis, BCP, and non-chronological backtracking. In all cases, we give a unified presentation which is applicable to both the existential and universal players, instead of using separate terminology and notation for the two players. Further, game states are 'well-behaved' theoretically, in that we no longer need learn and store tautological clauses (or contradictory cubes). Our second contribution is introducing the concept of *ghost literals*, allowing us to improve upon the propagation technique introduced in [17] by eliminating the asymmetry between the players so that the technique can reduce the search space for both the universal and existential players (instead of only the existential player). Experiments show that our techniques work particularly well on certain benchmarks related to formal verification. For future work, it may be worthwhile to investigate whether the ideas of dynamic partitioning [36] can be extended to allow dynamic unprenexing.

Chapter 3

Counterexample-Guided Abstraction Refinement (CEGAR) in QBF

3.1 Introduction

A number of approaches have been proposed for QBF, including (Q)DPLL (e.g., [16]), expansion [2, 4, 28], and Skolemization [3]. This chapter presents a new approach by M. Janota, W. Klieber, J. Marques-Silva, and E. Clarke [19]. It employs Counterexample-Guided Abstraction Refinement (CEGAR) [8] to gradually expand the input formula. The CEGAR approach differs from traditional expansion-based solvers in how the expansion is performed. For a quantifier block of n variables, traditional expansion-based solvers perform up to n expansions (one for each variable), and the formula grows exponentially with the number of expansions performed (in the worst case). In contrast, the CEGAR approach performs up to 2^n partial expansions (one for each possible assignment to all n variables), but the formula grows only linearly with the number of partial expansions performed. In practice, often only a relatively small number of partial expansions are needed, allowing the CEGAR approach to solve instances on which traditional expansion-based solvers run out of memory.

3.2 Preliminaries

We write "$\bar{Q}$" to denote "$\forall$" (if Q is "$\exists$") or "$\exists$" (if Q is "$\forall$").

We write "$moves(X)$" to denote the set of assignments to the variables X.

A *winning move* for X in a QBF $QX.\Phi$ is an assignment $\tau \in moves(X)$ such that $\Phi|_\tau$ is true (if Q is $\exists$) or $\Phi|_\tau$ is false (if Q is $\forall$).

The function $\texttt{SAT}(\phi)$ represents a call to a SAT solver on a propositional formula ϕ. The function returns a satisfying assignment for ϕ, if such exists, and returns NULL otherwise.

A formula is in *strictly alternating* prenex form iff no two adjacent quantifier blocks have the same quantifier type (existential or universal). In this chapter, we assume that the input formula is in strictly alternating prenex form.

3.3 Recursive CEGAR-based Algorithm

In previous work, a CEGAR approach was used to solve quantified boolean formulas with 2 levels of quantifiers [20]. Here we present a generalization that applies to formulas with any number of quantifier alternations.

The basic idea is as follows. Consider a QBF instance $\exists X. \forall Y. \Phi$. If Y has only a few variables, we can fully expand $\forall Y. \Phi$ by taking the conjunction over all assignments:

$$\forall Y. \Phi \quad \Leftrightarrow \quad \bigwedge_{\mu \in moves(Y)} \Big(\Phi|\mu\Big) \quad = \quad (\Phi|\mu_1) \wedge ... \wedge (\Phi|\mu_n)$$

where $\{\mu_1, ..., \mu_n\} = moves(Y)$. But what if there are many variables in Y? It turns out that, in many instances that arise in practice, only a small number of assignments (moves) need to be considered. Accordingly, we use a *partial expansion* defined as follows:

Definition 5 (Partial Expansion). Let ω be a subset of $moves(Y)$.

The *partial expansion* of $\exists Y.\ \Phi$ over ω is the formula $\bigvee_{\mu \in \omega} \Phi|\mu$.

The *partial expansion* of $\forall Y.\ \Phi$ over ω is the formula $\bigwedge_{\mu \in \omega} \Phi|\mu$.

The partial expansion of $QY.\Phi$ is considered an *abstraction* of $QY.\Phi$. It represents a handicap on player Q in the sense that player Q is allowed to play only those moves in ω rather than any move in $moves(Y)$. Thus, if Q wins a partial expansion of $QY.\Phi$, then Q also wins $QY.\Phi$:

- $\big(\big(\bigwedge_{\mu \in \omega} \Phi|\mu\big) \Leftrightarrow \mathsf{false}\big) \;\Rightarrow\; \big(\big(\bigwedge_{\mu \in moves(Y)} \Phi|\mu\big) \Leftrightarrow \mathsf{false}\big)$
- $\big(\big(\bigvee_{\mu \in \omega} \Phi|\mu\big) \Leftrightarrow \mathsf{true}\big) \;\Rightarrow\; \big(\big(\bigvee_{\mu \in moves(Y)} \Phi|\mu\big) \Leftrightarrow \mathsf{true}\big)$

Algorithm 1: Basic recursive CEGAR algorithm for QBF

Function `Solve` $(QX.\, \bar{Q}Y.\, \Phi)$
/* Return value: A winning assignment for X if there is one, NULL otherwise.
begin
  if $(Y = \varnothing)$ **then return** $\big(Q{=}\exists\ ?\ \mathtt{SAT}(\phi)\ :\ \mathtt{SAT}(\neg\phi)\big)$
  $\omega := \varnothing$
  while true do
    $\alpha := \begin{cases} \exists X.\, \bigwedge_{\mu\in\omega} \Phi|_\mu & \text{if } \bar{Q} = \forall \\ \forall X.\, \bigvee_{\mu\in\omega} \Phi|_\mu & \text{if } \bar{Q} = \exists \end{cases}$
    cand := `Solve`(`Prenex`(α)) // find a candidate solution
    if *cand* = NULL **then return** NULL
    Remove from *cand* any variables not in X
    cex := `Solve`($\bar{Q}Y.\, \Phi|_{cand}$) // find a counterexample
    if *cex* = NULL **then return** *cand*
    $\omega := \omega \cup \{cex\}$
  end
end

To solve $QX.\, \bar{Q}Y.\, \Phi$, we start with a coarse partial assignment and gradually refine it until we find an answer. At a high level, the algorithm is as follows:

1. Initialize ω such that $\omega \subseteq moves(Y)$. (Specifically, we use $\omega = \varnothing$.)
2. Let α be the partial expansion of $\bar{Q}Y.\, \Phi$ over ω.
3. Try to find $cand \in moves(X)$ such that Player Q wins $\alpha|_{cand}$.
4. If no such assignment, we're done: Player $\bar{Q}$ wins $QX.\bar{Q}Y.\, \Phi$.
5. Try to find $cex \in moves(Y)$ such that Player $\bar{Q}$ wins $(\Phi|_{cand})|_{cex}$.
6. If no such assignment, we're done: Player Q wins $QX.\bar{Q}Y.\, \Phi$.
7. Let $\omega := \omega \cup \{cex\}$ and go back to Step 2.

The details of this algorithm are fleshed out in Algorithm 1.

3.3.1 Improving Recursive CEGAR-based Algorithm

Note that Algorithm 1 requires prenexing α. This is harmful because it loses information about dependencies among variables. Algorithm 2 avoids this prenexing by using the concept of a *multi-game*:

Definition 6 (multi-game). A *multi-game* is denoted by $QX.\{\Phi_1, \ldots, \Phi_n\}$ where each Φ_i is a prenex QBF starting with $\bar{Q}$ or has no quantifiers. The free variables of each Φ_i must be in X and all Φ_i have the same number of quantifier blocks. We refer

Algorithm 2: RAReQS: Recursive Abstraction Refinement QBF Solver

Function RAReQS ($QX.\,\{\Phi_1,\ldots,\Phi_n\}$)
/* Return value: A winning assignment for X if there is one, NULL otherwise.
begin
  if (Φ_i have no quantifiers) **then return** $Q{=}\exists$? $\mathtt{SAT}(\bigwedge_i \Phi_i)$: $\mathtt{SAT}(\neg(\bigvee_i \Phi))$
  $\alpha := QX.\,\{\}$
  while true **do**
    cand := RAReQS(α) // find a candidate solution
    if *cand* = NULL **then return** NULL
    Remove from *cand* any variables not in X.
    for $i := 1$ **to** n **do** $cex_i :=$ RAReQS($\Phi_i|_{cand}$) // find a counterexample
    if cex_i = NULL for all $i \in \{1..n\}$ **then return** *cand*
    let $l \in \{1..n\}$ be such that $cex_l \neq$ NULL
    $\alpha := \mathtt{Refine}(\alpha,\, \Phi_l, cex_l)$
  end
end

Refine is defined as follows:

$\mathtt{Refine}\big(QX.\{\Psi_1,\ldots,\Psi_n\},\ \bar{Q}YQX_1.\,\Psi,\ \mu\big) \ = \ QXX_1'.\{\Psi_1,\ldots,\Psi_n,\Psi'|\mu\}$
where X_1' are fresh duplicates of X_1, and Ψ' is Ψ with X_1 replaced by X_1'

$\mathtt{Refine}\big(QX.\{\Psi_1,\ldots,\Psi_n\},\ \bar{Q}Y.\,\psi,\ \mu\big) \ = \ QX.\{\Psi_1,\ldots,\Psi_n,\psi|\mu\}$
where ψ is a propositional formula (where no duplicates are needed)

to the formulas Φ_i as *subgames* and QX as the *top-level prefix*. A *winning move* for a multi-game is an assignment to the variables X such that it is a winning move for each of the formulas $QX.\ \Phi_i$.

