Existential Heap Abstraction Entailment is Undecidable

Viktor Kuncak and Martin Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{vkuncak,rinard}@lcs.mit.edu

Abstract. In this paper we study constraints for specifying properties of data structures consisting of linked objects allocated in the heap. Motivated by heap summary graphs in role analysis and shape analysis we introduce the notion of *regular graph constraints*. A regular graph constraint is a graph representing the heap summary; a heap satisfies a constraint if and only if the heap can be homomorphically mapped to the summary. Regular graph constraints form a very simple and natural fragment of the existential monadic second-order logic over graphs.

One of the key problems in a compositional static analysis is proving that procedure preconditions are satisfied at every call site. For role analysis, precondition checking requires determining the validity of implication, i.e., *entailment* of regular graph constraints.

The central result of this paper is the undecidability of regular graph constraint entailment. The *undecidability* of the *entailment* problem is surprising because of the simplicity of regular graph constraints: in particular, the *satisfiability* of regular graph constraints is *decidable*.

Our undecidability result implies that there is no complete algorithm for statically checking procedure preconditions or postconditions, simplifying static analysis results, or checking that given analysis results are correct. While incomplete conservative algorithms for regular graph constraint entailment checking are possible, we argue that heap specification languages should avoid second-order existential quantification in favor of explicitly specifying a criterion for summarizing objects.

Keywords: Shape Analysis, Typestate, Monadic Second-Order Logic, Type Checking, Program Verification, Graph Homomorphism, Post Correspondence Problem

[1] This research was supported in part by DARPA Contract F33615-00-C-1692, NSF Grant CCR00-86154, NSF Grant CCR00-63513, and the Singapore-MIT Alliance.

[2] 10th Annual International Static Analysis Symposium (SAS 2003), Version of January 26, 2004, 8:08am

1 Introduction

Typestate systems. Types capture important properties of objects in the program, reflecting not only the format of stored information but also the set of applicable operations and the intended use of the objects in the program. Types therefore help avoid programming errors and increase the maintainability of the program. In an imperative language, the properties of objects change over time. However, in traditional type systems, the type of the object does not change over the object's lifetime. This property of traditional types therefore limits the set of properties that they can express. It is therefore desirable to develop abstractions that change as the properties of objects change. A *typestate* is a system where types of objects change over time. A simple typestate system was introduced in [34]; more recent examples include [8–11, 14, 21, 33, 36]. Similarly to [13], these typestate systems are a step towards the highly automated static checking of complex properties of objects.

One of the difficulties in specifying properties of objects in the presence of linked data structures is that a property of an object x may depend on properties of objects y that are linked to x in the heap. Some systems allow programmers to identify properties of an object x in terms of the properties of the objects y such that x references y. The idea that important properties of an object x depend on the the number and properties of objects z such that z references x was introduced in the role system [21].

Existential Semantics of Roles. To allow definitions of cyclic structures, in [21, Section 3.3] we have adopted the following semantics: a heap satisfies a set of properties if there *exists* some assignment of predicate names to heap objects such that the given local referencing constraints are satisfied. We call constraints defined in this way *role constraints*. The existential quantification over predicate names can be expressed in existential monadic second-order logic [12]. Role constraints explicitly specify constraints on incoming and outgoing fields of objects as well as inverse reference and acyclicity constraints. Role constraints encode may-reachability properties implicitly, through the reachability between summary nodes.

The Entailment Problem. One of the key problems for a compositional static analysis is checking that the precondition of a procedure is satisfied at every call site. In general, checking a precondition corresponds to verifying the validity of implication (entailment) of heap properties. In [21, Section 6.3.1] we present a *conservative* algorithm for checking the entailment of role constraints. In this paper we study the possibility of the existence of a *complete* sound algorithm for role constraint entailment. We argue that no such algorithm exists: the entailment problem is undecidable.

Regular Graph Constraints. What is interesting about our undecidability result is that the source of undecidability is a particularly weak fragment of role constraints. We call this fragment *regular graph constraints*. Regular graph constraints capture the problem of mutually recursive properties over potentially cyclic graphs, while abstracting from the details of the particular specification language. The only local properties expressible in regular graph constraints are points-to referencing relationships; unlike role constraints, regular graph constraints cannot express sharing, inverse reference or acyclicity properties. Despite this simplicity, the entailment of regular graph constraints turns out to be undecidable. The entailment of role constraints is therefore undecidable as well, and so is the entailment for any other constraints that can encode regular graph constraints. We thus hope that our study of regular graph constraints provides a useful guidance for researchers in choosing an appropriate abstraction for linked data structures.

A regular graph constraint is given by a graph G. A heap H satisfies the constraint iff there exists a graph homomorphism from H to G. The existential quantification over properties of objects is modeled in regular graph constraints as the existence of a homomorphism from H to G. Regular graph constraints allow specifying properties of graphs in some given class of graphs C. If C is the set of trees, regular graph constraints reduce to tree automata [6, 35]; if C is the set of grids, the constraints reduce to domino systems [17]. We therefore view regular graph constraints as a natural generalization of constraints on trees and grids, a generalization that is much weaker than the monadic second-order logic (for which undecidability over non-tree-like domains is well known [7]).

