Form Methods Syst Des (2019) 55:1–32
https://doi.org/10.1007/s10703-016-0267-2

Alloy*: a general-purpose higher-order relational constraint solver

Aleksandar Milicevic[3] · Joseph P. Near[2] ·
Eunsuk Kang[2] · Daniel Jackson[1]

Published online: 27 January 2017

Abstract The last decade has seen a dramatic growth in the use of constraint solvers as a computational mechanism, not only for analysis of software, but also at runtime. Solvers are available for a variety of logics but are generally restricted to first-order formulas. Some tasks, however, most notably those involving synthesis, are inherently higher order; these are typically handled by embedding a first-order solver (such as a SAT or SMT solver) in a domain-specific algorithm. Using strategies similar to those used in such algorithms, we show how to extend a first-order solver (in this case Kodkod, a model finder for relational logic used as the engine of the Alloy Analyzer) so that it can handle quantifications over higher-order structures. The resulting solver is sufficiently general that it can be applied to a range of problems; it is higher order, so that it can be applied directly, without embedding in another algorithm; and it performs well enough to be competitive with specialized tools. Just as the identification of first-order solvers as reusable backends advanced the performance of specialized tools and simplified their architecture, factoring out higher-order solvers may bring similar benefits to a new class of tools.

Keywords Alloy · Model finding · Higher-order logic · Synthesis · Design

✉ Aleksandar Milicevic
almili@microsoft.com

✉ Joseph P. Near
jnear@berkeley.edu

✉ Eunsuk Kang
eunsuk.kang@berkeley.edu

Daniel Jackson
dnj@csail.mit.edu

1 Massachusetts Institute of Technology, Cambridge, MA 02139, USA

2 University of California, Berkeley, USA

3 Microsoft, Cambridge, MA, USA

1 Introduction

As constraint solvers become more capable, they are increasingly being applied to problems previously regarded as intractable. Program synthesis, for example, requires the solver to find a single program that computes the correct output for all possible inputs. This problem is typically encoded as a higher-order formula of the form $\exists p \forall T \bullet \phi(p, T)$, where the domain of T ranges over *sets* instead of individual values (e.g., a set of program execution traces). Higher-order quantifiers are useful for expressing various other problems—for instance, reasoning about reachability over a graph, or finding a maximal or minimal solution to a given optimization problem.

Existing general-purpose solvers, however, are not designed to handle specifications with arbitrary higher-order quantifiers. Instead, tools that require a higher-order quantification use ad hoc methods to adapt existing solvers to the problem. A popular technique for the program synthesis problem is called CEGIS (counterexample guided inductive synthesis) [50], and involves using a first-order solver in a loop: first, to find a candidate program, and second, to verify that it satisfies the specification for all inputs. If the verification step fails, the resulting counterexample is transformed into a constraint that is used in generating the next candidate.

In this paper, we present Alloy*, a general-purpose, higher-order, bounded constraint solver based on the Alloy Analyzer [20]. Alloy is a specification language based on a first-order relational logic; the Alloy Analyzer performs bounded analysis of Alloy specifications. Alloy* admits higher-order quantifier patterns, and uses a general implementation of the CEGIS loop to perform bounded analysis. It retains the syntax of Alloy, and changes the semantics only by expanding the set of specifications that can be analyzed, making it easy for existing Alloy users to adopt.

Alloy* handles higher-order quantifiers in a generic and model-agnostic way, meaning that it allows higher-order quantifiers to appear anywhere where allowed by the Alloy syntax, and does not require any special idiom to be followed. Alloy* first creates a *solving strategy* by decomposing an arbitrary formula (possibly containing nested higher-order quantifiers) into a tree of subformulas and assigning a decision procedure to each of them. Each such tree is either (1) a higher-order "$\exists\forall$" pattern, (2) a disjunction where at least one disjunct is higher-order, or (3) a first-order formula. To solve the "$\exists\forall$" nodes, Alloy* applies CEGIS; for the disjunction leaves, Alloy* solves each disjunct separately; and for first-order formulas, Alloy* uses Kodkod [55].

We have evaluated Alloy* on a variety of case studies taken from the work of other researchers. In the first, we used Alloy* to solve classical higher-order NP-complete graph problems like `max-clique`, and found it to scale well for uses in modeling, bounded verification, and fast prototyping. In the second, we encoded all of the SyGuS [3] program synthesis benchmarks that did not involve 64-bit bitvectors and found that, while state-of-the-art purpose-built synthesizers are typically faster, the performance of Alloy* is competitive with the reference synthesizers provided by the competition organizers.

The contributions of this paper include:

– A framework for extending a first-order solver to the higher-order case, consisting of the design of datatypes and a general algorithm comprising syntactic transformations (skolemization, conversion to negation normal form, etc.) and an incremental solving strategy;
– A collection of case study applications demonstrating the feasibility of the approach in different domains (including synthesis of code, execution and bounded verification of

higher-order NP-hard algorithms), and showing encouraging performance on standard benchmarks;
– The release of a freely available implementation for others to use, comprising an extension of Alloy [1].

2 Example: classical graph algorithms

Classical graph algorithms have become prototypical Alloy examples, showcasing both the expressiveness of the Alloy language and the power of the Alloy Analyzer. Many complex problems can be specified declaratively in only a few lines of Alloy, and then in a matter of seconds fully automatically animated (for graphs of small size) by the Alloy Analyzer. This ability to succinctly specify and quickly solve problems like these—algorithms that would be difficult and time consuming to implement imperatively using traditional programming languages—has found its use in many applications, including program verification [12,16, 37,38,56], software testing [5,10,30,45], fast prototyping [33,46], teaching [9,14,36], etc.

For a whole category of interesting problems, however, the current Alloy engine is not powerful enough. Those are the higher-order problems, for which the specification has to quantify over relations rather than scalars. Many well-known graph algorithms fall into this category, including finding maximal cliques, max cuts, minimum vertex covers, and various coloring problems. In this section, we show such graph algorithms can be specified and analyzed using the new engine implemented in Alloy*.

2.1 Background on alloy

Suppose we want to check Turán's theorem, one of the fundamental results in graph theory [2]. Turán's theorem states that a $(k+1)$-free graph with n nodes can maximally have $\frac{(k-1)n^2}{2k}$ edges. A graph is $(k+1)$-free if it contains no clique with $k+1$ nodes (a clique is a subset of nodes in which every two nodes are connected by an edge).

Figure 1 shows how Turán's theorem might be formally specified in Alloy. At its core, each Alloy model consists of different types of *elements* and *relations* among those elements. The keyword **`sig`** is used to declare a *signature*, which represents a set of elements of a particular type. For example, **`sig`** `Node` introduces a set of elements that correspond to different nodes of a graph (line 1). A signature may be assigned one or more *fields*, each of which introduces a relation that maps an element of that signature to other elements in the model. For instance, field `val` in the `Node` signature is a relation of type `Node` $\times$ `Int`, which maps each node to some integer value that it stores.

An Alloy *predicate* encapsulates a set of logical constraints over zero or more parameters and relations in the model. Predicate `clique`, for example, defines what it means for a graph consisting of the given edges (parameter `edges`) and nodes (`clq`) to be a clique; namely, every distinct pair of nodes in the graph must be connected via some edge (lines 4–7). This predicate, in turn, is used inside `maxClique` to define the notion of a maximal clique (lines 10–16); in particular, `maxClique` asserts that no clique beside given `clq` contains more nodes.

Alloy provides basic operators for manipulating relational expressions. Predicate `edgeProps` illustrates how some of these operators[1] can be used to define what it means

[1] The expression `~r` corresponds to the transpose of relation `r`; (`r1` **`in`** `r2`) evaluates to true if and only if `r1` is a subset of `r2`; **`iden`** is a built-in relation that maps every element to itself, and; `r1 & r2` returns the intersection of the two relations.

```
 1  sig Node  {val: one Int}
 2
 3  // between every two nodes there is an edge
 4  pred clique[edges: Node -> Node, clq: set Node] {
 5    all disj n1, n2: clq |
 6      n1 -> n2 in edges
 7  }
 8
 9  // no other clique with more nodes
10  pred maxClique[edges: Node -> Node, clq: set Node] {
11    clique[edges, clq]
12    no clq2: set Node |
13       clq2 != clq and
14       clique[edges, clq2] and
15       #clq2 > #clq
16  }
17
18  // symmetric and irreflexive
19  pred edgeProps[edges: Node -> Node] {
20    (~edges in edges) and
21    (no edges & iden)
22  }
23
24  // max number of edges in a (k+1)-free graph with n nodes is (k-1)n^2/2k
25  check Turan {
26    all edges: Node -> Node | edgeProps[edges] implies
27      some mClq: set Node {
28        maxClique[edges, mClq]
29        let n = #Node, k = #mClq, e = (#edges).div[2] |
30          e <= k.minus[1].mul[n].mul[n].div[2].div[k]
31      }
32  } for 7 but 0..294 Int
```

Fig. 1 Automatic checking of *Turan's* theorem in Alloy*

for `edges` to be (1) *symmetric* (that is, the transpose of each tuple also belongs to the same relation) and (2) *irreflexive* (it does not contain any tuple that represents a self-loop edge).

Having defined maximal cliques in Alloy, we can proceed to formalize Turán's theorem. The `Turan` command (lines 25–32) asserts that for all possible `edge` relations that are symmetric and irreflexive (line 26), if the max-clique in that graph has k nodes (`k=#mClq`), the number of selected edges (`e=(#edges).div[2]`) must be at most $\frac{(k-1)n^2}{2k}$. (The number of tuples in `edges` is divided by 2 because the graph in the setup of the theorem in undirected.)

2.2 Solving higher-order formulas in Alloy*

Running the `Turan` command in the vanilla version of the Alloy Analyzer is not possible. Although the specification, as given in Fig. 1, is allowed by the Alloy language, trying to execute it causes the Analyzer to immediately return an error: "Analysis cannot be performed since it requires higher-order quantification that could not be skolemized". In Alloy*, in contrast, this check can be automatically performed to confirm that indeed no counterexample can be found within the specified scope. The scope we used (7 nodes, ints from 0 to 294) allows for all possible undirected graphs with up to 7 nodes. The upper bound for ints was chosen to ensure that the formula for computing the maximal number of edges ($\frac{(k-1)n^2}{2k}$) never overflows for $n \leq 7$ (which implies $k \leq 7$). The check completes in about 45 s.