3.4 CEGAR as a learning technique in DPLL

The previous section shows that CEGAR can give rise to a complete and sound algorithm for QBF. In this section we show that CEGAR enables us to extend existing DPLL solvers with an additional learning technique. To illustrate the basic idea consider the QBF $\forall X.\,(\exists Y.\ \phi)$ and a situation when the solver assigned values to variables in X and Y such that ϕ is satisfied, i.e., the existential player won. This assignment has two disjoint parts, π_{cand} and π_{cex}, which are assignments to X and Y, respectively. Conceptually, π_{cand} corresponds the candidate assignment in RAReQS and π_{cex} to its counterexample. In this case, the CEGAR-based learning will correspond to disjoining the formula $\phi|\pi_{\text{cex}}$ onto ϕ, resulting in $\forall X.\,(\exists Y.\ \phi) \vee \phi|\pi_{\text{cex}}$, so that π_{cand} is avoided in the future.

Algorithm 3: DPLL Algorithm with CEGAR Learning

```
global π_cur = ∅;
function dpll_solve(Φ_in) {
  while (true) {
     while (we don't know who has a winning strategy under π_cur) {
       decide_lit(); propagate();
     }
     Φ_in := dpll_learn(Φ_in);
     if (we learned who has a winning strategy under ∅) return;
     if (last decision literal is owned by winner) {
       Φ_in := cegar_learn(Φ_in);
     }
     backtrack();
     propagate(); // Learned information will force a literal.
  }
}
```

The CEGAR learning in DPLL is most naturally described in the context of a non-prenex, non-clausal solver such as GhostQ [24]. Given an assignment π, such a solver will tell us that either (1) the existential player wins under π, (2) the universal player wins under π, or (3) it is not yet known which player wins under π.

We modify such a solver by inserting a call to a new CEGAR-learning procedure after performing standard DPLL learning, as shown in Algorithm 3. We write "Φ_{in}" to denote the current input formula, i.e., the input formula enhanced with what the solver has learned up to now. Both standard DPLL learning and CEGAR learning are performed by modifying Φ_{in}. As shown in Algorithm 3, CEGAR learning is performed only if the last decision literal is owned by the winner. (The case where the last decision literal is owned by the losing player corresponds to the conflicts that take place *within* the underlying SAT solver in RAReQS.) The details of the DPLL CEGAR-learning procedure are provided in [19].

3.5 Experimental Results

A prototype of the CEGAR algorithm is implemented in a solver called RAReQS (Recursive Abstraction Refinement QBF Solver). For the underlying SAT solver, `minisat 2.2` [10] is used. We compared RAReQS to other solvers on the the *formal verification* and *planning* suites of QBF-LIB [32]. Several large and hard families were sampled with 150 files (`terminator`, `tipfixpoint`, `Strategic Companies`). The

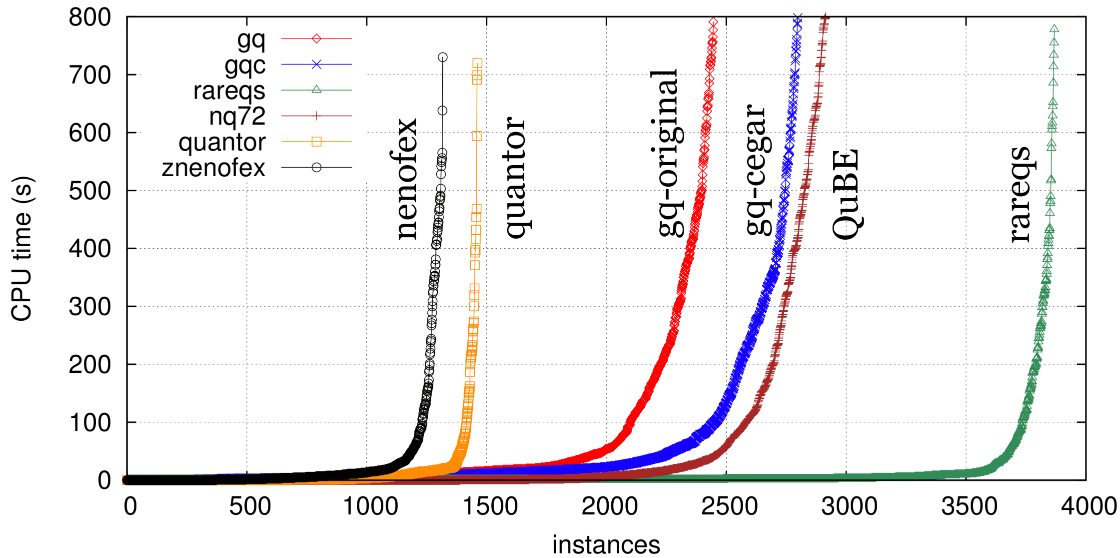

Figure 3.1: Cactus plot of the overall results

solvers QuBE7.2 [14], Quantor, and Nenofex were chosen for comparison. QuBE7.2 is a state-of-the-art DPLL-based solver; Quantor and Nenofex are expansion-based solvers. The experimental results were obtained on an Intel Xeon 5160 3GHz. The time limit was set to 800 seconds and the memory limit to 2GB.

All the instances were preprocessed by the preprocessor bloqqer [5] and instances solved by the preprocessor alone were excluded from further analysis. An exception was made for the family `Debug` where preprocessing turned out to be infeasible and the family was considered in its unpreprocessed form.

Unlike the other solvers, GhostQ's input format is not clause-based (QDIMACS) but it is circuit-based. To enable running GhostQ on the targeted instances, the solver was prepended with a reverse-engineering front-end. Since this front-end cannot handle bloqqer's output, GhostQ was run directly on the instances without preprocessing. The other solvers were run on the preprocessed instances (further preprocessing was disabled for QuBE7.2).

The relation between solving times and instances is presented by a cactus plot in Figure 3.1; number of solved instances per family are shown in Table 3.2; a comparison of RAReQS with other solvers is presented in Table 3.1.

On the considered benchmarks, RAReQS solved the most instances, approximately 33% more than the second solver QuBE7.2. RAReQS also turned out to be the best solver for most of the types of the considered instances. Table 3.1 further shows that for each of the other solvers, there is only a small portion of instances

	GhostQ	GhostQ-CEGAR	QuBE7.2	Quantor	Nenofex
Only RAReQS	1661	1336	998	2436	2564
Only competitor	242	269	46	30	13

Table 3.1: Number of instances solved by RAReQS but not by a competing solver, and *vice versa*

that the other solver can solve and RAReQS cannot.

In several families the addition of CEGAR learning to GhostQ worsened its performance. With the exception of `Robots2D`, however, the performance was worse only slightly. Overall, GhostQ benefited from the additional CEGAR learning and in particular for certain families. A family worth noting is `irqlkeapclte`, where no instances were solved by any of the solvers except for GhostQ-CEGAR.

3.6 Conclusion

This chapter has presented two novel techniques for solving QBF problems. First, a CEGAR-driven solver RAReQS has been presented which builds an abstraction of the given formula by constructing partial expansions. This solver has been experimentally shown to work very well on a wide variety of bencharmks. Second, CEGAR has been incorporated into DPLL solvers as an additional learning technique. While this technique does not take advantage of the full range of CEGAR learning exploited by RAReQS, it still provides a more powerful learning technique than standard clause/cube learning, and experimentally it has been shown helpful for a variety of benchmarks.