In this paper we consider as the class C the set of *heaps*. Our notion of heap (Definition 2) is motivated by the garbage collected heap in programming languages such as Java or ML. Heaps contain a "root" node (which models the roots of the heap such as global and local variables), and a "null" node (the contents of null-valued fields). All nodes in the heap are reachable from the root (because unreachable nodes in a garbage collected heap may be ignored), and all edges are total functions from nodes to nodes (the functions are total because we consider null to be a graph node as well). We present our results for the case when the heap contains two kinds of fields, labeled "1" and "2". A model with two fields captures the essence of the heap entailment problem, while simplifying our presentation. Note that the entailment problem becomes easily decidable if each object has only one field, because all heaps become lists. On the other hand, if the objects are allowed to have more than two fields, our undecidability result directly applies by picking some two-element subset of the fields in the program.

Undecidability of Entailment. In Section 2.4 we show that there exists a simple and efficient algorithm that decides if a regular graph constraint is satisfiable. In contrast, the entailment problem for regular graph constraints is undecidable. We sketch this undecidability result in Section 3 as the main technical contribution of the paper (additional proof details are in [22]).

A common way of showing the undecidability of problems over graphs is to encode Turing machine computation histories [32] as a special form of graphs called *grids*. The difficulty with showing the undecidability of entailment of regular graph constraints is that regular graph constraints cannot define the subclass of grids among the class of heaps (otherwise the *satisfiability* of regular graph constraints over heaps would be undecidable, which is not the case). To show the *undecidability* of the *entailment* of regular graph constraints, we use constraints on both sides of the implication to restrict the set of possible counterexample models for the implication. For this purpose we introduce a new class of graphs called *corresponder graphs* (Section 3.2). Satisfiability of regular graph constraints over corresponder graphs can encode the existence of a solution of a Post correspondence problem instance, and is therefore undecidable. We give a method for constructing an implication such that all counterexamples for the validity of implication are corresponder graphs which satisfy a given regular graph constraint. This construction shows that the validity of the implication is undecidable. The main difficulty in the proof is a characterization of corresponder graphs using a finite set of allowed and disallowed homomorphic summaries (Section 3.4), a construction vaguely resembling the characterization of planar graphs in terms of forbidden minors [29].

Some Consequences. Regular graph constraints are closed under conjunction and, in certain cases, closed under disjunction (Section 2.3). Due to closure under conjunction, implication $P \Rightarrow Q$ is reducible to the equivalence $P \land Q \Leftrightarrow P$ of regular graph constraints. As a result, the equivalence of two regular graph constraints is also undecidable.

These results place limitations on the completeness of systems such as role analysis [21]. The implication problem for graphs naturally arises in compositional checking of programs whenever procedure preconditions or postconditions are given as regular graph constraints. The complete checking of procedure preconditions at call sites and procedure postconditions is therefore undecidable. Furthermore, it is impossible to build a complete checker for role analysis results if the only inputs to the checker are regular graph constraints expressing the set of heaps at every program point. Similarly, there is no complete procedure for semantically checking equivalence or subsumption of dataflow facts expressed as regular graph constraints; every conservative fixpoint algorithm must perform some unnecessary iterations in some cases.

Related Work. [27] shows the undecidability of alias analysis for programs with general control-flow, strengthening the consequence of Rice's theorem [28] to the case where all program statements are reachable. In contrast, our result shows that local analysis of a *single statement* is undecidable.

Most shape analysis algorithms are non-compositional [5, 16, 23, 30, 31] and many of them were originally used for program optimization. In such an analysis, the imprecision in heap property entailment can cause the analysis to perform some extra fixpoint iterations but may lead to a result that is sufficiently precise for program optimization. We choose a *compositional* approach to program analysis in [21] because it ensures the conformance of the program with respect to the design, increases the scalability of the analysis, and allows the analysis of incomplete programs. Our primary goal is program reliability, and the precision requirements needed to avoid spurious warnings about procedure precondition and postcondition violations seem more demanding than the requirements of analyses intended for program optimization. It is these precision requirements of the compositional analysis that motivate the study of the completeness of heap property entailment algorithms.

Several recent systems support the analysis of tree-like data structures [3, 9, 15, 24, 33, 36]. The restriction to tree-like data structures is in contrast to our notion of a heap, which allows nodes with in-degree greater than one. The presence of non-tree data structures is one of the key factors that make the implication of regular graph constraints undecidable. [1] suggests an alternative way to gain decidability. The logic L_r in [1] allows specifying reachability properties between local variables. What L_r does not allow is defining a set of nodes A using some reachability property and then stating further properties of objects in the set A.

Our experience with regular graph constraints indicates that unrestricted existential quantification over sets of objects quickly leads to heap abstractions whose comparison is undecidable. It is interesting to note that the existential quantification over disjoint sets of objects also occurs in [31], whenever an instrumentation predicate has the "unknown" truth value 1/2. An advantage of the approach in [31] is the existence of *abstraction predicates* that induce a canonical homomorphism for any given concrete heap. Case analysis and appropriate compatibility constraints [31, Page 265] can be used to sharpen the heap properties and eliminate 1/2 values; the implication of heap properties can then be approximated by combining sharpening with simple structural comparison of three-valued structures.

Elements of the first-order logic with transitive closure [20, 31] or first-order logic with inductive definitions [25], [19, Page 57] seem to be necessary for naturally expressing reachability properties. Reachability properties are in turn useful as a criterion for summarizing sets of objects, leading to potentially more intuitive semantics and the possibility of verifying stronger properties. We are therefore considering extending role definitions with regular expressions and exploring the possibility of translating role constraints into three-valued structures [31].

2 Regular Graph Constraints

In this section we define the class of graphs considered in this paper as well as the subclass of heaps as deterministic graphs with reachable nodes. We define the notion of regular graph constraints and show that satisfiability of the constraints over heaps is efficiently decidable. We also state some closure properties of regular graph constraints.