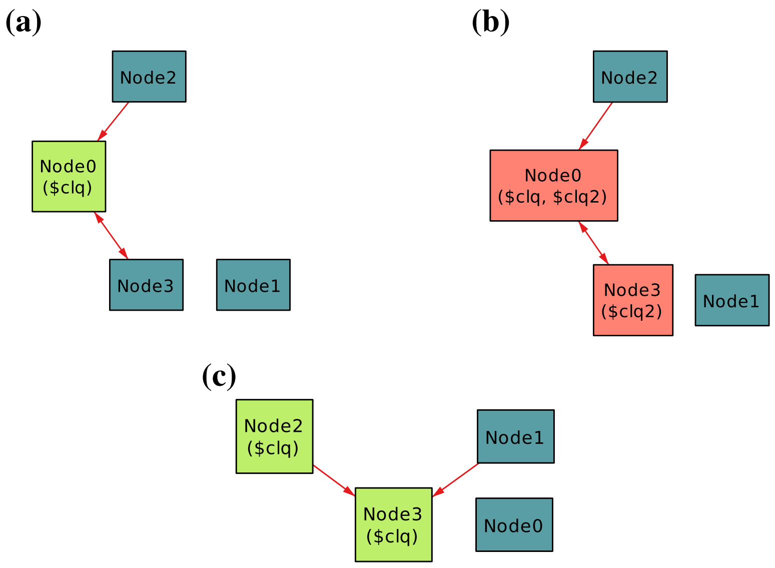

Fig. 2 An automatically generated instance satisfying `maxClique`. An edge from some node n to another n' represents the tuple (n, n') belonging to the `edges` relation in the given candidate instance. **a** A *maxClique* candidate. **b** A counterexample for **a**. **c** Final *maxClique* instance

To explain the analysis problems that higher-order quantifiers pose to the standard Alloy Analyzer, and how those problems are tackled in Alloy*, we look at a simpler task: finding an instance of a graph with a subgraph satisfying the `maxClique` predicate. The problematic quantifier in this case is the inner "`no clq2: set Node | ...`" (line 12) constraint, which requires checking that for all possible subsets of `Node`, not one of them is a clique with more nodes than the given set `clq`. A direct translation into the current SAT-based backend would require the Analyzer to explicitly, and upfront, enumerate all possible subsets of `Node`—which would be prohibitively expensive. Instead, Alloy* implements the CEGIS approach:

1. First, it finds a candidate instance, by searching for a clique `clq` and *only one* set of nodes `clq2` that *is not* a clique larger than `clq`. A possible first candidate is given in Fig. 2a (with the clique nodes are highlighted in green). At this point `clq2` could have been anything that is either not a clique or not larger than `clq`.
2. Next, Alloy* attempts to falsify the previous candidate by finding, again, *only one* set of nodes `clq2`, but this time such that `clq2` *is* a clique larger than `clq`, for the exact (concrete) graph found in the previous step. In this case, it finds one such *counterexample* clique (red nodes in Fig. 2b) refuting the proposition that `clq` from the first step is a maximal clique.
3. Alloy* continues by trying to find another candidate clique, encoding the previous counterexample to prune the remainder of the search space (as explained in detail in Sects. 4 and 5). After several iterations, it finds the candidate in Fig. 2c which cannot be refuted, so it returns that candidate as a satisfying solution.

Once written, the `maxClique` predicate (despite containing a higher-order quantification) can be used in other parts of the model, like any other predicate, just as we used it to formulate and check Turán's theorem. In fact, the `turan` check contains another higher-order quantifier, so the analysis ends up spawning two nested CEGIS loops and exhaustively iterating over them; every candidate instance and counterexample generated in the process can be opened and inspected in the Alloy visualizer. (For a screenshot of the user interface, see [1].)

3 Example: policy synthesis

Policy design and analysis is an active area of research. Several approaches [19,39,47] propose a declarative language for specifying policies, and employ a constraint-based analysis to check them against a high-level property. In this section, we demonstrate how Alloy* can be used to automatically *synthesize* a policy that satisfies a given property.

Figure 3 shows an Alloy model that describes the problem of grade assignment at a university, based on the running example from [15]. A policy specification contains three basic concepts: *roles*, *actions*, and *resources*. A system consists of a set of users, each having one or more roles and performing actions on a set of resources. A *policy* (`acl`) is represented as a relation of type `Role × Action × Resource`, where tuple `(ro,a,r)` specifies that users with role `ro` may perform action `a` on resource `r`. For example, a policy containing only a single tuple `Faculty->Assign->ExtGrade` means that a user may assign an external grade only if it has the `Faculty` role.

There are two desirable properties of the system: (1) students should not be able to assign external grades (line 22), and (2) no user should be able to both assign and receive external grades (line 27). The behavior of the system is modeled as relation `performs`, where tuple `(u,a,r)` means that user `u` performs action `a` on resource `r`. We say that a policy is *enforced* if, given some assignment of roles to users (`roles`), the system allows user `u` to perform action `a` on resource `r` only if `u` has a role that is specified by the policy as being allowed to perform `a` on `r` (lines 9–11).

A policy is considered *valid* if and only if, when enforced, it ensures that the properties hold over *all possible combinations* of user roles and behaviors (lines 37–41). Typically, in Alloy, the policy designer would specify a particular policy and check whether `valid` holds over it. Instead, here we will ask Alloy* to *generate* a policy (`acl`) that satisfies the predicate; in particular, the quantification over `roles` and `performs` triggers the higher-order solving facility of Alloy*, which will use the CEGIS approach to search for a valid `acl`.

Executing the command on line 52 completes in about 0.5 s, and returns an *empty* policy, which essentially prevents all users from performing any actions. This policy is technically valid, but also not very useful! Fortunately, we can leverage the higher-order feature of Alloy* to synthesize more interesting policies. In particular, we will define a policy to be *most permissive* if there is no other valid policy contains more tuples, thus allowing a greater number of actions to be performed on resources (lines 44–50).

Running the new command on line 53 takes about 3.5 s, and generates the following policy:

```
{Faculty,Receive,ExtGrade}, {Faculty,Assign,Resource},
{Student,Receive,Resource}, {Student,Assign,IntGrade},
      {TA,Receive,Resource}, {TA,Assign,IntGrade}
```

This policy provides a starting point for further exploration of the policy space. The designer may decide, for example, that students should not be able to assign `IntGrade`, add another property that states this new requirement, and then repeat the synthesis process.

```
/* Basic signatures */
abstract sig Resource {}           abstract sig Role {}
abstract sig Action {}             sig User {}

/* 'performs' describes the behavior of users */
pred enforce[acl:      Role->Action->Resource,
             roles:    User->Role,
             performs: User->Action->Resource] {
  all u: User, a: Action, r: Resource |
    /* 'u' can perform 'a' on 'r' only if allowed by 'acl' */
    u->a->r in performs => (some ro: u.roles | ro->a->r in acl)
}

/* Domain-specific concepts */
one sig Faculty,  Student, TA extends Role {}
one sig IntGrade, ExtGrade    extends Resource {}
one sig Assign,   Receive     extends Action {}

/* Properties */
pred prop1[roles: User->Role, performs: User->Action->Resource] {
  /* no student can assign external grade */
  no u: User | u.roles = Student and Assign->ExtGrade in u.performs
}

pred prop2[roles: User->Role, performs: User->Action->Resource] {
  /* no user can both receive and assign external grades */
  no u: User | Assign + Receive in u.performs.ExtGrade
}

/* Assumption: no user can both be a faculty and a student/TA */
pred noDualRoles[roles: User->Role] {
  no u: User |
    Faculty in u.roles and
    some (Student + TA) & u.roles

/* 'acl' satisfies properties over every user role and behavior */
pred valid[acl: Role->Action->Resource] {
  all roles: User->Role, performs: User->Action->Resource |
    (enforce[acl, roles, performs] and noDualRoles[roles]) implies
      (prop1[roles, performs] and prop2[roles, performs])
}

/* 'acl' allows the most number of actions while being valid */
pred mostPermissive[acl: Role->Action->Resource] {
  valid[acl]
  no acl': Role->Action->Resource |
    acl != acl' and
    valid[acl'] and
    #acl' > #acl
}

run valid for 5
run mostPermissive for 5
```

Fig. 3 Grade assignment policy in Alloy*

4 Background and key ideas

Skolemization Many first-order constraint solvers allow some form of higher-order quantifiers to appear at the language level. Part of the reason for this is that, in certain cases, quantifiers can be eliminated in a preprocessing step called *skolemization*. In a model finding setting, every top-level existential quantifier is eliminated by (1) introducing a *skolem constant* for the quantification variable, and (2) replacing every occurrence of that variable with the newly created skolem constant. For example, solving `some s: set univ |#s > 2`, which is higher-order, is equivalent to solving `$s in univ && #$s > 2`, which is first-order and thus solvable by general purpose constraint solvers. (Throughout, following the Alloy convention, skolem constants will be prefixed with a dollar sign.)

CEGIS CounterExample-Guided Inductive Synthesis [50] is an approach for solving higher-order synthesis problems, which is extended in Alloy* to the general problem of solving higher-order formulas. As briefly mentioned before, the CEGIS strategy applies only to formulas in the form $\exists p \forall e \cdot s(p, e)$ and prescribes the following three steps:

(1) search: attempt to find a candidate value for p by solving $\exists p \exists e \cdot s(p, e)$—a first-order problem;

(2) verification: if a candidate $\$p$ is found, try to verify it by checking if it holds for all possible bindings for e. The *verification condition*, thus, becomes $\forall e \cdot s(\$p, e)$. This check is done by refutation, i.e., by satisfying the negation of the verification condition; pushing the negation through yields $\exists e \cdot \neg s(\$p, e)$, which, again, is first-order.

(3) induction: if the candidate is verified, a solution is found and the algorithm terminates. Otherwise a concrete counterexample $\$e_{cex}$ is found. The search continues by searching for another candidate which must also satisfy the counterexample, that is, solving $\exists p \exists e \cdot s(p, e) \wedge s(p, \$e_{cex})$. This strategy in particular tends to be very effective at reducing the search space and improving the overall scalability.

CEGIS for a general purpose solver Existing CEGIS-based synthesis tools implement this strategy internally, optimizing for the target domain of synthesis problems. The key insight of this paper is that the CEGIS algorithm can be implemented, generically and efficiently, inside a general purpose constraint solver. For an efficient implementation, it is important that such a solver supports the following:

- *Partial instances* The verification condition must be solved against the previously discovered candidate; explicitly designating that candidate as a "partial instance", i.e., a part of the solution known upfront, is significantly more efficient than encoding it with constraints [55].
- *Incremental solving* Except for one additional constraint, the induction step solves exactly the same formula as the search step. Many modern SAT solvers are capable of *incremental solving*; that is, by reusing knowledge (e.g, learned clauses) obtained while solving a previous formula, an incremental solver may be able to solve the conjunction of the formula with additional constraints more efficiently.
- *Atoms as expressions* The induction step needs to be able to convert a concrete counterexample (given in terms of concrete atoms, i.e., values for each variable) to a formula to be added to the candidate search condition. All atoms, therefore, must be convertible to expressions. This is trivial for SAT solvers, but requires extra functionality for solvers offering a richer input language.

$$\begin{array}{rrcl}
\text{Variables} & x & \in & x_1, x_2, \ldots \\
\text{Expressions} & e & \in & e_1, e_2, \ldots \\
\text{Formulas} & f & ::= & e_1 = e_2 \mid f_1 \land f_2 \mid f_1 \lor f_2 \mid f_1 \Rightarrow f_2 \mid \neg f \\
 & & \mid & \forall x : e.f \mid \exists x : e.f \\
\text{Propositions} & p & ::= & \mathtt{FOL}(f) \mid \mathtt{OR}(\{f_1, \ldots, f_n\}) \mid \exists\forall(f, \{f_1, \ldots, f_n\}, \{f'_1, \ldots, f'_n\})
\end{array}$$

Fig. 4 Core syntax of formulas and propositions

– *Skolemization* Skolemizing higher-order existential quantifiers is necessary for all three CEGIS steps.

We formalize our approach in Sect. 5, assuming availability of a first-order constraint solver offering all the features above. In Sect. 6 we present our implementation as an extension to Kodkod [53] (a first-order relational constraint solver already equipped with most of the required features).

5 Semantics

This section describes our general-purpose decision procedure for bounded higher-order logic. The approach has two basic components: a **translation** from higher-order formulas to *propositions*, described in Sect. 5.1, and an **algorithm**, based on CEGIS, for solving the generated propositions, described in Sect. 5.2. Our approach is general: the algorithm requires a first-order solver, but any off-the-shelf solver can be used.

In Sect. 5.3, we outline the argument that our approach is both *sound* (the solutions it finds are correct) and *complete* (if a solution exists, it will be found).

5.1 Translation

Figure 4 summarizes the syntax of higher-order formulas and the *propositions* targeted by our translation. Propositions are designed to represent the class of higher-order formulas solvable by the CEGIS strategy. The types of propositions are:

– `FOL`: a first-order formula that can be solved in one step by the solver
– `OR`: a disjunction of formulas
– `∃∀`: a conjunction of a first-order formula, a number of higher-order formulas for the CEGIS algorithm to solve, and the original universally-quantified formulas used to generate them.

For simplicity of exposition, we exclude the treatment of bounds from our formalization; Sect. 6.1 describes how the bounds are constructed before every solver invocation.