Family	Lev.	RAReQS	GhostQ	GhostQ-Cg	QuBE7.2	Quantor	Nenofex
trafficlight-ctlr (1459)	1–287	**1459**	806	1001	1092	955	863
RobotsD2 (700)	2–2	**699**	350	271	630	0	30
incrementer-encoder (484)	3–119	**483**	285	477	284	51	27
blackbox-01X-QBF (320)	2–21	**320**	138	126	224	3	4
Strat. Comp. (samp.) (150)	1–2	**107**	12	12	**107**	18	12
BMC (85)	1–3	**73**	26	48	37	65	64
Sorting-networks (84)	1–3	**72**	24	32	45	38	38
blackbox-design (27)	5–9	**27**	**27**	**27**	18	0	0
conformant-planning (23)	1–3	**17**	7	16	5	13	12
Adder (28)	3–7	**11**	2	2	4	5	9
Lin. Bitvec. Rank. Fun. (60)	3–3	**9**	0	0	0	0	0
Ling (8)	1–3	**8**	6	**8**	**8**	**8**	**8**
Blocks (7)	3–3	**7**	6	**7**	5	**7**	**7**
fpu (6)	1–3	**6**	0	0	**6**	**6**	**6**
RankingFunctions (4)	2–2	**3**	0	0	**3**	0	0
Logn (2)	3–3	**2**	**2**	**2**	**2**	**2**	**2**
Mneimneh-Sakallah (163)	1–3	110	**148**	141	89	3	22
tipfixpoint-sample (150)	1–3	26	**128**	127	22	5	6
terminator-sample (150)	2–2	98	**109**	103	9	25	0
tipdiam (121)	1–3	55	**99**	93	54	21	14
Scholl-Becker (55)	1–29	37	**43**	40	29	32	27
evader-pursuer (15)	5–19	10	**11**	8	**11**	2	2
uclid (3)	4–6	0	**2**	**2**	0	0	0
toilet-all (136)	1–1	134	133	131	131	**135**	133
Counter (58)	1–125	30	14	11	20	**33**	15
Debug (38)	3–5	3	0	0	0	**24**	6
circuits (63)	1–3	8	4	5	5	**9**	8
Gent-Rowley (205)	7–81	52	67	67	**70**	2	0
jmc-quant (+squaring) (20)	3–9	2	0	0	**6**	0	2
irqlkeapclte (45)	2–2	0	0	**44**	0	0	0
total (4669)		**3868**	2449	2801	2916	1462	1317

Table 3.2: Number of instances solved within 800 seconds by each solver. "Lev" indicates the number of quantifier blocks (min–max) in the family, post-bloqqer.

Chapter 4

Inference and Verification of Program Invariants

4.1 Introduction

A major obstacle to the adoption of verification tools is the manual burden involved in both writing formal specifications and helping the analyzer prove them. For software that uses dynamically-allocated heap memory, fully automatic "push-button" static analyzers typically give many false alarms, and they are often unsound (i.e., they may fail to detect that a bad state is reachable). A more precise analysis can be obtained using techniques such as Separation Logic [33] or TVLA [35]. These techniques are very powerful and can used to verify complex programs. However, they are not completely automatic; they require the user to supply invariants and/or other annotations, which is often tedious and requires knowledge of formal methods that many programmers do not have. Automatic inference of invariants is therefore highly desirable.

We present an approach for automatically inferring universally quantified properties about heap data structures. Our approach uses the Abstract Interpretation [9] framework. Our analyzer annotates each statement of the program with a *precondition* and a *postcondition*. Throughout the course of the analysis, the pre-/post-condition annotations of a statement may be updated. When the analysis is finished, it is guaranteed that the annotated pre-/post-conditions of each statement are always satisfied (on statement entry/exit, respectively) in every possible execution trace of the program.

The annotations may be used for several purposes. First, they can be used to find likely bugs or to verify that the program is free of certain types of run-time errors, such as failed assertions. Given an assertion `assert(`e`)` with an annotated precondition Φ, we verify the implication $\Phi \Rightarrow e$ or raise an alarm if the implication cannot be verified to hold true. A second use of annotated preconditions, in the case of dynamic languages such as Python/JavaScript/Ruby and object-oriented languages such as Java, is to generate more optimized bytecode by, e.g., resolving the types of objects statically rather than dynamically. Third, the annotations be used to automatically generate documentation that may be useful to programmers.

4.2 Related Work

Recent work in loop-invariant inference [23, 25, 29, 38] has dealt with quantified loops invariants for arrays and linked lists. In contrast, our proposed technique can handle more complex data structures, such as binary-search trees. The template-based invariant generation technique in [38] requires the user to provide a template (a quantified formula with holes where each hole can be substituted only with a conjunction of atomic propositions). To express an invariant with disjunctions, a user has to explicitly specify the disjunctions in template. In contrast, our proposed technique is completely automatic; it doesn't require any user annotations.

Jensen et al. [21] have developed a type analysis for JavaScript that is able to automatically infer the types of variables and detect errors such as accessing a non-existent field of an object. Their approach is very fast, but it cannot infer relational properties such as "if $\texttt{n} \neq 0$ then $\texttt{p} \neq \texttt{null}$", so it can produce false alarms if the program correctness depends on such an invariant. Our technique is slower, but it is able to infer relational properties.

Separation Logic [33] provides a framework for reasoning about heap data structures. It provides a *separating conjunction* operator "$*$" with the following semantics: $P * Q$ holds true in a state if the heap can be split into two distinct parts h_1 and h_2 such that P holds true in h_1 and Q holds true in h_2. Separation Logic has had much success in proving programs correct manually and with semi-automated tools, but it appears difficult to use in a fully-automated tool.

TVLA [6, 26, 35] is a parametric static analyzer that can verify inductive invariants of heap data structures. For each program to be analyzed, the user must specify the relevant *instrumentation predicates* in first-order logic with transitive closure. TVLA can verify very complex programs such as a Deutsch-Schorr-Waite garbage collector [27]. In contrast to TVLA, our technique infers useful heap invariants completely automatically, but it can fail for very complex programs. [**TO DO**:

Elaborate]

4.3 Target Language

We consider a tiny dynamically-typed programming language with dictionaries (key-value mappings) as the sole type of recursive data structure. Our target language is designed to serve as an intermediate language for analysis; programs written in languages such as Python or Java can be translated down to it. Dictionaries can be used straight-forwardly to emulate other data structures such as `structs`/`classes` in languages such as C++ and Java (using the field names or their numeric offsets as keys to the dictionary) and arrays. In addition, our target language allows every dictionary to be tagged with a *class name* when it is allocated.

Figure 4.1 shows the domain of the concrete semantics. A *value* can be an integer, a string, the special value `nil`, or a *dict-addr* (the address of a dictionary in memory). We assume an arbitrary but fixed total ordering on values, and we write $v_1 < v_2$ to denote that value v_1 is prior to value v_2 in this total ordering. The *state* of a program is described by a triple consisting of: (1) the environment *env*, which maps each program variable to a value, (2) the heap, which maps each key of each dictionary to a value, and (3) a mapping of each *dict-addr* to its associated *class name*.

The syntax for our target language is given in Figure 4.2. We write $d\texttt{[}k\texttt{]}$ to denote the value that key k is mapped to in dictionary d. To represent that a key is absent from a dictionary, it is mapped to the special value `nil`. To simplify later analysis, we require that each dictionary write to $d\texttt{[}k\texttt{]}$ be immediately preceded by a dictionary read from $d\texttt{[}k\texttt{]}$. The builtin functions in the target language are as follows:

$$
\begin{array}{lcl}
\mathit{value} & ::= & \mathit{integer\text{-}const} \;\mid\; \mathit{string\text{-}const} \;\mid\; \texttt{nil} \;\mid\; \mathit{dict\text{-}addr} \\
\mathit{env} & ::= & \mathit{prog\text{-}var} \;\rightarrow\; \mathit{value} \\
\mathit{heap} & ::= & \mathit{dict\text{-}addr} \times \underbrace{\mathit{value}}_{\text{key}} \;\rightarrow\; \mathit{value} \\
\mathit{classes} & ::= & \mathit{dict\text{-}addr} \;\rightarrow\; \mathit{string\text{-}const} \\
\mathit{state} & ::= & \mathit{env} \times \mathit{heap} \times \mathit{classes}
\end{array}
$$

Figure 4.1: Concrete Semantic Domain

prog-var	::=	Variable in the program
$expr_{obj}$	::=	*prog-var* | *integer-const* | *string-const* | `nil`
$expr_{bool}$	::=	$(expr_{obj} = expr_{obj})$ | $(expr_{obj} \leq expr_{obj})$
stmt	::=	*simple-stmt* | *compound-stmt*
simple-stmt	::=	*prog-var* := $expr_{obj}$
	|	*prog-var* := *prog-var* [$expr_{obj}$] /* Dict Read */
	|	*prog-var* [$expr_{obj}$] := *prog-var* /* Dict Write */
	|	*prog-var* := *builtin-func*(...)
	|	`assume`($expr_{bool}$)
	|	`assert`($expr_{bool}$)
compound-stmt	::=	*stmt* ; *stmt*
	|	`if` $expr_{bool}$ `then` *stmt* `else` *stmt*
	|	`while` $expr_{bool}$ `do` *stmt*

Figure 4.2: Syntax of Target Language

- `first_key`(d): Returns the least key in dict d, or `nil` if d is empty. ("Least" is determined by the above-mentioned total ordering on values.)
- `next_key`(d, *prev*): Returns the least key k in d such that $prev < k$, or `nil` if no such k exists.
- `new_dict`(*class*): Allocates a dict; *class* must be a string constant and must not be "`int`", "`str`", "`nil`", or "`unalloc`".
- `free`(d): Deallocates dictionary d.
- `read_int()` and `read_str()`: Source of non-determinism.