Preliminaries If $r \subseteq A \times B$ and $S \subseteq A$, the *relational image* of set S under r is the set $r[S] = \{y \mid \exists x \in S.\ \langle x, y\rangle \in r\}$. A *word* is a finite sequence of symbols; if $w = a_1 \ldots a_n$ is a word then $|w|$ denotes the length n of w.

2.1 Graphs

We consider only directed graphs in this paper. Our graphs contain two kinds of edges, represented by relations s_1 and s_2. These relations represent fields of objects in an object-oriented program. The constant root represents the root of the graph. We use edges terminated at null to represent partial functions (and abstract representations of graphs containing partial functions).

Definition 1. *A* graph *is a relational structure*

$$G = \langle V, s_1, s_2, \mathsf{null}, \mathsf{root} \rangle$$

where

- V *is a finite set of nodes;*
- $\mathsf{root}, \mathsf{null} \in V$ *are distinct constants,* $\mathsf{root} \neq \mathsf{null}$*;*
- $s_1, s_2 \subseteq V \times V$ *are two kinds of graph edges, such that for all nodes* x

$$\langle \mathsf{null}, x \rangle \in s_i \text{ iff } x = \mathsf{null}$$

for $i \in 1, 2$*.*

Let $\mathcal{G}$ *denote the class of all graphs.*

An s_1-successor of a node x is any element of the set $s_1[\{x\}]$, similarly an s_2-successor of x is any element of $s_2[\{x\}]$. Note that there are exactly two edges originating from null. When drawing graphs we never show these two edges.

Definition 2. *A* heap *is a graph* $G = \langle V, s_1, s_2, \mathsf{null}, \mathsf{root} \rangle$ *where relations* s_1 *and* s_2 *are total functions and where for all* $x \neq \mathsf{null}$*, node* x *is reachable from* root*. Let* $\mathcal{H}$ *denote the class of all heaps.*

Example 3. We can define a heap representing list of length two by
$V = \{\mathsf{root}, x, \mathsf{null}\}$; $s_1 = \{\langle \mathsf{root}, x \rangle, \langle x, \mathsf{null} \rangle, \langle \mathsf{null}, \mathsf{null} \rangle\}$;
$s_2 = \{\langle \mathsf{root}, \mathsf{null} \rangle, \langle x, \mathsf{null} \rangle, \langle \mathsf{null}, \mathsf{null} \rangle\}$.

2.2 Graphs as Constraints

A regular constraint on a graph G is a constraint stating that G can be homomorphically mapped to another graph G'. The constraint satisfaction relation $\rightarrow$ corresponds to abstraction relation in program analyses, [26].

Definition 4. *We say that a graph* G *satisfies the constraints given by a graph* G'*, and write* $G \rightarrow G'$*, iff there exists a homomorphism from* G *to* G'*.*

Homomorphism between directed graphs is a special case of homomorphism of structures [18, Page 5].

Definition 5. *A function* $h : V \rightarrow V'$ *is a homomorphism between graphs*

$$G = \langle V, s_1, s_2, \mathsf{null}, \mathsf{root} \rangle$$

and

$$G' = \langle V', s'_1, s'_2, \mathsf{null}', \mathsf{root}' \rangle$$

iff all of the following conditions hold:

1. $\langle x, y \rangle \in s_i$ *implies* $\langle h(x), h(y) \rangle \in s'_i$*, for all* $i \in \{1, 2\}$
2. $h(x) = \mathsf{root}'$ *iff* $x = \mathsf{root}$

3. $h(x) = \mathsf{null}'$ *iff* $x = \mathsf{null}$

If there exists a homomorphism from G to G', we call G a model *for G'.*

In shape analysis, a homomorphism corresponds to the abstraction function mapping heap objects to the summary nodes in a shape graph. We do not require homomorphism to be onto or to be injective.

We can think of a homomorphism $h : V \to V'$ as a coloring of the graph G by nodes of the graph G'. The color $h(x)$ of a node x restricts the colors of the s_1-successors of x to the colors in $s_1[\{h(x)\}]$ and the colors of the s_2-successors to the colors in $s_2[\{h(x)\}]$. For example, a graph G can be colored with k colors so that the adjacent nodes have different colors iff G is homomorphic to a complete graph without self-loops.

The identity function is a homomorphism from any graph to itself. Therefore, $G \to G$ for every graph G. A composition of homomorphisms is a homomorphism, so $\to$ is transitive.

There is an isomorphism ι between the set of regular graphs constraints and certain subset S of the set of closed formulas in second-order monadic logic. All formulas in S have the form $\exists X_1 \ldots \exists X_k \forall x \forall y.\psi$ where $X_1, \ldots, X_k$ denote sets of nodes, x, y denote individual nodes and ψ is quantifier-free [22, Page 4]. The isomorphism ι has the following property: $H \to G$ iff $H \models \iota(G)$ where $\models$ is the standard Tarskian semantics of monadic second-order logic formulas [7] expressing that the closed formula $\iota(G)$ is true in the model H. With the isomorphism ι in mind, we introduce constraints that are propositional combinations of regular graph constraints and correspond to propositional combinations of the corresponding formulas: $H \to G_1 \wedge G_2$ iff $H \to G_1$ and $H \to G_2$; $H \to G_1 \vee G_2$ iff $H \to G_1$ or $H \to G_2$; $H \to \neg G_1$ iff not $H \to G_1$. Similarly, if C is a class of graphs, we define the satisfiability over C corresponding to satisfiability of formula over a class of models C, and the validity of implication over C corresponding to the validity of implication of formulas over a class of models C.