Normalizing formulas The first step in translating a higher-order formula is to convert the formula to negation normal form (NNF) and skolemize top-level existential quantifiers. Conversion to NNF pushes the quantifiers towards the roots of the formula, while skolemization eliminates top-level existential quantifiers (including the higher-order ones). Normalizing formulas this way simplifies translation. The normalization process is formalized in Fig. 5; the rules are standard. The Skolemization rule, SKOLEM, relies on the ability to construct fresh Skolem variables and substitute them into the formula. For an input formula f, we denote normalization into formula f' by $f \overset{\star}{\rightarrow}_n f'$.

Rewriting formulas into propositions Figure 6 contains the rules for rewriting formulas into propositions. We denote rewriting a formula f into a proposition p by $f \rightarrow_t p$.

$$\boxed{f \rightarrow_n f}$$

$$\textsc{Skolem}\ \frac{x' \text{ is a new skolem variable} \quad f\{x \mapsto x'\} \rightarrow_n f'}{\exists x : e.f \rightarrow_n f'}$$

$$\frac{f_1 \rightarrow_n f_1' \quad f_2 \rightarrow_n f_2'}{f_1 \Rightarrow f_2 \rightarrow_n \neg f_1' \vee f_2'} \qquad \frac{f_1 \rightarrow_n f_1' \quad f_2 \rightarrow_n f_2'}{\neg(f_1 \vee f_2) \rightarrow_n \neg f_1' \wedge \neg f_2'} \qquad \frac{f_1 \rightarrow_n f_1' \quad f_2 \rightarrow_n f_2'}{\neg(f_1 \wedge f_2) \rightarrow_n \neg f_1' \vee \neg f_2'}$$

$$\frac{f \rightarrow_n f'}{\neg\neg f \rightarrow_n f'} \qquad \frac{f \rightarrow_n f'}{\neg \exists x : e.f \rightarrow_n \forall x : e.\neg f'} \qquad \frac{f \rightarrow_n f'}{\neg \forall x : e.f \rightarrow_n \exists x : e.\neg f'}$$

Fig. 5 Normalization of formulas

The first two rules rewrite quantifiers. If the quantified formula is first-order, it is simply tagged with `FOL` (rule QUANT- FOL). If it is higher-order, the $\forall$ quantifier is changed to an $\exists$, and the resulting proposition is tagged with $\exists\forall$ (rule QUANT- HOL). This proposition, containing the existential version of a previously universally-quantified formula, is exactly the right input for finding a counterexample for a given candidate using CEGIS. The $\exists\forall$ syntax contains both the translated proposition (with flipped quantifier) and the original (universally quantified) formula. The original formula is re-used during solving to check the validity of candidate solutions.

The remaining rules deal with composing propositions. Composition of propositions is straightforward for the most part, directly following the distributivity laws of conjunction over disjunction and vice versa. Our goal is to minimize the number of CEGIS iterations required to solve the resulting proposition, so this composition must be designed carefully in order to minimize the number of $\exists\forall$ tags in the final proposition; the complicated rules for composition accomplish this. For example, a conjunction of two $\exists\forall$ propositions can be merged into a single $\exists\forall$, as can a conjunction of a `FOL` and an $\exists\forall$. With disjunction, however, we need to be more careful: since skolemization through disjunction is not sound, it would be wrong, for example, to try and recombine two `FOL` propositions into a single `FOL`. Instead, a safe optimization (which we implemented in Alloy*) would be to first check if $f_1 \vee f_2$ is first-order as a whole, and if so return `FOL`($f_1 \vee f_2$).

Example Consider a simple example, written in the syntax of Alloy*. This formula asks if some integer exists smaller than the number of nodes of every possible graph. The formula is true for the integer -1, because every graph has zero or more nodes.

```
some s: int | all ns: set Node | #ns > s
```

Call this formula f. Normalizing f requires skolemizing the variable s. Since the quantifier is higher-order, translating it into a proposition produces an $\exists\forall$, by rule QUANT- HOL:

$$\begin{aligned} f \quad &\overset{\star}{\rightarrow}_n \quad \forall ns.\#ns > \$s \\ &\rightarrow_t \quad \exists\forall(\texttt{TRUE}, \{\exists ns.\#ns > \$s\}, \{\forall ns.\#ns > \$s\}) \end{aligned}$$

5.2 Satisfiability solving

The procedure for satisfiability solving is given in Fig. 7. A first-order formula (enclosed in `FOL`) is given to the solver to be solved directly, in one step (line 6). An `OR` proposition is solved by attempting to solve its disjuncts (lines 9–11). An instance is returned as soon as one is found; otherwise, the algorithm returns $\bot$.

$$\boxed{f \to_t p}$$

$$\text{QUANT-FOL}\ \frac{\forall x : e.f \in FOL}{\forall x : e.f \to_t \texttt{FOL}(\forall x : e.f)} \qquad \text{QUANT-HOL}\ \frac{\forall x : e.f \notin FOL}{\forall x : e.f \to_t \exists\forall(\texttt{TRUE}, \{p\}, \{\forall x : e.f\})}$$

$$\frac{f_1 \to_t \texttt{FOL}(f_1') \quad f_2 \to_t \texttt{FOL}(f_2')}{f_1 \wedge f_2 \to_t \texttt{FOL}(f_1' \wedge f_2')} \qquad \frac{f_1 \to_t \texttt{FOL}(f_1') \quad f_2 \to_t \texttt{OR}(F)}{f_1 \wedge f_2 \to_t \texttt{OR}(\{f_1' \wedge f \mid f \in F\})}$$

$$\frac{f_1 \to_t \texttt{FOL}(f_1') \quad f_2 \to_t \exists\forall(f_q, F_q, F_o)}{f_1 \wedge f_2 \to_t \exists\forall(f_1' \wedge f_q, F_q, F_o \cup \{f_1'\})} \qquad \frac{f_1 \to_t \texttt{OR}(F_1) \quad f_2 \to_t \texttt{OR}(F_2)}{f_1 \wedge f_2 \to_t \texttt{OR}(\{f_1' \wedge f_2' \mid f_1', f_2' \in F_1 \times F_2\})}$$

$$\frac{f_1 \to_t \texttt{OR}(F) \quad f_2 \to_t \exists\forall(f_q, F_q, F_o)}{f_1 \wedge f_2 \to_t \texttt{OR}(\{f_1' \wedge f_2' \mid f_1', f_2' \in F \times F_o\})}$$

$$\frac{f_1 \to_t \exists\forall(f_{q_1}, F_{q_1}, F_{o_1}) \quad f_2 \to_t \exists\forall(f_{q_2}, F_{q_2}, F_{o_2})}{f_1 \wedge f_2 \to_t \exists\forall(f_{q_1} \wedge f_{q_2}, F_{q_1} \cup F_{q_2}, F_{o_1} \cup F_{o_2})}$$

$$\frac{f_1 \to_t \texttt{FOL}(f_1') \quad f_2 \to_t \texttt{FOL}(f_2')}{f_1 \vee f_2 \to_t \texttt{FOL}(f_1' \vee f_2')} \qquad \frac{f_1 \to_t \texttt{FOL}(f_1') \quad f_2 \to_t \texttt{OR}(F)}{f_1 \vee f_2 \to_t \texttt{OR}(\{f_1'\} \cup F)}$$

$$\frac{f_1 \to_t \texttt{FOL}(f_1') \quad f_2 \to_t \exists\forall(f_q, F_q, F_o)}{f_1 \vee f_2 \to_t \texttt{OR}(\{f_1'\} \cup F_o)} \qquad \frac{f_1 \to_t \texttt{OR}(F_1) \quad f_2 \to_t \texttt{OR}(F_2)}{f_1 \vee f_2 \to_t \texttt{OR}(F_1 \cup F_2)}$$

$$\frac{f_1 \to_t \texttt{OR}(F) \quad f_2 \to_t \exists\forall(f_q, F_q, F_o)}{f_1 \vee f_2 \to_t \texttt{OR}(F \cup F_o\})}$$

$$\frac{f_1 \to_t \exists\forall(f_{q_1}, F_{q_1}, F_{o_1}) \quad f_2 \to_t \exists\forall(f_{q_2}, F_{q_2}, F_{o_2})}{f_1 \vee f_2 \to_t \texttt{OR}(F_{o_1} \cup F_{o_2})}$$

$$\frac{f_2 \wedge f_1 \to_t p}{f_1 \wedge f_2 \to_t p} \qquad \frac{f_2 \vee f_1 \to_t p}{f_1 \vee f_2 \to_t p}$$

Fig. 6 Translation of higher-order formulas to propositions

The procedure for the $\exists\forall$ proposition implements the CEGIS loop (lines 14–30), following the algorithm in Sect. 4. The candidate condition is a conjunction of the first-order f and the existential propositions $f_{q1}, \ldots, f_{qn}$ constructed by the translation process (line 15); the verification condition is a conjunction of all original universal quantifiers within this $\exists\forall$. Encoding the counterexample back into the search formula boils down to obtaining a concrete value that each quantification variable has in that counterexample (in the form of a substitution σ returned from the first-order solver—line 19) and embedding that value directly in the body of the corresponding quantifier (line 21). Finally, we attempt to solve the new conjunction of the original formula and the counterexample.

Example Consider solving our running example. Line 3 of the solving algorithm sets $p = \exists\forall(\texttt{TRUE}, \{\exists ns.\#ns > \$s\}, \{\forall ns.\#ns > \$s\})$, and the third case of the top-level conditional is chosen. Line 15 sets $p_{cand} = \texttt{TRUE} \wedge \exists ns.\#ns > \$s$. Solving this formula results in a candidate solution, perhaps setting ns to have 3 nodes and $s = 1$.

Next, line 21 sets f_{check} by applying the candidate substitution to get $f_{check} = \forall ns.\#ns > 1$. Line 26 attempts to find a counterexample to f_{check}, and in this case, might set ns to a graph with zero nodes. The final step is to encode this counterexample in f_{cex}: line 28 sets f_{cex} to $\#\varnothing > \$s$. Line 30 invokes the next iteration of CEGIS on the formula:

```
1.  define $\mathcal{S}(f)$:
2.      – normalize and translate the formula f
3.      find $p$ such that $f \overset{\star}{\to}_n f'$ and $f' \to_t p$
4.
5.      if $p$ has form $\texttt{FOL}(f)$:
6.          return solve($f$)
7.
8.      if $p$ has form $\texttt{OR}(\{f_1, ..., f_n\})$:
9.          find $f_k \in f_1, ..., f_n$ such that $\mathcal{S}(f_k) \neq \bot$
10.         if $f_k$ exists, return $\mathcal{S}(f_k)$
11.         else return $\bot$
12.
13.     if $p$ has form $\exists\forall(f, \{f_{q_1}, ..., f_{q_n}\}, \{f_{o_1}, ..., f_{o_n}\})$:
14.         – try to find a candidate solution
15.         let $p_{cand} = f \wedge f_{q_1} \wedge ... \wedge f_{q_n}$
16.         if $\mathcal{S}(p_{cand}) = \bot$, return $\bot$
17.         else    – check whether the candidate is a true solution
18.             – call the candidate mapping from variables to values $\sigma_{cand}$
19.             let $\sigma_{cand} = \{x_1 \mapsto v_1, x_2 \mapsto v_2, ..., x_m \mapsto v_m\} = \mathcal{S}(p_{cand})$
20.             – use $\sigma_{cand}$ to replace variables in the original formula with candidate values
21.             let $f_{check} = \sigma_{cand}(f_{o_1} \wedge ... \wedge f_{o_n})$
22.             – if no counterexample is found, the candidate is a true solution
23.             if $\mathcal{S}(\neg f_{check}) = \bot$, return $\mathcal{S}(p_{cand})$
24.             else    – otherwise, add a counterexample and iterate
25.                 – call the counterexample's mapping from variables to values $\sigma_{cex}$
26.                 let $\sigma_{cex} = \mathcal{S}(\neg f_{check})$
27.                 – apply the substitution $\sigma_{cex}$ to the original formula
28.                 let $f_{cex} = \sigma_{cex}(f_{o_1} \wedge f_{o_2} \wedge ... \wedge f_{o_n})$
29.                 – try to solve the conjunction of the formula and the counterexample
30.                 return $\mathcal{S}(f_{o_1} \wedge f_{o_2} \wedge ... \wedge f_{o_n} \wedge f_{cex})$
```

Fig. 7 The generalized CEGIS algorithm for solving higher-order formulas

$$\forall ns.\#ns > \$s \quad \wedge \quad \#\varnothing > \$s$$

Normalized and translated, this formula becomes:

$$\exists\forall(\#\varnothing > \$s, \{\exists ns.\#ns > \$s\}, \{\forall ns.\#ns > \$s \quad \wedge \quad \#\varnothing > \$s\})$$

This iteration, line 15 sets $p_{cand} = \#\varnothing > \$s \wedge \exists ns.\#ns > \$s$, and the only possible solutions set $\$s$ to a negative integer. If the candidate solution sets $\$s = -1$, then line 21 sets $f_{check} = \forall ns.\#ns > -1 \wedge \#\varnothing > -1$. The negation of this formula has no counterexample, since no graphs have fewer than zero nodes, so line 23 returns the candidate solution as the final solution.