At the start of the program, all variables are initialized to `nil`.

4.4 Overview of Analysis

In our analysis, the annotated pre-/post-conditions are formulas in a fragment of first-order logic augmented with transitive closure. Such a formula can be viewed as representing the set of programs states that satisfy the formula. For example, the formula $a = 42$ represents the set of all states in which the value of variable a is 42.

In a formula, we write "`lookup`(d, k)" to denote the value associated with key k in dictionary d. We may abbreviate "`lookup`(d, k)" by "d`[`k`]`". For example, the `function Analyze(`c`) {`

  if c is the top-level statement in the program:

    precond(c) := $\bigwedge_{v \in \textit{prog-var}} v = \texttt{nil}$

  if c has the form "c_1`;`c_2":

    precond(c_1) := precond(c)
    `Analyze`(c_1)
    precond(c_2) := postcond(c_1)
    `Analyze`(c_2)
    postcond(c) := postcond(c_2)

  if c has the form "`if` e `then` c_1 `else` c_2":

    precond(c_1) := precond(c) $\wedge\, e$
    `Analyze`(c_1)
    precond(c_2) := precond(c) $\wedge \neg e$
    `Analyze`(c_2)
    postcond(c) := α(postcond(c_1) $\vee$ postcond(c_2))

  if c has the form "`while` e `do` c_1":

    Repeat until fixed point:

      precond(c_1) := precond(c) $\wedge\, e$
      `Analyze`(c_1)
      precond(c) := α(precond(c) $\vee$ postcond(c_1))
      postcond(c) := precond(c) $\wedge \neg e$

  if c has the form of a *simple-stmt*:

    postcond(c) := $\alpha(\llbracket c \rrbracket(\text{precond}(c)))$

`}`

Figure 4.3: Top-Level Algorithm

formula $\forall u_1.\ d_1\texttt{[}u_1\texttt{]} = d_2\texttt{[}u_1\texttt{]}$ represents the set of all states in which d_1 has the same key-value mapping as d_2. We write "$\texttt{class}(x)$" to denote the class of x. If x is an integer, string, or nil, then $\texttt{class}(x)$ is "`int`", "`str`", or "`nil`", respectively.

Figure 4.3 shows the top-level algorithm for our analysis; the following notation is used:

- precond(c) and postcond(c) denote the annotated precondition and postcondition of statement c.
- $[\![c]\!](\Phi)$ denotes the strongest postcondition of statement c for a given precondition Φ, provided that c that has the form of a *simple-stmt* (as defined in Figure 4.2). A full definition of $[\![c]\!]$ is found in Figure 4.4.
- α is an *abstraction function.* Given a formula Φ, $\alpha(\Phi)$ is an overapproximation of Φ (i.e., the set of states represented by $\alpha(\Phi)$ is a superset of the set of states represented by Φ). We say that $\alpha(\Phi)$ is a *sound* approximation of Φ because if Φ is satisfied by an error state σ then $\alpha(\Phi)$ is also satisified by the same error state σ. The range of the function α is a finite set of formulas that we will denote $\widehat{\textit{st-fmla}}$. (This finiteness, together with certain other conditions, ensures termination of the `Analyze` algorithm.)

4.5 Abstraction Function

There are two parameters in our abstraction method, q and m, described below.

We define $\widehat{\textit{st-fmla}}$ to be the set of formulas that comply with the grammar for $\widehat{\textit{st-fmla}}$ (defined in Figure 4.5) and meet the following conditions:

1. The formula is in *prenex form* with a prefix of q universal quantifiers: $\forall u_1 ... \forall u_q.\, \phi$ where ϕ is quantifier-free.
2. The syntactic nesting depth of `lookup` is limited to a maximum depth m (e.g., if $m = 2$, the term $d\texttt{[}k_1\texttt{][}k_2\texttt{]}$ is valid but $d\texttt{[}k_1\texttt{][}k_2\texttt{][}k_3\texttt{]}$ is not, where d, k_1, k_2, and k_3 are program variables).
3. The only constants allowed are those that occur in the program text.
4. Restrictions on the $\texttt{reach}^+$ predicate described in Section 4.6.1.

In Figure 4.4, we give a formal semantics of the non-compound statements in our target language. In defining the semantics, we need to refer to the values that the program variables had before executing the statement and their values after. To refer to the prior values, we write a subscript "pre". For example, given the precondition

Notation: We write $t[x \to e]$ to denote the result of substituting e for x in t.
In the below rules, any variable subscripted "pre" is to be understood as a fresh variable that is implicitly existentially quantified at the outermost scope.

1. $[\![v := e]\!](\Phi) \;=\; \big(\Phi[v \to v_{\text{pre}}] \;\wedge\; v = e[v \to v_{\text{pre}}]\big)$
   where e is a program variable or a constant expression. Note that the right-hand side comes from the Floyd assignment axiom [12].

   Example: $[\![a := 42]\!](a = 5 \;\wedge\; b = a) \;=\; \exists a_{\text{pre}}.\,(a_{\text{pre}} = 5 \;\wedge\; b = a_{\text{pre}} \;\wedge\; a = 42)$

2. $[\![\texttt{assume(}e\texttt{)}]\!](\Phi) \;=\; \Phi \wedge e$

3. $[\![\texttt{verify(}e\texttt{)}]\!](\Phi) \;=\; \Phi \wedge e$
   Our static analyzer raises an alarm if it cannot ascertain that $\Phi \Rightarrow e$.

4. $[\![d := \texttt{new\_dict(c)}]\!](\Phi) \;=\; \big(\Phi[\texttt{class} \to \texttt{class}_{\text{pre}}, d \to d_{\text{pre}}] \;\wedge$
   $\texttt{class}_{\text{pre}}(d) = \texttt{"unalloc"} \;\wedge\; \texttt{class}(d) = c \;\wedge$
   $\bigwedge_{t \in \mathit{terms}(\Phi[d \to d_{\text{pre}}])} (t \neq d \;\Rightarrow\; \texttt{class}(t) = \texttt{class}_{\text{pre}}(t)) \;\wedge\; \forall u_1.\; d\texttt{[}u_1\texttt{]} = \texttt{nil}\big)$

5. $[\![d := \texttt{free(d)}]\!](\Phi) \;=\; \big(\Phi[\texttt{class} \to \texttt{class}_{\text{pre}}] \;\wedge\; \texttt{class}(d) = \texttt{"unalloc"} \;\wedge$
   $\bigwedge_{t \in \mathit{terms}(\Phi[d \to d_{\text{pre}}])} (t \neq d \;\Rightarrow\; \texttt{class}(t) = \texttt{class}_{\text{pre}}(t))\big)$

6. $[\![v := d\texttt{[}k\texttt{]}]\!](\Phi) \;=\; \big(\Phi[v \to v_{\text{pre}}] \;\wedge\; v = (d\texttt{[}k\texttt{]})[v \to v_{\text{pre}}]\big)$

7. $[\![d\texttt{[}k\texttt{]} := e]\!](\Phi) \;=\; \big(\Phi[\texttt{lookup} \to \texttt{lookup}_{\text{pre}}] \;\wedge\; (d\texttt{[}k\texttt{]} = e) \;\wedge$
   $\bigwedge_{d'\texttt{[}k'\texttt{]}_{\text{pre}} \in \mathit{terms}(\Phi[\texttt{lookup} \to \texttt{lookup}_{\text{pre}}])} (d' \neq d \vee k' \neq k) \Rightarrow (d'\texttt{[}k'\texttt{]} = d'\texttt{[}k'\texttt{]}_{\text{pre}})\big)$
   where $d'\texttt{[}k'\texttt{]}_{\text{pre}}$ abbreviates $\texttt{lookup}_{\text{pre}}(d', k')$.