Definition 6 (Satisfiability). *A graph G is satisfiable over the class of graphs C iff there exists a graph $H \in C$ such that $H \to G$. The satisfiability problem over the class of graphs C is: given a graph G, determine if G is satisfiable.*

Definition 7 (Implication). *We say that G_1 implies G_2 over the class of graphs C, and write $G_1 \leadsto_C G_2$, iff $(H \to G_1)$ implies $(H \to G_2)$ for all graphs $H \in C$. The* implication problem *(or* entailment problem*) for C is: given graphs G_1 and G_2, determine if $G_1 \leadsto_C G_2$.*

We say that a regular graph constraint G_1 is equivalent over C to a regular graph constraint G_2 (and write $G_1 \approx_C G_2$) iff for every $H \in C$, $H \to G_1$ iff $H \to G_2$. Note that $G_1 \sim_C G_2$ iff $C \models \iota(G_1) \Leftrightarrow \iota(G_2)$.

In this paper we consider $C = \mathcal{H}$ as the set of models of regular graph constraints; see Table 1 and [22] for the summary of satisfiability and entailment over different classes of graphs.

2.3 Closure Properties

In this section we give a construction for computing the conjunction of two graphs and a construction for computing the disjunction of two graphs. We use these constructions in Section 3.

Conjunction We show how to use a Cartesian product construction to obtain a conjunction of two graphs G_1 and G_2.

Definition 8 (Cartesian Product). *Let* $G^1 = \langle V^1, s_1^1, s_2^1, \mathsf{null}^1, \mathsf{root}^1 \rangle$ *and* $G^2 = \langle V^2, s_1^2, s_2^2, \mathsf{null}^2, \mathsf{root}^2 \rangle$ *be graphs. Then* $G^0 = G^1 \times G^2$ *is the graph* $G^0 = \langle V^0, s_1^0, s_2^0, \mathsf{null}^0, \mathsf{root}^0 \rangle$ *such that* $\mathsf{null}^0 = \langle \mathsf{null}^1, \mathsf{null}^2 \rangle$*,* $\mathsf{root}^0 = \langle \mathsf{root}^1, \mathsf{root}^2 \rangle$*,*

$$V^0 = \{\mathsf{null}^0, \mathsf{root}^0\} \cup (V^1 \setminus \{\mathsf{null}^1, \mathsf{root}^1\}) \times (V^2 \setminus \{\mathsf{null}^2, \mathsf{root}^2\})$$

and

$$s_i^0 = \{\langle \langle x^1, x^2 \rangle, \langle y^1, y^2 \rangle \rangle \mid \langle x^1, y^1 \rangle \in s_i^1; \langle x^2, y^2 \rangle \in s_i^2\}$$

for $i \in \{1, 2\}$*.*

The proof of the following Proposition 9 is straightforward, see [22].

Proposition 9 (Conjunction via Product). *For every graph* G*,* $G \to G_1 \times G_2$ *iff* ($G \to G_1$ *and* $G \to G_2$)*.*

Disjunction Given our definition of graphs, there is no construction that yields disjunction of arbitrary graphs over the class of heaps [22, Example 26]. To ensure that we can find union graphs over the set of heaps, we simply require $s_2[\{\mathsf{root}\}] = \{\mathsf{null}\}$.

Definition 10 (Orable Graphs). *A graph* $G = \langle V, s_1, s_2, \mathsf{null}, \mathsf{root} \rangle$ *is* orable *iff for all* $x \in V$*,* $\langle \mathsf{root}, x \rangle \in s_2$ *iff* $x = \mathsf{null}$*.*

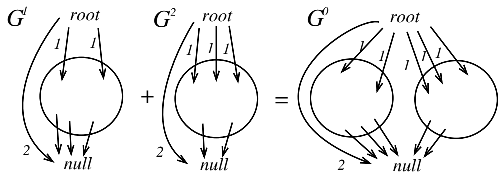

Fig. 1. Graph Sum

Definition 11 (Graph Sum). *Let* $G^1 = \langle V^1, s_1^1, s_2^1, \mathsf{null}, \mathsf{root} \rangle$ *and* $G^2 = \langle V^2, s_1^2, s_2^2, \mathsf{null}, \mathsf{root} \rangle$ *be orable graphs such that* $V^1 \cap V^2 = \{\mathsf{null}, \mathsf{root}\}$*. Then* $G^0 = G^1 + G^2$ *is the graph* $G^0 = \langle V^0, s_1^0, s_2^0, \mathsf{null}, \mathsf{root} \rangle$ *where* $V^0 = V^1 \cup V^2$*,* $s_1^0 = s_1^1 \cup s_2^2$*, and* $s_2^0 = s_2^1 \cup s_2^2$ *(see Figure 1).*

The previous definition is justified by the following Proposition 12. The proof of Proposition 12 uses the fact that every non-null node in a heap is reachable from root, and the assumption $s_2[\{\mathsf{root}\}] = \{\mathsf{null}\}$ for orable graphs, see [22].