5.3 Correctness

This section outlines the argument that our approach for solving higher-order formulas is correct. The correctness of our approach is based on the preservation of formula semantics by the translation from formulas to propositions, and the correctness of the CEGIS algorithm. We argue that the combination is both sound (if it finds a solution, then the solution is correct) and complete (if a solution exists, our approach will find it).

More formally, given an oracle $\mathcal{O}$ mapping higher-order formulas to models $\mathcal{M}$, we claim that:

$$\forall f.\mathcal{O}(f) = m \iff \exists f', p.f \overset{\star}{\to}_n f' \wedge f' \to_t p \wedge \mathcal{S}(p) = m$$

Soundness The soundness of our approach rests on the fact that the CEGIS algorithm confirms the candidate solution *using the original formula*. After substituting witnesses from the candidate solution into the original formula, checking whether a counterexample exists is a first-order problem. In the formula $\exists x.\forall y.f$, for example, the CEGIS algorithm first finds a candidate witness for x, then substitutes the witness for each occurrence of x in f to obtain f'. Finally, the solver is asked to solve $\forall y.f'$. As long as the substitution process is done correctly, we need only rely on the solver to produce a correct answer for the substituted formula in order to achieve soundness in the overall approach.

Completeness Showing that our approach is complete is slightly more involved. In particular, if our translation process is incorrect, or the CEGIS algorithm might miss counterexamples, then our approach may incorrectly report that no solution exists. The argument for correctness has two parts:

1. The translation process correctly preserves the semantics of formulas
2. The CEGIS algorithm correctly considers the entire space of possible witnesses for each existential quantifier

To show part (1), we examine the translation rules in Fig. 6 (we rely on previous work to show the soundness and completeness of negation normal form and skolemization). The rules for composition follow the standard rules for distribution conjunction and disjunction, and therefore do not change the semantics of the input formula. Similarly, when the formula is first-order (rule QUANT- FOL), the formula is unchanged. The only tricky case is the rule QUANT- HOL, which flips the quantifier from universal to existential to allow for finding a candidate witness. As we will see, the CEGIS algorithm handles this case in a special way to preserve semantics.

To show part (2), we examine the algorithm in Fig. 7. The `FOL` and `OR` cases are straightforward. In the `∃∀` case, the algorithm constructs a candidate witness by solving the formulas constructed by the rule QUANT- HOL during translation. The algorithm constructs p_{cand}, a formula whose solution space represents *all possible witnesses* for the existentially quantified variables. This follows from the fact that p_{cand} is constructed by simply flipping the type of the quantifier in the original formula. The resulting counterexample σ encodes *only a single possible witness*, and because it is used only when the candidate check has failed, we are guaranteed that *this witness is not the correct one*. The algorithm encodes the failed witness in the formula f_{cex} and starts a new iteration.

A single iteration of the CEGIS loop, then, takes the entire space p_{cand} of possible witnesses and eliminates from consideration a single impossible witness represented by f_{cex}. The algorithm continues to loop until a correct witness is found or the set of witnesses considered and rejected is equal to p_{cand}. The algorithm therefore considers all possible witnesses, and is complete.

6 Implementation

We implemented our decision procedure for higher-order constraint solving as an extension to Kodkod [53]. Kodkod, the backend engine used by the Alloy Analyzer, is a bounded constraint solver for *relational* first-order logic (thus, ‘variable’, as used previously, translates to ‘relation’ in Kodkod, and ‘value’ translates to ‘tuple set’). It works by translating a given relational formula (together with bounds finitizing relation domains) into an equisatisfiable propositional formula and using an of-the-shelf SAT solver to check its satisfiability. The

Alloy Analyzer delegates all its model finding (constraint solving) tasks to Kodkod. No change was needed to the existing translation from Alloy to Kodkod.

The official Kodkod distribution already offers most of the required features identified in Sect. 4. While efficient support for partial instances has always been an integral part of Kodkod, only the latest version (2.0) comes with incremental SAT solvers. Kodkod performs skolemization of top-level (including higher-order) existential quantifiers.

Conversion from atoms to expressions, however, was not available in Kodkod prior to this work. Being able to treat all atoms from a single domain as indistinguishable helps generate a stronger symmetry-breaking predicate. We extended Kodkod with the ability to create a singleton relation for each declared atom, after which converting atoms back to expressions (relations) becomes trivial. We also updated the symmetry-breaking predicate generator to ignore all such singleton relations that are not explicitly used. As a result, this modification does not seem to incur any performance overhead; we ran the existing Kodkod test suite with and without the modification and observed no time difference (in both cases running the 249 tests took around 230s).

Our Java implementation directly follows the semantics defined in Figs. 6 and 7. Additionally, it performs the following important optimizations: (1) rather than creating an `OR` node for every disjunction, it first checks if the disjunction is a first-order formula as a whole, in which case it creates a `FOL` node instead, and (2) it uses incremental solving to implement line 30 from Fig. 7 whenever possible.

6.1 Treatment of bounds

Bounds are a required input of any bounded analysis; for an analysis involving structures, the bounds may include not only the cardinality of the structures, but may also indicate that a structure includes or excludes particular tuples. Such bounds serve not only to finitize the universe of discourse and the domain of each variable, but may also specify a *partial instance* that embodies information known upfront about the solution to the constraint. If supported by the solver, specifying the partial instance through bounds (as opposed to enforcing it with constraints) is an important mechanism that generally improves scalability significantly.

Bounds may change during the translation phase by means of skolemization: every time an existential quantifier is skolemized, a fresh variable is introduced and a bound for it is added. Therefore, we associate bounds with propositions, as different propositions may have different bounds. Whenever a composition of two propositions is performed, the resulting proposition gets the union of the two corresponding bounds.

During the solving phase, whenever the *solve* function is applied, bounds must be provided as an argument. We simply use the bounds associated with the input proposition. When supplying bounds for the translation of the verification condition, it is essential to encode the candidate solution (*cand*) as a partial instance, to ensure that the check is performed against that particular candidate, and not some other arbitrary one. This is done by bounding every variable in the proposition to the exact value it was given in *cand*.

Finally, when translating the formula obtained from the counterexample to be used in a search for the next candidate, the same bounds are used as for the current candidate.

7 Case study: program synthesis

Program synthesis is one of the most popular applications of higher-order constraint solving. Given a logical formula ϕ representing a function specification, the functional synthesis problem involves finding a function definition f such that ϕ is valid in f. Since the specification ϕ must hold for all inputs, the logical formulation of the synthesis problem is:

$$\exists f.\forall i.f(i) = \phi(i)$$

This quantifier alternation is an example of a higher-order constraint. Many approaches to solving this constraint implement some form of CEGIS, along with a strategy for shrinking the search space to a more manageable size.

Syntax-guided synthesis constrains the problem in an additional way: the specification of the function to be synthesized comes with a grammar restricting the syntactic forms to be considered during the search for a solution. A restricted grammar can have the effect of shrinking the search space considerably, making the problem much easier to solve; at the same time, a restricted grammar specific to the problem domain is usually more than expressive enough to contain the solution.

7.1 The SyGuS project

The SyGuS [3] project has proposed an extension to SMTLIB for encoding such problems. The project has also organized a competition between solvers for the format, and provides three reference solvers. The SyGuS benchmarks consist of problem specifications like the one in Fig. 8, which specifies the problem of synthesizing a function to find the maximum of two integers x and y.

The `synth-fun` form specifies the function's name (`max2`) and its type (here, `Int` → `Int` → `Int`), plus the grammar restricting its definition. For `max2`, the grammar specifies

```
(set-logic LIA)
(synth-fun max2 ((x Int) (y Int)) Int
    ((Start Int (x
                 y
                 0
                 1
                 (+ Start Start)
                 (- Start Start)
                 (ite StartBool Start Start)))
     (StartBool Bool ((and StartBool StartBool)
                      (or  StartBool StartBool)
                      (not StartBool)
                      (<=  Start Start)
                      (=   Start Start)
                      (>=  Start Start)))))
(declare-var x Int)
(declare-var y Int)
(constraint (>= (max2 x y) x))
(constraint (>= (max2 x y) y))
(constraint (or (= x (max2 x y))
                (= y (max2 x y))))
(check-synth)
```

Fig. 8 `max2.sl`: a specification of the synthesis problem of defining a function to find the maximum of two integers references to x and y, the integer literals 0 and 1, addition, subtraction, and conditional expressions. Boolean expressions may contain logical connectives and compare integers. From this specification, solvers are expected to synthesize a function definition like the following:

```
(ite (>= x y) x y)
```

The SyGuS project has collected a total of 173 such problems (though many of these specify the same problem at different difficulty levels). All benchmarks specify loop-free functions. They fall into several basic categories: those that involve primarily integer arithmetic, those that use bitvectors, those that use extra language features like multiple functions or let-binding, and the *Hacker's Delight* benchmarks, a set of bit-twiddling strategies that encode useful algorithms in optimized bitvector operations.

7.2 SyGuS in Alloy*

We encoded a subset of the SyGuS benchmarks in Alloy* to test its expressive power and scalability. These benchmarks are a good target for evaluating Alloy*, since they have a standard format, are well tested, and allow comparison to the performance of the reference solvers.

We did not implement the "icfp" problems from the SyGuS collection, because those problems make use of 64-bit bitvectors, which Alloy* does not support. None of the reference solvers is capable of solving these problems, either. Our set of converted benchmarks therefore includes 123 of the total 173 SyGuS problems. These are available as part of the Alloy* distribution.