8. $[\![v := \texttt{first\_key}(d)]\!](\Phi) \;=\; \big(\Phi[v \to v_{\text{pre}}] \wedge$
   $\big((v = \mathsf{nil} \;\wedge\; \forall u_1.\; d'\texttt{[}u_1\texttt{]} = \mathsf{nil}) \;\vee$
   $(v \neq \mathsf{nil} \;\wedge\; d'\texttt{[}v\texttt{]} \neq \mathsf{nil} \;\wedge\; \forall u_1.\; d'\texttt{[}u_1\texttt{]} \neq \mathsf{nil} \;\Rightarrow\; v \leq u_1)\big)\big)$
   where d' denotes $d[v \to v_{\text{pre}}]$.

9. $[\![v := \texttt{next\_key}(d, p)]\!](\Phi) \;=\; \big(\Phi[v \to v_{\text{pre}}] \wedge$
   $\big((v = \mathsf{nil} \;\wedge\; \forall u_1.\; p' < u_1 \;\Rightarrow\; d'\texttt{[}u_1\texttt{]} = \mathsf{nil}) \;\vee$
   $(v \neq \mathsf{nil} \;\wedge\; d'\texttt{[}v\texttt{]} \neq \mathsf{nil} \;\wedge\; \forall u_1.\; (p' < u_1 \;\wedge\; d'\texttt{[}u_1\texttt{]} \neq \mathsf{nil}) \;\Rightarrow\; v \leq u_1)\big)\big)$
   where d' denotes $d[v \to v_{\text{pre}}]$ and p' denotes $p[v \to v_{\text{pre}}]$.

Figure 4.4: Semantics for Non-Compound Statements

$$
\begin{array}{lll}
\mathit{term} & ::= & \mathit{integer\text{-}const} \;\mid\; \mathit{string\text{-}const} \;\mid\; \texttt{nil} \\
& \mid & \mathit{prog\text{-}var} \;\mid\; u_i \\
& \mid & \texttt{lookup}(\underbrace{\mathit{term}}_{\text{dict}}, \underbrace{\mathit{term}}_{\text{key}}) \\
\mathit{ap} & ::= & \mathit{term} = \mathit{term} \\
& \mid & \mathit{term} \le \mathit{term} \\
& \mid & \texttt{class}(\mathit{term}) = \mathit{string\text{-}const} \\
& \mid & \texttt{reach}^{+}(\underbrace{\mathit{term}}_{\text{dest}}, \underbrace{\mathit{term}}_{\text{origin}}, \underbrace{\text{set of } \mathit{terms}}_{\text{stop-points}}, \underbrace{\text{set of } \mathit{string\text{-}const} \text{ pairs}}_{\text{edges}}) \\
\widehat{\mathit{st\text{-}fmla}} & ::= & \texttt{true} \mid \texttt{false} \mid \mathit{ap} \mid \neg\widehat{\mathit{st\text{-}fmla}} \\
& \mid & \widehat{\mathit{st\text{-}fmla}} \wedge \widehat{\mathit{st\text{-}fmla}} \;\mid\; \widehat{\mathit{st\text{-}fmla}} \vee \widehat{\mathit{st\text{-}fmla}} \\
& \mid & \forall u_i \,.\, \widehat{\mathit{st\text{-}fmla}}
\end{array}
$$

Figure 4.5: Atomic Propositions and State Formulas

$a{=}5 \wedge b{=}a$ and the assignment statement $a := 42$, we may compute a strongest postcondition $\exists a_{\text{pre}}.\ a_{\text{pre}} = 5 \ \wedge\ b = a_{\text{pre}} \ \wedge\ a = 42$. Note that the symbols subscripted with "pre" are not allowed in the abstract domain $\widehat{\mathit{st\text{-}fmla}}$ (because don't comply with the grammar in Figure 4.5). Accordingly, we define an abstraction function α that eliminates the existentially quantified "pre" symbols while preserving soundness. For the above example, the abstraction function might yield a postcondition such as $b = 5 \ \wedge\ a = 42$. The requirements on the abstraction function α are:

1. $\alpha(\Phi)$ must represent a superset of the states represented by Φ, in order to ensure soundness, and
2. $\alpha(\Phi)$ must be in $\widehat{\mathit{st\text{-}fmla}}$ (the *abstract domain*).

Additionally, we would like α to possess two additional properties:

1. $\alpha(\Phi)$ should be a reasonably precise overapproximation of Φ. (E.g., an abstraction function $\lambda\Phi.\texttt{true}$, which always yields `true` regardless of the input formula Φ, meets the two requirements, but it is terribly imprecise!)
2. α should produce reasonably small formulas, so that the analysis uses a reasonable amount of time and memory.

Note that in Figure 4.3 and in the definition of the semantics in Figure 4.4, α is only applied to formulas that are in (or can easily be re-written in) the form $\forall u_1 ... \forall u_q.\, \phi$ where ϕ has no quantifiers (but may contain terms with the "pre" subscript). So, we only need to define α on formulas of this form.

At a high level, we compute the abstraction $\alpha(\forall \vec{u}.\phi)$ as illustrated in Figure 4.6:

1. **Universal Instantiation.** We instantiate clauses implied by $\forall \vec{u}.\phi$ that contain

universally quantified variables. If such a clause C contains a term d[u_i] and a term d[k] occurs in ϕ, we instantiate clause C with k substituted for u_i.

2. **Reachability Predicate.** This is discussed in detail in Section 4.6.

3. **Rewriting.** We rewrite the formula to avoid unnecessary references to "pre" terms. The rewritten formula must be logically equivalent to original formula under the theory of equality, uninterpreted functions, and total order. E.g., we may rewrite $a_{\text{pre}}{=}5 \wedge b{=}a_{\text{pre}}$ as $a_{\text{pre}}{=}5 \wedge b{=}5$. The purpose of this step is to prepare for the next step, wherein we treat atomic propositions as independent boolean variables. Currently, we convert the formula to *disjunctive normal form (DNF)* and process each cube separately, but this does not scale up to even moderate-size programs. We are now investigating a method based on combining techniques employed by Binary Decision Diagrams (BDDs) [7] with the closed-QBF techniques discussed in Chapters 1 and 2 of this thesis proposal. We expect that this investigation will yield good experimental results.

4. **Existential Elimination.** In this step, we treat all atomic propositions as boolean variables. The atomic propositions that contain a subscripted "pre" are treated as existentially quantified boolean variables. All other atomic propositions are treated as free (unquantified) boolean variables. This yields a formula in QBF with free variables: The goal is find a formula that is logically equivalent but doesn't contain any existentially quantified boolean variables. This step can be handled by existing BDD tools. Alternatively, our investigation of a way of combining BDD techniques with closed-QBF techniques may produce an efficient algorithm for this step.

```
function α(Φ) {
    Φ := univ_instantiation(Φ);
    Φ := update_reach(Φ);
    Φ := rewrite_modulo_theory(Φ);
    Φ := existential_elim(Φ);
    return Φ;
}
```

Figure 4.6: Abstraction Function

4.6 Reachability Predicate

The reachability predicate $\texttt{reach}^{+}(\mathit{dest}, \mathit{orig}, \mathit{stop}, \mathit{edges})$ is used to summarize portions of the heap memory that are not represented concretely. The purpose of the third argument (*stop*) may not be immediately intuitive, so in order to motivate the need for it, let us first consider a simpler, coarser-grained predicate $\texttt{reach}_{\mathrm{c}}$. Given two terms *orig* and *dest*, we define $\texttt{reach}_{\mathrm{c}}(\mathit{dest}, \mathit{orig})$ to be true iff *dest* can be reached from *orig* via a series of dictionary lookups. Formally, $\texttt{reach}_{\mathrm{c}}$ is recursively defined by:

$$\texttt{reach}_{\mathrm{c}}(\mathit{dest}, \mathit{orig}) \;=\; (\mathit{orig} = \mathit{dest}) \vee \exists k.\, \texttt{reach}_{\mathrm{c}}(\mathit{dest}, \mathit{orig}\texttt{[}k\texttt{]})$$

Consider a program that creates a circularly-linked list of strings and then processes each list node, counting the number of occurrences of each word in the strings. Figure 4.7 shows the heap memory for such a program. Figure 4.7(a) illustrates the heap before processing any nodes, and Figure 4.7(b) illustrates the heap after