Proposition 12 (Disjunction via Sum). *Let G be a heap and G^1 and G^2 be orable graphs. Then $G \to G^1 + G^2$ iff ($G \to G^1$ or $G \to G^2$).*

If G^1 and G^2 are orable graphs, then $G^1 \times G^2$ and $G^1 + G^2$ are also orable. In the sequel we deal only with orable graphs.

```
GraphCleanup:
Repeat the following two operations until the graph stabilizes:

    remove an unreachable node v ≠ null as well as edges incident with v
    remove a node x such that s1[{x}] = ∅ or s2[{x}] = ∅

Mark(x):

    if marked[x] then return, otherwise:
      marked[x] := true;
      pick a s1-successor y of x; marked[⟨x, y⟩] := true; mark(y)
      pick a s2-successor z of x; marked[⟨x, z⟩] := true; mark(z)

SatisfiabilityCheck:

    perform GraphCleanup;
    if the resulting graph G' does not contain root, then G is unsatisfiable;
    otherwise a heap satisfying G can be obtained as follows:
      let all graph nodes and edges be unmarked;
      Mark(root);
      return subgraph containing marked nodes and edges
```

Fig. 2. Satisfiability check for Heaps

2.4 Satisfiability over Heaps

We next consider the satisfiability problem for a regular graph constraint G over the class $\mathcal{H}$ of all heaps. In the context of program checking, graph G denotes a property of the heap. The satisfiability problem is interesting in program checking for several reasons. If graph G is not satisfiable, it represents a contradictory specification. If G was supplied by the developer, it is likely that the specification contains an error. If G was derived by a program analysis considering several cases, then the case corresponding to G can be omitted from consideration because it represents no concrete heaps. Finally, satisfiability is easier than entailment, so it is natural to explore the satisfiability first.

Satisfiability of graphs over the class $\mathcal{H}$ of heaps is efficiently decidable by the algorithm in Figure 2. The goal of the algorithm is to find, given a graph G, whether there exists a heap H such that $H \to G$. Recall the property of a heap that every node has exactly one s_1 outgoing edge and exactly one s_2 outgoing edge. This property need not hold for G, so we cannot take $H = G$ to be the heap proving satisfiability of G. However, if G is satisfiable then G has a subgraph which is a heap. The algorithm in Figure 2 updates the current graph until it becomes a heap or *GraphCleanup* removes the root node. The correctness of the algorithm follows from the fact that *GraphCleanup* removes only nodes which are never in the range of any homomorphism (see [22] for a correctness proof).

3 Undecidability of Implication over Heaps

This section presents the central result of this paper: *The implication of graphs over the class of heaps ($\rightsquigarrow_{\mathcal{H}}$) is undecidable.* To understand the implication problem, observe first that the following Proposition 13 holds.

Proposition 13. *Let C be any class of graphs. Let $G \to G'$. Then $G \rightsquigarrow_C G'$.*

Proposition 13 provides a sufficient condition for the graph implication to hold and is a direct consequence of the transitivity of relation $\to$. The implication problem is difficult because the converse of Proposition 13 for $C = \mathcal{H}$ does not hold. For example, if G is a graph that contains some nodes that can never be an image of a homomorphism and G' is the result of eliminating these nodes, then it is not the case that $G \to G'$, although $G \approx G'$ and thus $G \rightsquigarrow G'$. Moreover, the undecidability of implication $\rightsquigarrow$ means that the incompleteness of $\to$ as an implication test is a fundamental one: $\to$ is a computable relation whereas $\rightsquigarrow$ is not computable. Preceding a $\to$ check with some computable graph-cleanup operation such as one in Figure 2 cannot yield a complete implication test.

3.1 The Idea of the Undecidability Proof

As we have seen in Section 2.4, the satisfiability problem of regular graph constraints over heaps $\mathcal{H}$ is decidable. On the other hand, there are subclasses of $\mathcal{H}$ that have an undecidable satisfiability problem. One such subclass is the class of grids. For grids, regular graph constraints correspond to tiling problems [2, 17], which are undecidable because they can represent Turing machine computation histories [32]. A smaller class can have a more difficult regular graph constraint satisfiability problem if it is not definable within the larger class using regular graph constraints. To show the undecidability of the implication problem, we therefore use constraints on *both sides* of the implication to describe a subclass CG of graphs over which the satisfiability problem is undecidable. We construct the class CG in such a way that we can represent the solutions of the Post Correspondence Problem instances as colorings of graphs in CG. (See [32, Page 183] for Post Correspondence Problem, PCP for short.) We call the elements of CG "corresponder graphs". We choose CG over the class of grids because it seems easier to use the presence and the absence of homomorphisms to characterize CG than to characterize the class of grids. The definition of Corresponder Graphs (Definition 15 and Figure 3) captures the essence of our construction: corresponder graphs need to be sufficiently rich to make Proposition 16 true, and sufficiently simple to make Proposition 17 true. Once we have proven Proposition 16 and Proposition 17, the following Theorem 14 yields the undecidability result which is the central contribution of this paper.

Theorem 14. *The implication of graphs is undecidable over the class of heaps.*

Proof. We reduce satisfiability of graphs over the class of corresponder graphs to the problem of finding a counterexample to an implication of graphs over the class of heaps. Given the reduction in Proposition 16, this establishes that the implication of graphs is undecidable.

Let G be a graph. Consider the implication

$$(G \times P) \rightsquigarrow_{\mathcal{H}} Q \tag{1}$$

We claim that a heap H is a counterexample for this implication iff H is a corresponder graph such that $H \rightarrow G$.

Assume that H is a corresponder graph and $H \rightarrow G$. By Proposition 17, we have $H \rightarrow P$ and $\neg(H \rightarrow Q)$. We then have $H \rightarrow (G \times P)$. Since $\neg(H \rightarrow Q)$, we conclude that H is a counterexample for (1).