To demonstrate our strategy for encoding program synthesis problems in Alloy*, we present the Alloy* specification for the `max-2` benchmark from Fig. 8. In Alloy*, we use signatures to represent the production rules of the program grammar, and predicates to represent both the semantics of programs and the constraints restricting the target program's semantics. Programs are composed of abstract syntax nodes, which can be integer- or boolean-typed.

```
abstract sig Node {}
abstract sig IntNode, BoolNode extends Node {}
abstract sig Var              extends IntNode {}
one      sig X, Y             extends Var {}
sig ITE extends IntNode {
  condition: BoolNode,
  then, elsen: IntNode
}
sig GTE extends BoolNode {
  left, right: IntNode
}
```

Integer-typed nodes include variables and if-then-else expressions, while boolean-typed nodes include greater-than-or-equal expressions. Programs in this space evaluate to integers or booleans; integers are built into Alloy, but we must model boolean values explicitly.

```
abstract sig Bool{}   one sig BoolTrue, BoolFalse extends
  Bool{}
```

The standard evaluation semantics can be encoded in a predicate that constrains the evaluation relation. It works by constraining all compound syntax tree nodes based on the results

```
(set-logic BV)

(define-fun parity ((a Bool) (b Bool) (c Bool) (d Bool)) Bool
  (xor (not (xor a b)) (not (xor c d))))

(synth-fun AIG ((a Bool) (b Bool) (c Bool) (d Bool)) Bool
 ((Start Bool ((and Start Start) (not Start) a b c d))))

(declare-var a Bool)
(declare-var b Bool)
(declare-var c Bool)
(declare-var d Bool)
```

Fig. 9 `parity-AIG-d1.sl`: a specification of the synthesis problem of defining a parity-checking circuit with AND and NOT gates

of evaluating their children, but does not constrain the values of variables, allowing them to range over all values.[2]

```
pred semantics[eval: Node -> (Int+Bool)] {
  all n: ITE | eval[n] in Int and
    eval[n.condition] = BoolTrue implies
      eval[n.then] = eval[n] else eval[n.elsen] = eval[n]
  all n: GTE | eval[n] in Bool and
    eval[n.left] >= eval[n.right] implies
      eval[n] = BoolTrue else eval[n] = BoolFalse
  all v: Var | one eval[v] and eval[v] in Int }
```

The specification says that the maximum of two numbers is equal to one of them, and greater than or equal to both.

```
pred spec[root: Node, eval: Node -> (Int+Bool)] {
  (eval[root] = eval[X] or eval[root] = eval[Y]) and
  (eval[root] >= eval[X] and eval[root] >= eval[Y]) }
```

Finally, the problem itself requires solving for some abstract syntax tree such that for all possible valuations for the variables, the specification holds.

```
pred synth[root: IntNode] {
  all eval: Node -> (Int+Bool) |                          (A.1)
    semantics[eval] implies spec[root, eval] }


run synth for 4
```

We present the results of our evaluation, including a performance comparison between Alloy* and existing program synthesizers, in Sect. 9.2.

7.3 Bitvector programs

Figure 9 contains an example of a bitvector-based SyGuS benchmark. This particular specification calls for a parity checking circuit using AND and NOT gates. The correct solution is specified using XOR gates in the function `parity`, and the grammar allows the synthesizer to use AND gates, NOT gates, and the four input variables.

We can define the `parity` function in Alloy* using the built-in definitions of bits, XOR, and NOT:

[2] For linear arithmetics and Boolean operators listed in Fig. 8, our evaluation procedure employs corresponding built-in operators in Alloy. Since they have the standard semantics, we omit their definitions.

```
fun parity[a, b, c, d: Bit]: Bit {
  Xor[Not[Xor[a, b]],
      Not[Xor[c, d]]]
}
```

Then, we can define AST nodes corresponding to the grammar defined by the synthesis problem and encode the specification of the problem, which says that the result of evaluating the synthesized program should be equal to the result of calling the `parity` function:

```
abstract sig Node {}
abstract sig BoolNode extends Node {}
abstract sig IntNode  extends Node {}
abstract sig BoolVar extends BoolNode {}

one sig A, B, C, D extends BoolVar {}

pred spec[root: Node, eval: Node -> (Int + Bit)] {
  let a = eval[A], b = eval[B], c = eval[C], d = eval[D] |
    parity[a, b, c, d] = eval[root]
}
```

Finally, we define the semantics of the target language and write the synthesis command. This semantics actually supports the target languages of all the bitvector benchmarks that are part of SyGuS ; for benchmarks with more restricted grammars, we simply craft a command to exclude AST nodes of the missing types.

```
pred semantics[eval: Node -> (Int + Bit)] {
  all n: IntNode   | one eval[n] and eval[n] in Int
  all n: BoolNode  | one eval[n] and eval[n] in Bit

  all n: ITE {
    eval[n.condition] = BitTrue implies
      eval[n.then] = eval[n] else eval[n.elsen] = eval[n]
  }
  all n: Equals    | eval[n.left] = eval[n.right] implies
                       eval[n] = BitTrue else eval[n]
                       = BitFalse
  all n: GTE       | eval[n.left] >= eval[n.right] implies
                       eval[n] = BitTrue else eval[n]
                       = BitFalse
  all n: LTE       | eval[n.left] <= eval[n.right] implies
                       eval[n] = BitTrue else eval[n]
                       = BitFalse
  all n: GT        | eval[n.left] > eval[n.right] implies
                       eval[n] = BitTrue else eval[n]
                       = BitFalse
  all n: LT        | eval[n.left] < eval[n.right] implies
                       eval[n] = BitTrue else eval[n] =
                       BitFalse

  all n: And       | eval[n] = And[eval[n.left],
                         eval[n.right]]
  all n: Nand      | eval[n] = Nand[eval[n.left],
                          eval[n.right]]
  all n: Or        | eval[n] = Or [eval[n.left],
                        eval[n.right]]
  all n: Nor       | eval[n] = Nor[eval[n.left],
                         eval[n.right]]
  all n: Xor       | eval[n] = Xor[eval[n.left],
                         eval[n.right]]
  all n: Not       | eval[n] = Not[eval[n.arg]]

  all n: BvShl     | eval[n] = bvshl[eval[n.left],
                          eval[n.right]]
  all n: BvShr     | eval[n] = bvshr[eval[n.left],
                          eval[n.right]]
  all n: BvSha     | eval[n] = bvsha[eval[n.left],
                           eval[n.right]]
  all n: BvAnd     | eval[n] = bvand[eval[n.left],
                          eval[n.right]]
  all n: BvOr      | eval[n] = bvor[eval[n.left],
                          eval[n.right]]
  all n: BvXor     | eval[n] = bvxor[eval[n.left],
                          eval[n.right]]
  all n: BvNot     | eval[n] = bvnot[eval[n.arg]]
  all n: BvNeg     | eval[n] = bvneg[eval[n.arg]]
  all n: BvAdd     | eval[n] = bvadd[eval[n.left],
                           eval[n.right]]
  all n: BvMul     | eval[n] = bvmul[eval[n.left],
                          eval[n.right]]
  all n: BvDiv     | eval[n] = bvdiv[eval[n.left],
                          eval[n.right]]
  all n: BvSub     | eval[n] = bvsub[eval[n.left],
                           eval[n.right]]

  all n: AndInv    | eval[n] = Xor[And[Xor[eval[n.left],
                           n.invLhs],
                                      Xor[eval[n.right],
                                      n.invRhs]], n.invOut]
  all n: OrInv     | eval[n] = Xor[Or[Xor[eval[n.left],
                         n.invLhs],
                                      Xor[eval[n.right],
                                      n.invRhs]], n.invOut]
  all i: IntLit    | eval[i] = i.val
  all b: BoolLit   | eval[b] = b.val
}
```

These semantics use Alloy* 's built-in support for bitvectors to perform the standard logical operations (AND, OR, XOR, NOT, and so on) and also bitvector shift operations. Since some benchmarks also include integer values, these semantics support them as well.

Compared to integer programs, bitvector programs are difficult for Alloy* to synthesize. The original benchmark, which calls for a solution in terms of AND and NOT gates, cannot be solved by Alloy* in more than 12 h. A simplified version of the benchmark, which allows NAND gates (and only NAND gates) can be encoded this way (using the “when” syntax introduced in Sect. 8.1):

```
  run {
    all envB: BoolVar -> one Bit {
      some eval: IntNode->Int + BoolNode->Bit when {
        envB in eval
        semantics[eval]
      }{
        spec[root, eval]
      }
    }
  } for 0 but -1..0 Int, exactly 15 AndInv
```

This command finishes in roughly 20 s. The other SyGuS bitvector benchmarks perform similarly: when restricted in certain ways, Alloy* can find a solution; in general, however, Alloy* does not seem well suited to solving bitvector synthesis problems.

7.4 Hacker's Delight

The book *Hacker's Delight* contains clever bit-twiddling programs that can perform common bitvector operations more efficiently than the naive solution. The SyGuS benchmarks encode 20 of these programs. The input specification is the naive version, and then the synthesizer is expected to produce the optimization.

We have encoded these problems in Alloy* . Since they are bitvector-based, they are not especially well suited to Alloy* 's logic, but because the programs are relatively short compared to the larger set of bitvector benchmarks, Alloy* is able to solve many of them.

The ability to encode these benchmarks—even if Alloy* is not yet capable of solving them all efficiently—is an encouraging sign that Alloy* 's approach to higher-order constraint solving is general enough to express common synthesis tasks. This capability suggests that with targeted future optimizations of Alloy* 's CEGIS solver, it could become a useful research tool for exploring program synthesis problems.

8 Optimizations

Originally motivated by the formalization of the synthesis problem (as presented in Sect. 7), we designed and implemented two general purpose optimizations for Alloy* .

8.1 Quantifier domain constraints

As defined in Listing A.1, the `synth` predicate, although logically sound, suffers from serious performance issues. The main issue is the effect on the CEGIS loop of the implication inside the universal quantifier. To trivially satisfy the implication, the candidate search step can simply return an instance for which the semantics does not hold. Furthermore, adding the encoding of the counterexample refuting the previous instance is not going to constrain the next search step to find a program and a valuation for which the spec holds. This cycle can go on for unacceptably many iterations.

To overcome this problem, we can add syntax to identify the constraints that should be treated as part of the bounds of a quantification. The `synth` predicate, e.g., now becomes

```
pred synth[root: IntNode] {
  all eval: Node -> (Int + Bool) when semantics[eval] |
    spec[root, eval] }
```

The existing first-order semantics of Alloy is unaffected, i.e.,

```
all  x when D[x] | P[x]  ⟺  all x |
D[x] implies P[x]                                              (A.2)
some x when D[x] | P[x]  ⟺  some x | D[x] and P[x]
```

The rule for pushing negation through quantifiers (used by the converter to NNF) becomes:

```
not (all  x when D[x] | P[x])  ⟺  some x when D[x]
                | not P[x]
not (some x when D[x] | P[x])  ⟺  all  x when D[x]
                | not P[x]
```

(which is consistent with classical logic).

The formalization of the Alloy* semantics needs only a minimal change. The change is made on how the existential counterpart of a universal quantifier is obtained through negation—only by flipping the quantifier, and keeping the domain and the body the same (line 5, Fig. 6). Consequently, the candidate condition always searches for an instance satisfying both the domain and the body constraint (i.e., both the semantics and the spec). The same is also true for counterexamples obtained in the verification step. The only actual change to be made to the formalization is expanding *q.body* in line 38 according to the rules in Listing A.2.

Going back to the synthesis example, even after rewriting the `synth` predicate, unnecessary overhead is still incurred by quantifying over valuations for all the nodes, instead of valuations for just the input variables. Consequently, the counterexamples produced in the CEGIS loop do not guide the search as effectively. This observation leads us to our final formulation of the `synth` predicate:

```
pred synth[root: IntNode] {
  all env: Var -> Int | some eval: Node -> (Int+Bool)
    when env in eval && semantics[eval] |
spec[root, eval] }
```
(A.3)

Despite using nested higher-order quantifiers, it is the most efficient: the inner quantifier (over `eval`) always takes exactly one iteration (to either prove or disprove the current `env`), because for a fixed `env`, `eval` is uniquely determined.

8.2 Strictly first-order increments

We already pointed out the importance of implementing the induction step (line 30, Fig. 7) using incremental SAT solving. A problem, however, arises when the encoding of the counterexample (as defined in line 28) is not a first-order formula—since not directly translatable to SAT, it cannot be incrementally added to the existing SAT translation of the candidate search condition. In such cases, the semantics in Fig. 7 demands that f_{cex} be solved from scratch, losing any benefits from previously learned SAT clauses. This problem occurs in our final formulation of the `synth` predicate (Listing A.3), due to the nested higher-order quantifiers.