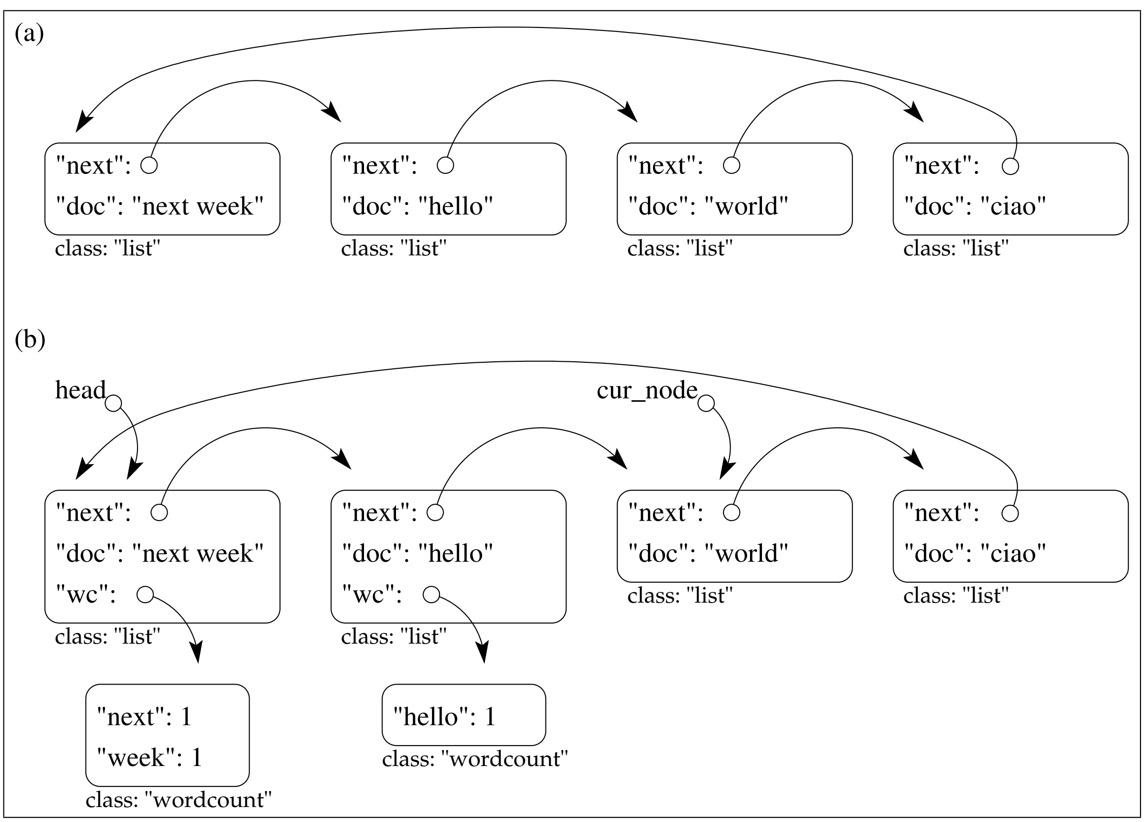

Figure 4.7: Heap memory for Word Count program: (a) before processing any nodes, and (b) after processing the first two nodes.

processing the first half of the list.

First let us consider the heap memory illustrated in Figure 4.7(a). We can express the following property using the $\texttt{reach}_c$ predicate: "For every object u_1 that is reachable from `head`, if u_1 has class `list`, then u_1`["doc"]` is a string and u_1`["next"]` is a `list` object". Formally:

$$\forall u_1.\,(\texttt{reach}_c(u_1, \texttt{head}) \wedge \texttt{class(}u_1\texttt{)}=\texttt{"list"}) \Rightarrow \Big((\texttt{class(}u_1\texttt{["doc"])} = \texttt{"str"}) \wedge (\texttt{class(}u_1\texttt{["next"])} = \texttt{"list"})\Big)$$

Now let us consider the heap memory illustrated in Figure 4.7(b). It consists of (1) a list segment in which each node has a `wc` field of class `wordcount`, followed by (2) a list segment in which each node lacks a `wc` field. We want to summarize the heap memory in such a way that retains this information. If we were to use the simple $\texttt{reach}_c$ predicate to describe what is reachable from the head of the list in Figure 4.7(b), we would lose precision because it fails to distinguish between the processed and unprocessed segments of the list.

We would like a predicate that identifies the set of nodes between the head of the list and the first unprocessed node. To do this, we include a set of *stop-points* as an argument to the $\texttt{reach}^+$ predicate. Informally, $\texttt{reach}^+(\mathit{dest}, \mathit{orig}, \mathit{stop}, \mathit{edges})$ is true iff *dest* can be reached from *orig* via *edges* without passing through any stop-points. The *edges* parameter is a set of $(\mathit{class}, \mathit{key})$ pairs. For example, $\mathit{edges} = \{(\texttt{"list"}, \texttt{"next"})\}$ corresponds to reachability by following the `next` field of `list` objects. A star ("`*`") in place of a class/key indicates that any class/key may be followed. Formally, we define two reachability predicates as follows:

$$\begin{aligned}
&\texttt{reach}^1(\mathit{dest}, \mathit{orig}, \mathit{stop}, \mathit{edges}) \;=\\
&\quad \exists c.\exists k.\;\; \mathit{orig}\texttt{[}k\texttt{]} = \mathit{dest} \;\wedge\; \mathit{dest} \notin \mathit{stop} \;\wedge\; \texttt{class}(\mathit{orig}) = c \;\wedge\\
&\quad ((c,\ k) \in \mathit{edges} \;\vee\; (c,\ \texttt{"*"}) \in \mathit{edges} \;\vee\; (\texttt{"*"}, \texttt{"*"}) \in \mathit{edges})
\end{aligned}$$

$$\begin{aligned}
&\texttt{reach}^+(\mathit{dest}, \mathit{orig}, \mathit{stop}, \mathit{edges}) \;=\; \texttt{reach}^1(\mathit{dest}, \mathit{orig}, \mathit{stop}, \mathit{edges}) \vee\\
&\quad \exists x.\ (\texttt{reach}^1(x, \mathit{orig}, \mathit{stop}, \mathit{edges}) \;\wedge\; \texttt{reach}^+(\mathit{dest}, x, \mathit{stop}, \mathit{edges}))
\end{aligned}$$

So, for example, we can use the $\texttt{reach}^+$ predicate to describe the following property of the heap memory state in Figure 4.7(b): "Every `list` object between `head` and `cur_node` has a field `wc` of class `wordcount`, and every `list` object between `cur_node` and `head` lacks a `wc` field (i.e., the `wc` key is mapped to `nil`)". Formally:

$$\begin{aligned}\forall u_1.\,\big(&(\texttt{reach}(u_1,\,\texttt{head},\,\mathit{stop},\,\mathit{edges}) \wedge \texttt{class}(u_1)\texttt{="list"}) \Rightarrow \\ &\quad \texttt{class}(u_1\texttt{["wc"]}) = \texttt{"wordcount"}\big) \wedge \\ \big(&(\texttt{reach}(u_1,\,\texttt{cur\_node},\,\mathit{stop},\,\mathit{edges}) \wedge \texttt{class}(u_1)\texttt{="list"}) \Rightarrow \\ &\quad u_1\texttt{["wc"]} = \texttt{nil}\big)\end{aligned}$$

where $\mathit{edges} = \{(\texttt{"list"}, \texttt{"next"})\}$ and $\mathit{stop} = \{\texttt{cur\_node}, \texttt{head}\}$.

4.6.1 Restrictions on Reachability Predicates in $\widehat{\textit{st-fmla}}$

An atomic proposition $\texttt{reach}^{+}(\mathit{dest}, \mathit{orig}, \mathit{stop}, \mathit{edges})$ may appear in a formula in $\widehat{\textit{st-fmla}}$ only if:

- *dest* is a universal variable $(u_1, ..., u_q)$.
- *orig* is a program variable.
- *stop* is the set of all program variables.

4.6.2 Updating the Reachability Predicate

In $\widehat{\textit{st-fmla}}$, the *stop* argument of the $\texttt{reach}^{+}$ predicate is required to be exactly the set of program variables. Let V denote this set. Note that the semantic definitions in Figure 4.4 can produce $\texttt{reach}^{+}$ predicates with different *stop* arguments. For example, given a precondition $\Phi \in \widehat{\textit{st-fmla}}$ and an assignment statement $v := e$, the corresponding postcondition $[\![v := e]\!](\Phi)$ would contain the subformula $\Phi[v \to v_{\text{pre}}]$. So if a $\texttt{reach}^{+}$ predicate appears in $\Phi[v \to v_{\text{pre}}]$, then its *stop* argument would include v_{pre} rather than v.