Assume now that H is a counterexample for (1). Then $H \rightarrow G \times P$ and $\neg(H \rightarrow Q)$. Since $H \rightarrow P$ and $\neg(H \rightarrow Q)$, by Proposition 17 we conclude that H is a corresponder graph. Furthermore, $H \rightarrow G$. ■

3.2 Corresponder Graphs

Corresponder graphs are a subclass of the class of heaps. Figure 3 shows an example corresponder graph. To encode the matching of words in a PCP instance, a corresponder graph has an upper list of U-nodes and a lower list of L-nodes. These lists are formed using s_1 edges (drawn horizontally in Figure 3). The U-list nodes and L-list nodes are connected using s_2 edges (drawn vertically). These s_2 edges allow a coloring of a corresponder graph to express the matching of letters in words. A solution to a PCP instance is a list of indices of word pairs; our construction encodes this list by the colors of C-nodes of the corresponder graph. s_2 edges from C-nodes partition U-list nodes and L-list nodes into disjoint consecutive list segments. The coloring constraints along these s_2 edges ensure that a coloring of a sequence of U-nodes and a coloring of a sequence of L-nodes encode words from the same pair of the PCP instance. There are twice as many C-nodes as there are word pairs in a PCP instance, to allow an edge to both a U-node and an L-node. The lists of U-nodes and L-nodes in a corresponder graph both have the length $2n$ where the n is the length of the concatenated words in the solution of a PCP instance.

Definition 15 (Corresponder Graphs). *Let $k \geq 2$, $n \geq 2$, $0 = u_0 < u_1 < \ldots < u_{k-1} < n$, and $0 = l_0 < l_1 < \ldots < l_{k-1} < n$. A corresponder graph*

$$\mathsf{CG}(n, k, u_1, \ldots, u_{k-1}, l_1, \ldots, l_{k-1})$$

is a graph isomorphic to $G = \langle V, s_1, s_2, \mathsf{null}, \mathsf{root}\rangle$ *where*

$$\begin{aligned} V = &\{\mathsf{null}, \mathsf{root}\} \cup \{C_0, C_1, \ldots, C_{2k-1}\} \\ &\cup \{U_0, U_1, \ldots, U_{2n-1}\} \cup \{L_0, L_1, \ldots, L_{2n-1}\} \end{aligned}$$

$$\begin{aligned} s_1 = &\{\langle \mathsf{root}, C_0\rangle\} \\ &\cup \{\langle C_i, C_{i+1}\rangle \mid 0 \le i < 2k-1\} \cup \{\langle C_{2k-1}, \mathsf{null}\rangle\} \\ &\cup \{\langle U_i, U_{i+1}\rangle \mid 0 \le i < 2n-1\} \cup \{\langle U_{2n-1}, \mathsf{null}\rangle\} \\ &\cup \{\langle L_i, L_{i+1}\rangle \mid 0 \le i < 2n-1\} \cup \{\langle L_{2n-1}, \mathsf{null}\rangle\} \end{aligned}$$

$$\begin{aligned} s_2 = &\{\langle \mathsf{root}, \mathsf{null}\rangle\} \\ &\cup \{\langle C_{2i}, U_{2u_i}\rangle \mid 0 \le i < k\} \\ &\cup \{\langle C_{2i+1}, L_{2l_i+1}\rangle \mid 0 \le i < k\} \\ &\cup \{\langle U_{2i}, L_{2i}\rangle \mid 0 \le i < n\} \\ &\cup \{\langle L_{2i+1}, U_{2i+1}\rangle \mid 0 \le i < n\} \\ &\cup \big\{\langle U_{2i+1}, \mathsf{null}\rangle \mid i \in \{0, \ldots, n-1\} \setminus \{l_0, \ldots, l_{k-1}\}\big\} \\ &\cup \big\{\langle U_{2i+1}, \mathsf{root}\rangle \mid i \in \{l_0, \ldots, l_{k-1}\}\big\} \\ &\cup \big\{\langle L_{2i}, \mathsf{null}\rangle \mid i \in \{0, \ldots, n-1\} \setminus \{u_0, \ldots, u_{k-1}\}\big\} \\ &\cup \big\{\langle L_{2i}, \mathsf{root}\rangle \mid i \in \{u_0, \ldots, u_{k-1}\}\big\} \end{aligned}$$

We denote the set of all corresponder graphs $\mathsf{CG}(n, k, u_1, \ldots, u_{k-1}, l_1, \ldots, l_{k-1})$ *by* CG.

3.3 Satisfiability over Corresponder Graphs is Undecidable

Proposition 16. *Satisfiability of regular graph constraints over the class of corresponder graphs is undecidable.*

Proof. We give a reduction from PCP. Let $m \ge 2$ and let

$$\langle v_0, w_0\rangle, \langle v_1, w_1\rangle, \ldots, \langle v_{m-1}, w_{m-1}\rangle$$

be an instance of PCP where v_i, w_i are nonempty words

$$\begin{aligned} v_i &= v_i^0 v_i^1 \ldots v_i^{p_i-1} \quad 0 \le i \le m-1 \\ w_i &= w_i^0 w_i^1 \ldots w_i^{q_i-1} \quad 0 \le i \le m-1 \end{aligned}$$

where $p_i = |v_i|$ and $q_i = |w_i|$. We construct a graph G such that there exists a corresponder graph G_0 with the property $G_0 \to G$ iff the PCP instance has a solution.