To address this issue, we relax the semantics of the induction step by changing universal quantifiers in f_{cex} to existential ones as necessary to make f_{cex} first-order. This change means that f_{cex} can always be added as an increment to the current SAT translation of the candidate condition. The trade-off involved here is that this new encoding of the counterexample is potentially not as strong, and therefore may lead to more CEGIS iterations before a resolution is reached. For that reason, Alloy* accepts a configuration parameter (accessible via the "Options" menu), offering both strategies. In Sect. 9 we provide experimental data showing that for all of our synthesis examples, the strictly first-order increments yielded better performance.

9 Evaluation

9.1 Micro benchmarks

To assess scalability, we measure the time Alloy* takes to solve 4 classical, higher-order graph problems for graphs of varying size: `max clique`, `max cut`, `max independent set`,

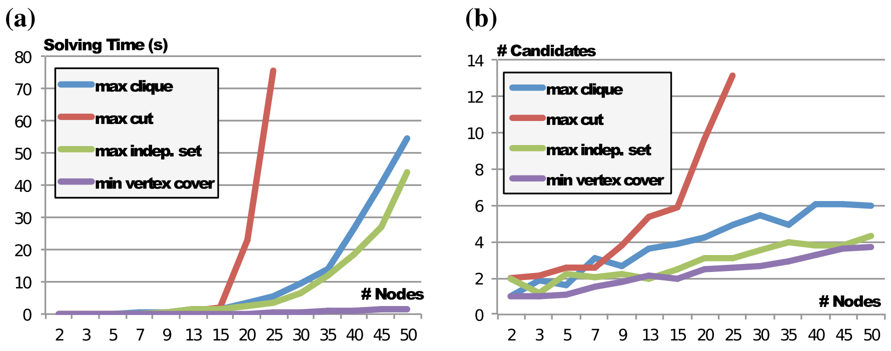

Fig. 10 Average (over 5 different edge densities). **a** Solving times, and **b** number of explored candidates for the graph algorithms

and `min vertex cover`. We used the Erdős–Rényi model [13] to randomly generate graphs to serve as inputs to the benchmark problems. We generated graphs of sizes ranging from 2 to 50. To cover a wide range of edge densities, for each size we generated 5 graphs, using different threshold values (T) from the set $\{0.1, 0.3, 0.5, 0.7, 0.9\}$, where the probability of inserting an edge between a pair of nodes was set to $1-\text{T}$. We specified [32, Fig. 9] the graph problems in Alloy and used Alloy* to solve them. To ensure correctness of the results returned by Alloy*, we made sure they matched those of known imperative algorithms for the same problems.[3] The timeout for each Alloy* run was set to 100 s.

Figure 10 plots two graphs: (a) the average solving time across graphs size, and (b) the average number of explored candidates per problem.

The performance results show that for all problems but `max cut`, Alloy* was able to handle graphs of sizes up to 50 nodes in less than a minute (`max cut` started to time out at around 25 nodes). Our original goal for these benchmarks was to be able to solve graphs with 10–15 nodes, and claim that Alloy* can be effectively used for teaching, specification animation, and small scope bounded verification, all within the Alloy Analyzer IDE (which is one of the most common uses of the Alloy technology). These results, however, suggest that executing higher-order specifications may be feasible even for declarative programming (where a constraint solver is embedded in a programming language, e.g., [31,33,54]), which is very encouraging.

The average number of explored candidates (Fig. 10b) confirms the effectiveness of the CEGIS induction step at pruning the remainder of the search space. Even for graphs of size 50, in most cases the average number of explored candidates is around 6; the exception is, again, `max cut`, where this curve is closer to being linear. We further analyze how the total time is split over individual candidates in Sect. 9.4.

Figure 11 shows average solving times over different probability thresholds used to generate graphs for the micro benchmark experiments. Lower the threshold (T), denser a graph is (similarly, higher T values lead to sparser graphs). In general, the solving time tends to be greater for denser graphs, since Alloy* needs to explore a larger number of edges than in sparser graphs. This trend is especially evident in the max cut problem (Fig. 11b), where the solving time increases drastically for $\text{T} = 0.1$ until Alloy* times out at graphs of size 20.

[3] For `max clique` and `max independent set`, we used the Bron–Kerbosch heuristic algorithm; for the other two, no good heuristic algorithm is known, and so we implemented enumerative search. In both cases, we used Java.

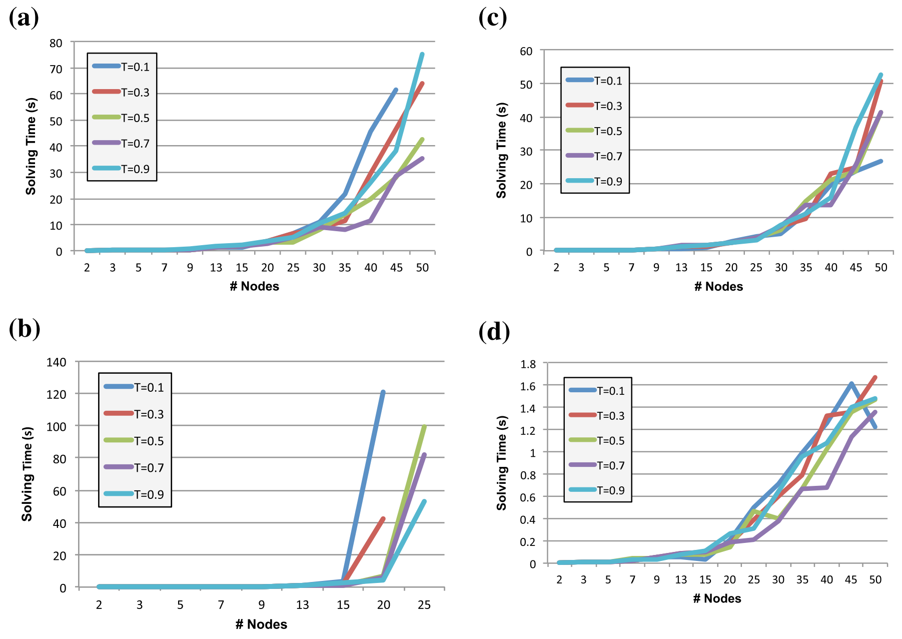

Fig. 11 Average times over thresholds for graph algorithms. **a** max clique, **b** max cut, **c** max independent set, **d** min vertex

On the other hand, sparsity does not necessarily lead to faster performance. For example, in the max clique and max independent set, note how Alloy* takes longer on graphs with T = 0.9 (Fig. 11a, b) than on those with lower threshold as the graph size increases. We believe that this is because large, sparser connected graphs tend to permit fewer cliques (and independent sets) than more densely connected graphs. As the figures show, Alloy* tends to perform best on graphs that lie in the middle of the spectrum (T = 0.5 or T = 0.7).

9.2 Program synthesis

To demonstrate the expressive power of Alloy* , we encoded 123 out of 173 benchmarks available in the SyGuS 2014 GitHub repository [52]—all except those from the "icfp-problems" folder. We skipped the 50 "icfp" benchmarks because they all use large (64-bit) bit vectors, which are not supported by Alloy; none of them could be solved anyway by any of the solvers that entered the SyGuS 2014 competition. All of our encoded benchmarks come with the official Alloy* distribution [1] and are accessible from the main menu: `File→Open Sample Models`, then choose `hol/sygus`.

Some SyGuS benchmarks require synthesizing multiple functions at once, and some require multiple applications of the synthesized functions. All of these cases can be handled with small modifications to our encoding presented in Sect. 7. For example, to synthesize multiple functions at once, we add additional root node pointers to the signature of the `synth` predicate; to allow for multiple applications of the same function we modify the `synth` predicate to compute multiple `eval` relations (one for each application). Finally, to support SyGuS benchmarks involving bit vectors, we exposed the existing Kodkod bitwise operators over integers at the Alloy level, and also made small changes to the Alloy grammar

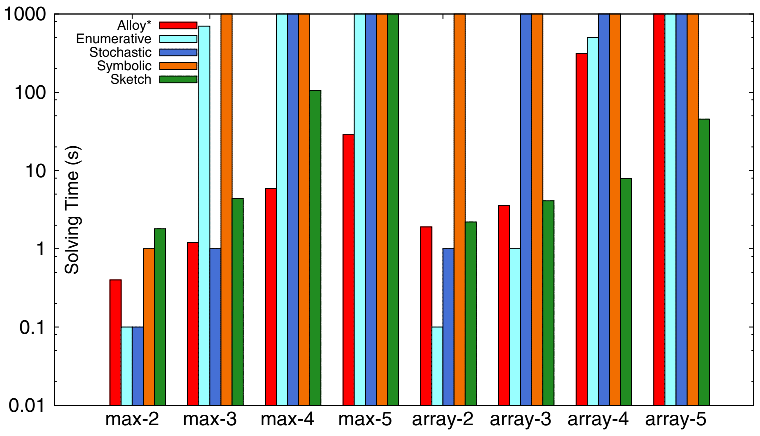

Fig. 12 Comparison between Alloy* and reference solvers

to allow for a more flexible integer scope specification (so that integer atoms can be specified independently of integer bitwidth).

To evaluate Alloy* 's performance, we ran the same benchmarks on the same computer using the three reference solvers. We limit the benchmarks to those found in the "integer-benchmarks" folder because they: (1) do not use bit vectors (which Alloy does not support natively), and (2) allow for the scope to be increased arbitrarily, and thus are suitable for performance testing. Our test machine had an Intel dual-core CPU, 4GB of RAM, and ran Ubuntu GNU/Linux and Sun Java 1.6. We set Alloy* 's solver to be MiniSAT.

Figure 12 compares the performance of Alloy* against the three SyGuS reference solvers and Sketch [50], a highly-optimized, state-of-the-art program synthesizer. According to these results, Alloy* scales better than the three reference solvers, and is even competitive with Sketch. On the `array-search` benchmarks, Sketch outperforms Alloy* for larger problem sizes, but on the `max` benchmarks, the opposite is true. Both solvers scale far more predictably than the reference solvers, but Alloy* has the additional advantage, due to its generality, of a flexible encoding of the target language's semantics, while Sketch relies on the semantics of the benchmark problems being the same as its own.

Other researchers have reported [21] that benefits can be gained by specifying tighter bounds on the abstract syntax tree nodes considered by the solver. Table 1 confirms that for `max` and `array` significant gains can be realized by tightening the bounds. In the case of `max`, tighter bounds allow Alloy* to improve from solving the 5-argument version of the problem to solving the 7-argument version. For these experiments, Scope 1 specifies the exact number of each AST node required; Scope 2 specifies exactly which types of AST nodes are necessary; and Scope 3 specifies only how many total nodes are needed. Other solvers also ask the user to bound the analysis—Sketch, for example, requires both an integer and recursion depth bound—but do not provide the same fine-grained control over the bounds as Alloy* . For the comparison results in Fig. 12, we set the most permissive scope (Scope 3) for Alloy* .

These results show that Alloy* , in certain cases, not only scales better than the reference solvers, but can also be competitive with state-of-the-art solvers based on years of optimization. Such cases are typically those that require a structurally complex program AST (adhering to complex relational invariants) be discovered from a large search space. When the

Table 1 Performance on synthesis benchmarks

Problem	Scope 1		Scope 2		Scope 3	
	Steps	Time (s)	Steps	Time (s)	Steps	Time (s)
max-2	3	0.3	3	0.4	3	0.4
max-3	6	0.9	7	0.9	8	1.2
max-4	8	1.5	8	3.0	15	5.9
max-5	25	4.2	23	36.3	19	28.6
max-6	29	16.3	n/a	t/o	n/a	t/o
max-7	34	256.5	n/a	t/o	n/a	t/o
array-2	8	1.6	8	2.4	8	1.9
array-3	13	4.0	9	8.1	7	3.6
array-4	15	16.1	11	98.0	15	310.5
array-5	19	386.9	n/a	t/o	n/a	t/o

size of the synthesized program is small, however, the results of the SyGuS 2014 competition show [4] that non-constraint based techniques, such as enumerative and stochastic search, tend to be more efficient.