To recap, we have a formula Ψ that contains a $\texttt{reach}^{+}$ predicate whose *stop* argument is $V[v \to v_{\text{pre}}]$, but we want a logically equivalent formula that contains only $\texttt{reach}^{+}$ predicates whose *stop* argument is V. To accomplish this, we use Equation 4.1 (illustrated in Figure 4.8), which defines one $\texttt{reach}^{+}$ predicate in terms of another. In particular, we conjoin Ψ with two instantiations of Equation 4.1 below (one with s substituted with v, and one with s substituted with v_{pre}) and then existentially quantify out all undesired $\texttt{reach}^{+}$ predicates:

$$\begin{aligned}
&\texttt{reach}^{+}(\mathit{dest},\ \mathit{orig},\ \mathit{stop}_0,\ \mathit{edges}) = \\
&\qquad \texttt{reach}^{+}(\mathit{dest},\ \mathit{orig},\ \mathit{stop}_0 \cup \{s\},\ \mathit{edges}) \vee \\
&\qquad (\exists x.\ \texttt{reach}^{*}(x,\ \mathit{orig},\ \mathit{stop}_0 \cup \{s\},\ \mathit{edges}) \wedge \\
&\qquad\qquad \texttt{reach}^{1}(s,\ x,\ \mathit{stop}_0,\ \mathit{edges}) \wedge \\
&\qquad\qquad \texttt{reach}^{*}(\mathit{dest},\ s,\ \mathit{stop}_0 \cup \{s\},\ \mathit{edges}))
\end{aligned} \tag{4.1}$$

where stop_0 is $V \setminus \{v\}$ and $\texttt{reach}^{*}(\mathit{dest}, \mathit{orig}, \mathit{stop}, \mathit{edges})$ is defined as the formula $\texttt{reach}^{+}(\mathit{dest}, \mathit{orig}, \mathit{stop}, \mathit{edges}) \vee (\mathit{dest} = \mathit{orig})$. Equation 4.1 deserves some explanation. By elementary graph theory, if *dest* is reachable from *orig*, then it is reachable via a loop-free path. So without loss of generality, consider a loop-free path from *orig* to *dest*. Such a path has either 0 occurrences of s (corresponding to the first disjunct in the RHS of Equation 4.1) or exactly 1 occurrence (corresponding to the second disjunct). In the case where there is 1 occurrence, there exists a predecessor of s (let's call it "x") that is reachable from *orig* without passing through s, and *dest* is reachable from s, as illustrated in Figure 4.8.

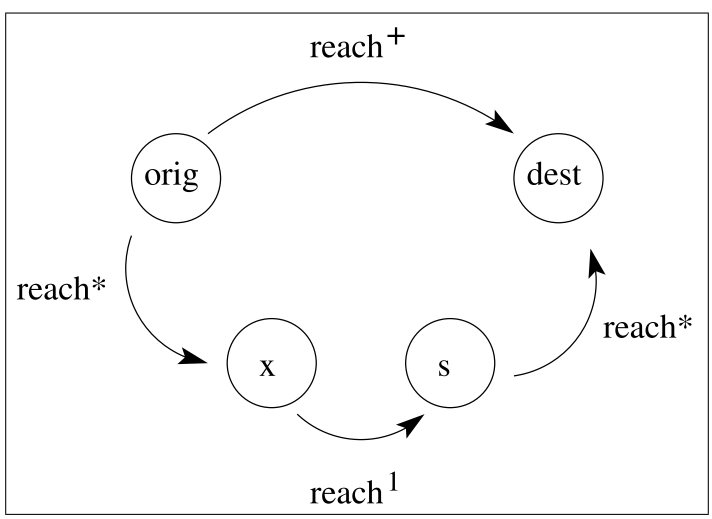

Figure 4.8: Illustration for Equation 4.1

4.7 Future Work

There is a working implementation of the analysis described in this chapter, but it is too slow and consumes too much memory for programs of any appreciable size. We are working on a more efficient way of computing the result of a suitable abstraction function, as discussed in Section 4.5. By combining recently-developed techniques for solving closed-QBF problems with techniques used for BDDs, we aim to implement an abstraction function capable of handling industrial-size programs.

Additionally, it is desired to augment the target language with syntactic constructs for defining functions and to extend the analyzer to efficiently handle such functions. The main work needed is a technique for *function summarization* such as that used in [22]. This would allow the analyzer to avoid needlessly re-analyzing the body of a function when it is called in contexts that differ only in immaterial aspects.

Chapter 5

Proposed Work and Timeline

To finish my thesis work, I will complete at least one of the following tasks:

1. **Invariant Inference for Heap Data Structures.** Design and implement a more efficient algorithm for the abstraction function α. The implementation should be able automatically to infer invariants and verify assertions for programs with heap data structures. The analyzer provides a general interface: the user can write specifications using `assert`s in the target language. (Specifications that are quantified over elements of a collection can be expressed by writing an `assert` inside a loop.) The analyzer will be tested on various benchmarks used in Separation Logic and other example programs that manipulate heap data structures. Some of the examples that we expect the analyzer to be able to handle include:
   (a) verifying the correctness of the `insert`, `find`, and `remove` operations for AVL binary search trees (without verifying balancedness),
   (b) verifying that reversing a singly-linked list twice yields the original list,
   (c) verifying that an adjacency-list representation of an undirected graph remains self-consistent — i.e., verifying, for all pairs of nodes (n_1, n_2), that n_1 is in the adjacency list of n_2 iff n_2 is in the adjacency list of n_1,
   (d) inferring the invariants maintained by an implementation of the two-watched-literals scheme for SAT solvers,
   (e) inferring the shape of nested data structures such as double-linked lists of DAGs of trees.
2. **Integrating CEGAR with DPLL techniques for QBF.** Design and implement a technique that tightly integrates CEGAR learning with DPLL tech-

niques. The resulting solver should significantly outperform both the existing GhostQ and RAReQS solvers on certain families of benchmarks, and it should perform at least as well (modulo a small overhead cost) as the existing version of RAReQS on the remaining benchmark families.

Timeline

In December through January or February, I will work on invariant inference for programs with heap data structures. Then I will discuss the progress of my work with the thesis committee. If there are good experimental results, then I will finish writing my thesis and defend as soon as the thesis committee believes is appropriate. If, on the other hand, the experimental results are poor, then I will instead focus on integrating CEGAR learning with DPLL techniques for the remaining part of my thesis work.

Improved Algorithm for Abstraction Function α

A principal component of the abstraction function requires solving a quantified boolean formula with *free variables* (i.e., variables not bound by quantifiers). The problem of QBF with free variables can, of course, be solved using BDDs. However, as part of my thesis work, I am interested in investigating whether it might be advantageous for a solver to combine BDD techniques with recently developed closed-QBF techniques. The reason that I believe it may be advantageous is as follows. For some formulas, the set of satisfying assignments can be expressed compactly as a BDD and can be found quickly with BDD tools. Yet for other formulas, the set of satisfying assignments can be expressed compactly in DNF or CNF and can be found quickly with SAT/QBF-based techniques. Accordingly, a technique that combines BDD techniques with SAT/QBF techniques may potentially perform better than either a pure BDD or a pure SAT/QBF approach.

Consider a QBF formula Φ_{in}. The existing version of my solver GhostQ only considers *closed* QBF instances (i.e., instances without free variables). It learns sequents of the form $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\Phi \Leftrightarrow \texttt{true})$ and $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\Phi \Leftrightarrow \texttt{false})$, where Φ is a subformula of Φ_{in}. To handle free variables, my proposed solver will allow sequents of the more general form $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\Phi \Leftrightarrow \psi)$ where Φ is a quantified boolean formula and ψ is a propositional formula represented as an *unordered* BDD.

The proposed solver will use the (Q)DPLL algorithm with several modifications.

Free variables are considered upstream of all quantified variables, so all free variables must be assigned before a quantified variable may be chosen as a decision variable. (However, quantified variables may be *forced* even if not all free variables have been assigned.) During learning, resolution is performed as usual if the resolvent is a quantified variable. If the resolvent r is a free variable, then we derive a new sequent from the two existing sequents as follows:

$$\frac{\begin{array}{l}\langle\{x_1, ..., x_n,\, r\},\, L_1^{\text{fut}}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi_1)\\ \langle\{y_1, ..., y_n,\, \neg r\},\, L_2^{\text{fut}}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi_2)\end{array}}{\langle\{x_1, ..., x_n,\, y_1, ..., y_n\},\, L_1^{\text{fut}} \cup L_2^{\text{fut}}\rangle \models (\Phi_{\text{in}} \Leftrightarrow (r \;?\; \psi_1 : \psi_2))}$$

where $(r \;?\; \psi_1 : \psi_2)$ denotes the unordered BDD with root node v such that $\text{var}(v) = r$, $\text{high}(v) = \psi_1$, and $\text{low}(v) = \psi_2$ (except if $\psi_1 = \psi_2$, in which case the BDD must be reduced). The above-derived sequent is used in Unit Propagation in a manner similar to sequents learned for a closed QBF. If the literals in $\{x_1, ..., x_n,\, y_1, ..., y_n\}$ are assigned true under the current assignment, then the sequent is conflicting. If all but one of the literals are true, then the sequent forces the remaining literal false.