Consider a PCP instance $\langle c, bc\rangle, \langle ab, a\rangle$. Figure 3 illustrates how a corresponder graph G_0 with a homomorphism from G_0 to G encodes a solution of this PCP instance. Graph G constructed for this PCP instance is presented in Figure 4 using the monadic second-order logic formula $\iota(G)$.

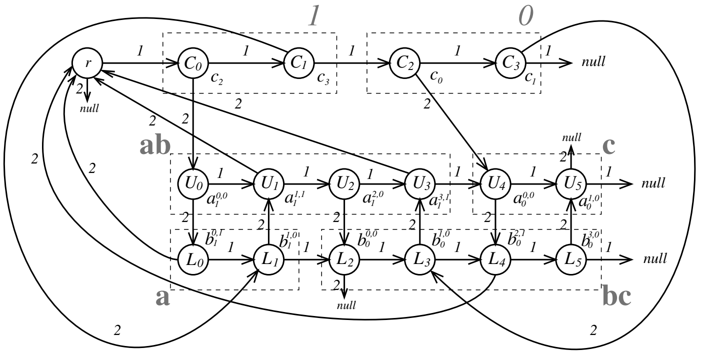

Fig. 3. An example corresponder graph with a homomorphism (coloring) that encodes the solution 1,0 of the PCP instance $\langle v_0, w_0\rangle$, $\langle v_1, w_1\rangle$ where $v_0 = \mathrm{c}, v_1 = \mathrm{ab}$, $w_0 = \mathrm{bc}, w_1 = \mathrm{a}$. The solution is the sequence of indices $1, 0$ which is encoded by the fact that the C-nodes have colors $c_{2\cdot 1}, c_{2\cdot 1+1}$ followed by colors $c_{2\cdot 0}, c_{2\cdot 0+1}$. The four U-node colors $a_1^{0,0}, a_1^{1,1}$, $a_1^{2,0}, a_1^{3,1}$ encode the two positions in the word v_1. The two U-node colors $a_0^{0,0}, a_0^{1,0}$ encode the only position of the word v_0. Analogously, the two L-node colors $b_1^{0,1}, b_1^{1,0}$ encode the only position of the word w_1, whereas $b_0^{0,0}, b_0^{1,0}$, $b_0^{2,1}, b_0^{3,0}$ encode the two positions of the word w_0.

$$
\begin{array}{rl}
\iota(G) \equiv & \exists\, c_0, c_1, c_2, c_3, a_0^{0,0}, a_0^{1,0}, a_0^{1,1}, a_1^{0,0}, a_1^{1,0}, a_1^{1,1}, a_1^{2,0}, a_1^{3,0}, a_1^{3,1}, \\
& \quad b_0^{0,0}, b_0^{0,1}, b_0^{1,0}, b_0^{2,0}, b_0^{2,1}, b_0^{3,0}, b_1^{0,0}, b_1^{0,1}, b_1^{1,0}, \mathsf{null}, \mathsf{root}. \\
& \quad \forall\, x, y.\ \mathsf{disjoint} \wedge \mathsf{null}_{\mathrm{def}} \wedge \mathsf{root}_{\mathrm{def}} \wedge \\
& \quad (s_1(x,y) \Rightarrow C \wedge L_{v_0} \wedge L_{v1} \wedge L_{v_0,v_1} \wedge L_{w_0} \wedge L_{w_1} \wedge L_{w_0,w_1}) \wedge \\
& \quad (s_2(x,y) \Rightarrow I \wedge M_{\mathrm{a}} \wedge M_{\mathrm{b}} \wedge M_{\mathrm{c}} \wedge T) \\
C \equiv & (c_0(x) \Rightarrow c_1(y)) \ \wedge\ (c_2(x) \Rightarrow c_3(y)) \wedge \\
& (\mathsf{root}(x) \vee c_1(x) \vee c_3(x) \Rightarrow c_0(y) \vee c_2(y) \vee \mathsf{null}(y)) \\
L_{v_0} \equiv & a_0^{0,0}(x) \Rightarrow a_0^{1,0}(y) \vee a_0^{1,1}(y) \\
L_{v_1} \equiv & (a_1^{0,0}(x) \Rightarrow a_1^{1,0}(y) \vee a_1^{1,1}(y)) \ \wedge\ (a_1^{1,0}(x) \vee a_1^{1,1}(x) \Rightarrow a_1^{2,0}(y)) \wedge \\
& (a_1^{2,0}(x) \Rightarrow a_1^{3,0}(y) \vee a_1^{3,1}(y)) \\
L_{v_0,v_1} \equiv & a_0^{1,0}(x) \vee a_0^{1,1}(x) \vee a_1^{3,0}(x) \vee a_1^{3,1}(x) \Rightarrow a_0^{0,0}(y) \vee a_1^{0,0}(y) \vee \mathsf{null}(y) \\
& L_{w_0}, L_{w_1}, L_{w_0,w_1} \text{ analogous to } L_{v_0}, L_{v_1}, L_{v_0,v_1} \text{ with } b_k^{i,j} \text{ instead of } a_r^{p,q} \\
I \equiv & (c_0(x) \Rightarrow a_0^{0,0}(y)) \wedge (c_1(x) \Rightarrow b_0^{1,0}(y)) \wedge (c_2(x) \Rightarrow a_1^{0,0}(y)) \wedge (c_3(x) \Rightarrow b_1^{1,0}(y)) \\
M_{\mathrm{a}} \equiv & (a_1^{0,0}(x) \Rightarrow b_1^{1,0}(y)) \ \wedge\ (b_1^{1,0}(x) \Rightarrow a_1^{1,1}(y)) \\
& M_{\mathrm{b}}, M_{\mathrm{c}} \text{ analogous to } M_{\mathrm{a}} \text{ with positions for letter 'b' and 'c' instead of 'a'} \\
T \equiv & (\mathsf{root}(x) \vee a_0^{1,0}(x) \vee a_1^{1,0}(x) \vee a_1^{3,0}(x) \vee b_0^{0,0}(x) \vee b_0^{2,0}(x) \vee b_1^{0,0}(x) \Rightarrow \mathsf{null}(y)) \\
& \wedge\ (a_0^{1,1}(x) \vee a_1^{1,1}(x) \vee a_1^{3,1}(x) \vee b_0^{0,1}(x) \vee b_0^{2,1}(x) \vee b_1^{0,1}(x) \Rightarrow \mathsf{root}(y))
\end{array}
$$