Finally, Alloy* requires only the simple model presented here—which is easier to produce than even the most naive purpose-built solver. Due to its generality, Alloy* is also, in some respects, a more flexible program synthesis tool—it makes it easy, for example, to experiment with the semantics of the target language, while solvers like Sketch have their semantics hard-coded.

9.3 Benefits of the Alloy* optimizations

We used the program synthesis benchmarks (with the tightest, best performing scope), and the bounded verification of Turán's theorem from Sect. 2 to evaluate the optimizations introduced in Sect. 8 by running the benchmarks with and without them. The baseline was a specification written without using domain constraints and an analysis without first-order increments. The next two rows correspond to (1) adding exactly one optimization (rewriting the specification to use the domain constraints for the synthesis benchmarks, and using first-order increments for Turán's theorem[4], and (2) adding both optimizations. Table 2 shows the results. Across the board, writing domain constraints makes a huge difference; using first-order increments often decreases solving time significantly, and, in the synthesis cases, causes the solver to scale to slightly larger scopes.

9.4 Distribution of solving time over individual candidates

In Fig. 13 we take the three hardest benchmarks (`max-7`, `array-search-5`, and `turan-10`, with tightest bounds and both optimizations applied) and show how the total solving time is distributed over individual candidates (which are sequentially explored by Alloy*). Furthermore, for each candidate we show the percentage of time that went into each of the three CEGIS phases (search, verification, induction). Instead of trying to draw strong conclusions, the main idea behind this experiment is to illustrate the spectrum of possible behaviors the Alloy* solving strategy may exhibit at runtime.

[4] If we first rewrote Turán's theorem to use domain constraints, there would be no nested CEGIS loops left, so increments would be first-order even without the other optimization.

Table 2 Performance of Alloy* (in seconds) with and without optimizations

	Base	Base + 1 optimization	Base + both optimizations
`max2`	0.4	0.4	0.
`max3`	7.6	1.0	0.9
`max4`	t/o	4.7	1.5
`max5`	t/o	10.3	4.2
`max6`	t/o	136.4	16.3
`max7`	t/o	t/o	163.6
`max8`	t/o	t/o	987.3
`arr2`	140	2.9	1.6
`arr3`	t/o	6.3	4.0
`arr4`	t/o	76.9	16.1
`arr5`	t/o	t/o	485.6
`tur5`	3.5	1.1	0.5
`tur6`	12.8	5.1	2.1
`tur7`	235	43	3.8
`tur8`	t/o	t/o	15
`tur9`	t/o	t/o	45
`tur10`	t/o	t/o	168

The problems seen early in the search tend to be easy in all cases. The results reveal that later on, however, at least three different scenarios can happen. In the case of `max-7`, the SAT solver is faced with a number of approximately equally hard problems, which is where the majority of time is spent. In `array-search-5`, in contrast, saturation is reached very quickly, and the most time is spent solving very few hard problems. For `turan-10`, the opposite is true: the time spent solving a large number of very easy problems dominates the total time.

9.5 Discussion

Alloy* can solve arbitrary higher-order formulas and is sound and complete for the given bounds. The current Alloy language, however, is not necessarily the most intuitive way to express certain higher-order properties. Its semantics for encoding a candidate solution as a partial instance during the verification step, on the other hand, might not always be what the user wants.

Alloy users are used to using sigs and fields for representing relations (which are always implicitly existentially quantified over), but to express certain higher-order properties, explicit quantification is necessary. For example, to verify Turán's theorem from Fig. 1, for most Alloy users it would be more natural to represent edges as a field of sig `Node` of type `Node`, and then, in the `Turan` predicate, somehow say "for all possible graph structures, assert the property of Turán's theorem". But in Alloy, **`all`** `g: Graph` already has a very different semantics than the one needed for this example [20,24]; in Alloy*, we wanted to preserve backward compatibility, as well as not introduce any significant changes to the language, so the user has to explicitly quantify over all possible `edges` relations (line 14, Fig. 1) to achieve the desired behavior.

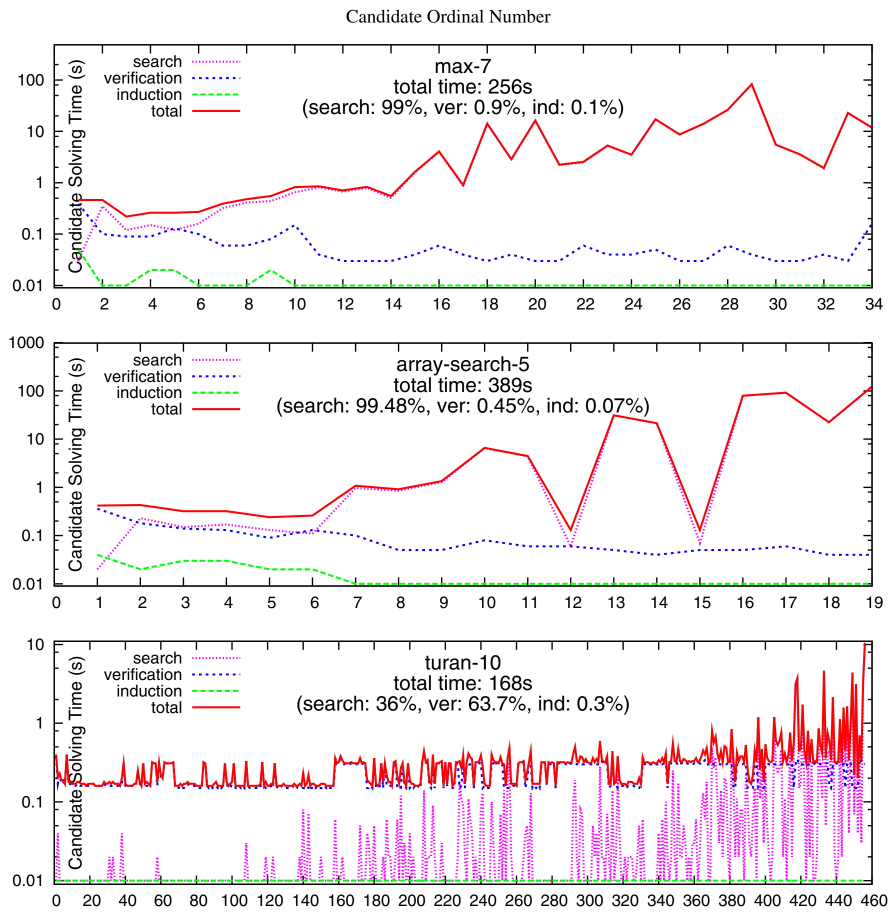

Fig. 13 Distribution of total solving time over individual (sequentially explored) candidates for the three hardest benchmarks. Each candidate time is further split into times for each CEGIS phase

Several higher-order idioms can be solved more efficiently than by applying CEGIS. Minimization/maximization are probably the most obvious such idioms. To find a maximal clique in a graph, rather than finding a clique and asserting that there is no other clique that is larger than it, it is more efficient to start with one (arbitrary) clique, and then iteratively keep searching for a larger clique, until one cannot be found. We believe that many such special higher-order idioms can be implemented in a general-purpose solver, but would require adding special new quantifiers, which would conflict with our initial decision to retain the existing Alloy language.

Regarding Alloy*'s semantics for encoding a candidate solution as a partial instance during the verification step, because it has no domain-specific knowledge of the problem being solved, to proceed from candidate search to verification, Alloy* always encodes all relations except the higher-order quantification variable as a partial instance. As said, Alloy users, mostly for convenience reasons, write sigs to represent relations that are implicitly existentially quantified over, so it is possible that it is not always desirable to include all of them in the partial instance.

10 Related work

Constraint solvers SMT solvers, by definition, find satisfying interpretations of first-order formulas over unbounded domains. In that context, only quantifier-free fragments are decidable. Despite that, many solvers (e.g., Z3 [35]) support certain forms of quantification by implementing an efficient matching heuristic based on patterns provided by the user [11]. Certain non-standard extension allow quantification over functions and relations for the purpose of checking properties over recursive predicates [7], as well as model-based quantifier instantiation [17]. In the general case, however, this approach often leads to "unknown" being return as the result. Tools that build on top of SMT raise the level of abstraction of the input language and provide quantification patterns that work more reliably in practice (e.g., [6,27]), but are limited to first-order forms.

SAT solvers, on the other hand, are designed to work with bounded domains. Tools built on top may support logics richer than propositional formulas, including higher-order quantifiers. One such tool is Kodkod [53]. At the language level, it allows quantification over arbitrary relations, but the analysis engine, however, is not capable of handling those that are higher-order. Rosette [54] builds on top of Kodkod a suite of tools for embedding constraint solvers into programs for a variety of purposes, including synthesis. It implements a synthesis algorithm internally, so at the user level, unlike Alloy*, this approach enables only one predetermined form of synthesis, namely, finding an instantiation of a user-provided grammar that satisfies a specified property.

A higher-order specification with bounded domains can be encoded as a first-order fragment by "grounding out" each higher-order quantifier (i.e., enumerating all possible tuples of a relation, and having the quantified variable take on a value from this finite tuple set). Therefore, in principle, any first-order constraint solver can be used to solve this fragment, although this approach is likely to run into scalability problems as the number of tuples tends to grow rapidly. In fact, this feature was included in an older version of Alloy, but was found to be inefficient and discontinued in subsequent releases.

Synthesizers State-of-the-art synthesizers today are mainly purpose-built. Domains of application include program synthesis (e.g., [25,28,49–51]), automatic grading of programming assignments [48], synthesis of data manipulation regular expressions [18], and so on, all using different ways for the user to specify the property to be satisfied. Each such specialized synthesizer, however, would be hard to apply in a domain different than its own. A recent effort has been made to establish a standardized format for program synthesis problems [3]; this format is syntax-guided, similar to that of Rosette, and thus less general than the language (arbitrary predicate logic over relations) offered by Alloy*.

Interactive Theorem Provers Nitpick [8] and its successor, Nunchaku [43], are tools for generating counterexamples for higher-order specifications in Isabelle/HOL [41]. Like Alloy*, Nitpick translates an input specification into a Kodkod formula to leverage the latter's model finding facility, but their translation scheme is significantly different from ours. Given a higher-order quantifier over a function, Nitpick constructs a relation whose tuples range over all possible values of the function (essentially grounding out the quantifier), thereby achieving a translation to a FOL formula that can be solved with a single call to the model finder. In comparison, our strategy involves iteratively exploring the space of candidate functions by making multiple calls to Kodkod.

Program Verifiers Program verifiers benefit directly from expressive specification languages equipped with more powerful analysis tools. In recent years, many efforts have been made towards automatically verifying programs in higher-order languages. Liquid types [44] and

HMC [22] respectively adapt known techniques for type inference and abstract interpretation for this task. Bjørner et al. examine direct encodings into Horn clauses, concluding that current SMT solvers are effective at solving clauses over integers, reals, and arrays, but not necessarily over algebraic datatypes. Dafny [27] is the first SMT-based verifier to provide language-level mechanisms specifically for automating proofs by co-induction [29].

Executable Specifications Many research projects explore the idea of extending a programming language with symbolic constraint-solving features (e.g., [23,33,46,54,57]). Limited by the underlying constraint solvers, none of these tools can execute a higher-order constraint. In contrast, we used αRby [31] (our most recent take on this idea where we embed the entire Alloy language directly into Ruby), equipped with Alloy* as its engine, to run all our graph experiments (where αRby automatically translated input partial instances from concrete graphs, as well as solutions returned from Alloy back to Ruby objects), demonstrating how a higher-order constraint solver can be practical in this area.