A second sequent can also be derived:

$$\frac{\langle\{x_1, ..., x_n,\, y_1, ..., y_n\},\, L_1^{\text{fut}} \cup L_2^{\text{fut}}\rangle \models (\Phi_{\text{in}} \Leftrightarrow (r \;?\; \psi_1 : \psi_2))}{\langle\varnothing,\, L_1^{\text{fut}} \cup L_2^{\text{fut}}\rangle \models (\Phi_{\text{in}}|\{x_1, ..., x_n,\, y_1, ..., y_n\} \Leftrightarrow (r \;?\; \psi_1 : \psi_2))}$$

Sequents of this type are not useful in Unit Propagation; however, they can be utilized in manner similar to a BDD operation cache. In particular, whenever a new literal is added to the current assignment π_{cur}, the solver performs a lookup to see if it has already learned a sequent of the form $\langle\varnothing,\, L^{\text{fut}}\rangle \models (\Phi_{\mathsf{SEQ}} \Leftrightarrow \psi)$, where Φ_{SEQ} is equal to $\Phi_{\text{in}}|\pi_{\text{cur}}$ and no literal in L^{fut} is falsified by π_{cur}. If such a sequent is found, then the solver derives $\langle\pi_{\text{cur}},\, L^{\text{fut}}\rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$, which immediately becomes conflicting. Note that if the solver is made to always assign variables in a fixed order (which requires turning off Unit Propagation), then it builds an ordered BDD representation.

Integrating CEGAR with DPLL

A major advance in the field of SAT solvers was the introduction of *conflict analysis* and *non-chronological backtracking* [37]. Existing DPLL-based QBF solvers also incorporate non-chronological backtracking; a single backtrack can undo the assignments of variables in multiple quantifier blocks. However, the existing version of RAReQS can only backtrack a single quantifier block at a time. This leads to the algorithm being exponential in the number of quantifier alternations, even for simple problems that are trivial for DPLL-based solvers. For example, the QBF formula

$$\forall u_n\, \exists e_{n-1} \ldots \forall u_2\, \exists e_1.\, (e_1 \vee u_2) \wedge (\neg e_1 \vee \neg u_2)$$

could not be solved by RAReQS for $n = 40$ (without preprocessing[1], and given a timeout of 60 seconds), despite it being trivial for DPLL-based solvers. We plan to address this issue by extending the CEGAR approach to incorporate conflict analysis when a counterexample is found. At the expense of additional book-keeping, this will allow the solver to non-chronologically backtrack over irrelevant quantifier blocks, greatly improving performance in certain cases.

[1]RAReQS expects its input to be preprocessed with `bloqqer` [5], so the use of an unpreprocessed formula is a bit artificial, but it nonetheless serves to illustrate how the inability to non-chronologically backtrack can be harmful.

Bibliography

[1] C. Ansótegui, C. P. Gomes, and B. Selman. The Achilles' Heel of QBF. In *AAAI 2005*. 2.1

[2] A. Ayari and D. A. Basin. QUBOS: Deciding Quantified Boolean Logic Using Propositional Satisfiability Solvers. In *FMCAD*, 2002. 3.1

[3] M. Benedetti. Evaluating QBFs via Symbolic Skolemization. In *LPAR 2004*. 2.5, 3.1

[4] A. Biere. Resolve and Expand. In *SAT*, 2004. 2.5, 3.1

[5] A. Biere, F. Lonsing, and M. Seidl. Blocked Clause Elimination for QBF. In *CADE*, 2011. 3.5, 1

[6] I. Bogudlov, T. Lev-Ami, T. W. Reps, and M. Sagiv. Revamping TVLA: Making Parametric Shape Analysis Competitive. In *CAV*, pages 221–225, 2007. 4.2

[7] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986. 3

[8] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5), 2003. 3.1

[9] P. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT*, 1977. 4.1

[10] N. Eén and N. Sörensson. An Extensible SAT-solver. In *SAT*, 2003. 3.5

[11] U. Egly, M. Seidl, and S. Woltran. A Solver for QBFs in Nonprenex Form. In *ECAI 2006*. 2.1

[12] R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967. 1

[13] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *DAC 2002*. 2.1

[14] E. Giunchiglia, P. Marin, and M. Narizzano. QuBE 7.0 System Description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7, 2010. 3.5

[15] E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantifier structure in search based procedures for QBFs. In *DATE 2006*. 2.1, 2.5

[16] E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE: A System for Deciding Quantified Boolean Formulas Satisfiability. In *IJCAR*, pages 364–369, 2001. 3.1

[17] A. Goultiaeva, V. Iverson, and F. Bacchus. Beyond CNF: A Circuit-Based QBF Solver. In *SAT 2009*. 2.1, 2.4.1, 2.5, 2.6

[18] H. Jain, C. Bartzis, and E. M. Clarke. Satisfiability Checking of Non-clausal Formulas Using General Matings. In *SAT 2006*. 2.1

[19] M. Janota, W. Klieber, J. Marques-Silva, and E. M. Clarke. Solving QBF with Counterexample Guided Refinement. In *SAT*, pages 114–128, 2012. 3.1, 3.4

[20] M. Janota and J. Marques-Silva. Abstraction-Based Algorithm for 2QBF. In *SAT*, 2011. 3.3

[21] S. Jensen and A. Møller. Type analysis for JavaScript. *SAS 2009*. 4.2

[22] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *SAS'10*. Springer-Verlag, 2010. 4.7

[23] Y. Jung, W. Lee, B. Wang, and K. Yi. Predicate generation for learning-based quantifier-free loop invariant inference. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 205–219, 2011. 4.2

[24] W. Klieber, S. Sapra, S. Gao, and E. M. Clarke. A Non-prenex, Non-clausal QBF Solver with Game-State Learning. In *SAT*, 2010. 2.4.2, 3.4

[25] S. Kong, Y. Jung, C. David, B. Wang, and K. Yi. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. *Programming Languages and Systems*, pages 328–343, 2010. 4.2

[26] T. Lev-Ami and S. Sagiv. TVLA: A System for Implementing Static Analyses. In *SAS*, pages 280–301, 2000. 4.2

[27] A. Loginov, T. W. Reps, and M. Sagiv. Automated Verification of the Deutsch-Schorr-Waite Tree-Traversal Algorithm. In *SAS*, pages 261–279, 2006. 4.2

[28] F. Lonsing and A. Biere. Nenofex: Expanding NNF for QBF Solving. In *SAT*

2008. 2.1, 3.1

[29] K. McMillan. Quantified invariant generation using an interpolating saturation prover. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 413–427, 2008. 4.2

[30] M. Narizzano, L. Pulina, and A. Tacchella. QBFEVAL. `http://www.qbfeval.org/`. 2.5

[31] F. Pigorsch and C. Scholl. Exploiting structure in an AIG based QBF solver. In *DATE 2009*. 2.1, 2.5

[32] The Quantified Boolean Formulas Library. `http://www.qbflib.org/`. 3.5

[33] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74, 2002. 4.1, 4.2

[34] A. Sabharwal, C. Ansótegui, C. P. Gomes, J. W. Hart, and B. Selman. QBF Modeling: Exploiting Player Symmetry for Simplicity and Efficiency. In *SAT 2006*. 2.1

[35] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002. 4.1, 4.2

[36] H. Samulowitz and F. Bacchus. Dynamically Partitioning for Solving QBF. In *SAT 2007*. 2.6

[37] J. P. M. Silva and K. A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996. 5

[38] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. *ACM SIGPLAN Notices*, 44(6):223–234, 2009. 4.2

[39] C. Thiffault, F. Bacchus, and T. Walsh. Solving Non-clausal Formulas with DPLL Search. In *Constraint Programming – CP 2004*. 2.1

[40] L. Zhang. Solving QBF by Combining Conjunctive and Disjunctive Normal Forms. In *AAAI 2006*. 2.1

[41] L. Zhang and S. Malik. Conflict Driven Learning in a Quantified Boolean Satisfiability Solver. In *ICCAD 2002*. 2.3

[42] L. Zhang and S. Malik. Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In *Constraint Programming – CP 2002*. 2.4.2