Existing Alloy Tools Certain tools built using Alloy already provide means for achieving tasks similar to those we used as Alloy* examples. Aluminum [40], for instance, extends the Alloy Analyzer with a facility for minimizing solutions. It does so by using the low-level Kodkod API to selectively remove tuples from the resulting tuple set. To achieve minimality, by contrast, we used a purely declarative constraint to assert that there is no other satisfying solution with fewer tuples. In another instance, Montaghami et al. recognize a particular Alloy idiom involving a universal quantifier, and change the semantics to force all atoms in the domain of that quantifier to be always included in the analysis [34], effectively supporting only a small class of higher-order models.

Rayside et al. used the Alloy Analyzer to synthesize iterators from abstraction functions [42], as well as complex (non-pure) AVL tree operations from abstract specifications [26]. In both cases, they target very specific categories of programs, and their approach is based on insights that hold only for those particular categories.

11 Conclusion

Software analysis and synthesis tools have typically progressed by the discovery of new algorithmic methods in specialized contexts, and then their subsequent generalization as solutions to more abstract mathematical problems. This trend—evident in the history of dataflow analysis, symbolic evaluation, abstract interpretation, model checking, and constraint solving—brings many benefits. First, the translation of a class of problems into a single, abstract and general formulation allows researchers to focus more sharply, resulting in deeper understanding, cleaner APIs and more efficient algorithms. Second, generalization across multiple domains allows insights to be exploited more widely, and reduces the cost of tool infrastructure through sharing of complex analytic components. And third, the identification of a new, reusable tool encourages discovery of new applications.

In this paper, we have argued that the time is ripe to view higher-order constraint solving in this context, and we have proposed a generalization of a variety of algorithms that we believe suggests that the productive path taken by first-order solving might be taken by higher-order solving too.

Acknowledgements This material is based upon work partially supported by the National Science Foundation under Grant Nos. CCF-1138967, CRI-0707612, and CCF-1438982.

References

1. Alloy* home page. http://alloy.mit.edu/alloy/hola
2. Aigner M, Ziegler GM (2001) Turán's graph theorem. In: Proofs from THE BOOK. Springer, Berlin, pp 183–187
3. Alur R, Bodík R, Juniwal G, Martin MMK, Raghothaman M, Seshia SA, Singh R, Solar-Lezama A, Torlak E, Udupa A (2013) Syntax-guided synthesis. In: FMCAD. IEEE, pp 1–17
4. Alur R, Bodík R, Juniwal G, Martin MMK, Raghothaman M, Seshia SA, Singh R, Solar-Lezama A, Torlak E, Udupa A (2014) Syntax-guided synthesis competition report (2014). http://sygus.seas.upenn.edu/files/sygus_extended
5. Alvaro P, Hutchinson A, Conway N, Marczak WR, Hellerstein JM (2012) Bloomunit: declarative testing for distributed programs. In: Proceedings of the fifth international workshop on testing database systems, DBTest 2012, Scottsdale, AZ, USA, 21 May 2012, p 1
6. Barnett M, Chang BYE, DeLine R, Jacobs B, Leino KRM (2006) Boogie: a modular reusable verifier for object-oriented programs. In: FMCO 2005, LNCS, vol 4111. Springer, pp 364–387
7. Bjørner N, McMillan K, Rybalchenko A (2012) Program verification as satisfiability modulo theories. In: SMT workshop at IJCAR, vol 20
8. Blanchette JC, Nipkow T (2010) Nitpick: a counterexample generator for higher-order logic based on a relational model finder. In: Interactive theorem proving, first international conference, ITP 2010, Proceedings, Edinburgh, UK, 11–14 July 2010, pp 131–146
9. Boyatt R, Sinclair J (2008) Experiences of teaching a lightweight formal method. Electronic Notes in Theoretical Computer Science, pp 71–80
10. de Andrade FR, Faria JP, Lopes A, Paiva ACR (2012) Specification-driven unit test generation for java generic classes. In: Integrated formal methods—9th international conference, IFM 2012, Proceedings, Pisa, Italy, 18–21 June 2012, pp 296–311
11. De Moura L, Bjørner N (2007) Efficient e-matching for SMT solvers. In: Automated deduction—CADE-21. Springer, pp 183–198
12. Dennis G (2009) A relational framework for bounded program verification. PhD thesis, MIT
13. Erdos P, Renyi A (1960) On the evolution of random graphs. Math Inst Hung Acad Sci 5:17–61
14. Ferreira JF, Mendes A, Cunha A, Baquero C, Silva P, Barbosa LS, Oliveira JN (2011) Logic training through algorithmic problem solving. In: Tools for teaching logic. Springer, pp 62–69
15. Fisler K, Krishnamurthi S, Meyerovich LA, Tschantz MC (2005) Verification and change-impact analysis of access-control policies. In: Proceedings of the 27th ICSE. ACM, pp 196–205
16. Galeotti JP, Rosner N, López Pombo CG, Frias MF (2010) Analysis of invariants for efficient bounded verification. In: ISSTA. ACM, pp 25–36
17. Ge Y, De Moura L (2009) Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Computer aided verification. Springer, pp 306–320
18. Gulwani S, Harris WR, Singh R (2012) Spreadsheet data manipulation using examples. Commun ACM 55(8):97–105
19. Hughes G, Bultan T (2008) Automated verification of access control policies using a sat solver. STTT 10(6):503–520
20. Jackson D (2006) Software abstractions: logic, language, and analysis. MIT Press, Cambridge
21. Jha S, Gulwani S, Seshia SA, Tiwari A (2010) Oracle-guided component-based program synthesis. In: ICSE, ICSE'10. ACM, New York, pp 215–224
22. Jhala R, Majumdar R, Rybalchenko A (2011) HMC: verifying functional programs using abstract interpreters. In: Computer aided verification. Springer, pp 470–485
23. Köksal AS, Kuncak V, Suter P (2010) Constraints as control. In: ACM SIGPLAN Notices
24. Kuncak V, Jackson D (2005) Relational analysis of algebraic datatypes. In: ACM SIGSOFT software engineering notes, vol 30. ACM, pp 207–216
25. Kuncak V, Mayer M, Piskac R, Suter P (2010) Comfusy: a tool for complete functional synthesis. In: CAV, pp 430–433
26. Kurilova D, Rayside D (2013) On the simplicity of synthesizing linked data structure operations. In: Proceedings of the 12th international conference on generative programming: concepts and experiences. ACM, pp 155–158
27. Leino KRM (2010) Dafny: an automatic program verifier for functional correctness. In: LPAR-16, LNCS, vol 6355. Springer, pp 348–370
28. Leino KRM, Milicevic A (2012) Program extrapolation with Jennisys. In: Proceedings of the international conference on object oriented programming systems languages and applications, pp 411–430
29. Leino KRM, Moskal M (2013) Co-induction simply: automatic co-inductive proofs in a program verifier. Technical report, MSR-TR-2013-49, Microsoft Research

30. Marinov D, Khurshid S (2001) Testera: a novel framework for automated testing of java programs. In: Automated software engineering. IEEE, pp 22–31
31. Milicevic A, Efrati I, Jackson D (2014) αRby—an embedding of alloy in ruby. In: Abstract state machines, alloy, B, TLA, VDM, and Z. Springer, pp 56–71
32. Milicevic A, Near JP, Kang E, Jackson D (2014) Alloy*: a higher-order relational constraint solver. Technical report, MIT-CSAIL-TR-2014-018, Massachusetts Institute of Technology. http://hdl.handle.net/1721.1/89157
33. Milicevic A, Rayside D, Yessenov K, Jackson D (2011) Unifying execution of imperative and declarative code. In: ICSE, pp 511–520
34. Montaghami V, Rayside D (2014) Staged evaluation of partial instances in a relational model finder. In: Abstract state machines, alloy, B, TLA, VDM, and Z. Springer, pp 318–323
35. de Moura L, Bjørner N (2008) Z3: an efficient SMT solver. In: TACAS 2008, LNCS, vol 4963. Springer, pp 337–340
36. Nakajima S (2014) Using alloy in introductory courses of formal methods. In: Structured object-oriented formal language and method—4th international workshop, SOFL+MSVL 2014, Revised selected papers, Luxembourg, 6 Nov 2014, pp 97–110
37. Near JP, Jackson D (2012) Rubicon: bounded verification of web applications. In: 20th ACM SIGSOFT symposium on the foundations of software engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA, 11–16 Nov 2012, p 60
38. Near JP, Jackson D (2016) Finding security bugs in web applications using a catalog of access control patterns. In: Proceedings of the 38th international conference on software engineering, ICSE 2016, Austin, TX, USA, 14–22 May 2016, pp 947–958
39. Nelson T, Barratt C, Dougherty DJ, Fisler K, Krishnamurthi S (2010) The Margrave tool for firewall analysis. In: Proceedings of the international conference on large installation system administration, pp 1–8
40. Nelson T, Saghafi S, Dougherty DJ, Fisler K, Krishnamurthi S (2013) Aluminum: principled scenario exploration through minimality. In: ICSE. IEEE Press, pp 232–241
41. Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL—a proof assistant for higher-order logic, Lecture notes in computer science, vol 2283. Springer
42. Rayside D, Montaghami V, Leung F, Yuen A, Xu K, Jackson D (2012) Synthesizing iterators from abstraction functions. In: Proceedings of the international conference on generative programming and component engineering, pp 31–40
43. Reynolds A, Blanchette JC, Cruanes S, Tinelli C (2016) Model finding for recursive functions in SMT. In: Automated reasoning—8th international joint conference, IJCAR 2016, Proceedings, Coimbra, Portugal, 27 June–2 July 2016, pp 133–151
44. Rondon PM, Kawaguci M, Jhala R (2008) Liquid types. In: ACM SIGPLAN Notices, vol 43. ACM, pp 159–169
45. Rosner N, Galeotti J, Bermúdez S, Blas GM, De Rosso SP, Pizzagalli L, Zemín L, Frias MF (2013) Parallel bounded analysis in code with rich invariants by refinement of field bounds. In: ISSTA. ACM, pp 23–33
46. Samimi H, Aung ED, Millstein TD (2010) Falling back on executable specifications. In: ECOOP, pp 552–576
47. Schaad A, Moffett JD (2002) A lightweight approach to specification and analysis of role-based access control extensions. In: SACMAT, pp 13–22
48. Singh R, Gulwani S, Solar-Lezama A (2013) Automated feedback generation for introductory programming assignments. In: Proceedings of the 34th PLDI. ACM, pp 15–26
49. Singh R, Solar-Lezama A (2011) Synthesizing data structure manipulations from storyboards. In: Proceedings of the symposium on the foundations of software engineering, pp 289–299
50. Solar-Lezama A, Tancau L, Bodik R, Seshia S, Saraswat V (2006) Combinatorial sketching for finite programs. In: Proceedings of the international conference on architectural support for programming languages and operating systems, pp 404–415
51. Srivastava S, Gulwani S, Chaudhuri S, Foster JS (2011) Path-based inductive synthesis for program inversion. In: PLDI 2011. ACM, pp 492–503
52. SyGuS github repository. https://github.com/rishabhs/sygus-comp14.git
53. Torlak E (2008) A constraint solver for software engineering: finding models and cores of large relational specifications. PhD thesis, MIT
54. Torlak E, Bodik R (2013) Growing solver-aided languages with rosette. In: Proceedings of the 2013 ACM international symposium on new ideas, new paradigms, and reflections on programming and software. ACM, pp 135–152

55. Torlak E, Jackson D (2007) Kodkod: a relational model finder. In: Tools and algorithms for the construction and analysis of systems. Springer, pp 632–647
56. Vaziri M, Jackson D (2003) Checking properties of heap-manipulating procedures with a constraint solver. In: 9th international conference on tools and algorithms for the construction and analysis of systems, TACAS 2003, held as part of the joint european conferences on theory and practice of software, ETAPS 2003, Proceedings, Warsaw, Poland, 7–11 Apr 2003, pp 505–520
57. Yang J, Yessenov K, Solar-Lezama A (2012) A language for automatically enforcing privacy policies. In: Proceedings of the symposium on principles of programming languages, pp 85–96