Fig. 4. Formula $\iota(G)$ for the graph G constructed for the PCP instance from Figure 3. $\mathsf{disjoint}$ denotes that all existentially quantified sets are disjoint. $\mathsf{null}_{\mathrm{def}}$ and $\mathsf{root}_{\mathrm{def}}$ define singleton sets contain null and root node, respectively. L_{v_0} connects colors that encode positions in word v_0, similarly for $L_{v_1}, L_{w_0}, L_{w_1}$. L_{v_0,v_1} allows any sequence $(v_0|v_1)*$ of words as U-node colors, analogously for L_{w_0,w_1}. Formula I connects each node representing the choice of word pair k to the first position of the word v_k and w_k. M_{a} connects word positions containing the letter a, similarly for $M_{\mathrm{b}}, M_{\mathrm{c}}$.

In general, define the components of $G = \langle V, s_1, s_2, \mathsf{null}, \mathsf{root}\rangle$ as follows. For every pair of words $\langle v_i, w_i\rangle$ in PCP instance introduce two nodes $c_{2i}, c_{2i+1} \in V$. These nodes summarize C-nodes of a corresponder graph. For every position v_i^j of the word v_i introduce nodes $a_i^{2j,0}$ and $a_i^{2j+1,0}$ and for every position w_i^j introduce nodes $b_i^{2j,0}$ and $b_i^{2j+1,0}$. The a-nodes summarize U-nodes and the b-nodes summarize the L-nodes of the corresponder graph. Introduce further the nodes $b_i^{2j,1}$ to encode the property of a U-node that the matching L-node is colored by some $a_j^{0,0}$ denoting the first position of a word. For analogous reasons introduce $a_i^{2j+1,1}$ nodes. Let

$$\begin{aligned} V = {} & \{\mathsf{null}, \mathsf{root}\} \cup \{c_0, c_1, \dots, c_{2m-1}\} \\ & \cup \{a_i^{j,0} \mid 0 \le i < m; 0 \le j < 2p_i\} \cup \{b_i^{j,0} \mid 0 \le i < m; 0 \le j < 2q_i\} \\ & \cup \{a_i^{2j+1,1} \mid 0 \le i < m; 0 \le j < p_i\} \cup \{b_i^{2j,1} \mid 0 \le i < m; 0 \le j < q_i\} \end{aligned}$$

Define s_1 graph edges as follows.

The c_i nodes are connected into a list that begins with root; every c_{2i} is followed by c_{2i+1}. The pairs c_{2i}, c_{2i+1} for different i can repeat in the list any number of times and in arbitrary order. This list encodes colorings that represent PCP instance solutions.

The nodes representing word positions are linked in the order in which they appear in the word. The last position in a word can be followed by the first position of any other word, or by null. The nodes for the v_i words and the nodes for the w_i words form disjoint lists along the s_1 edges.

$$\begin{aligned} s_1 = {} & \{\langle \mathsf{root}, c_{2i}\rangle \mid 0 \le i < m\} \ \cup \ \{\langle c_{2i}, c_{2i+1}\rangle \mid 0 \le i < m\} \\ & \cup \{\langle c_{2i+1}, c_{2j}\rangle \mid 0 \le i, j < m\} \ \cup \ \{\langle c_{2i+1}, \mathsf{null}\rangle \mid 0 \le i < m\} \\ & \cup \{\langle a_i^{2j,0}, a_i^{2j+1,\alpha}\rangle \mid 0 \le i < m; 0 \le j < p_i; \alpha \in \{0,1\}\} \\ & \cup \{\langle a_i^{2j+1,\alpha}, a_i^{2j+2,0}\rangle \mid 0 \le i < m; 0 \le j < p_i - 1; \\ & \qquad \alpha \in \{0,1\}\} \\ & \cup \{\langle a_i^{2p_i-1,\alpha}, a_j^{0,0}\rangle \mid 0 \le i, j < m; \alpha \in \{0,1\}\} \\ & \cup \{\langle a_i^{2p_i-1,\alpha}, \mathsf{null}\rangle \mid 0 \le i < m; \alpha \in \{0,1\}\} \\ & \cup \{\langle b_i^{2j,\alpha}, b_i^{2j+1,0}\rangle \mid 0 \le i < m; 0 \le j < q_i; \alpha \in \{0,1\}\} \\ & \cup \{\langle b_i^{2j+1,0}, b_i^{2j+2,\alpha}\rangle \mid 0 \le i < m; 0 \le j < q_i - 1; \\ & \qquad \alpha \in \{0,1\}\} \\ & \cup \{\langle b_i^{2q_i-1,0}, b_j^{0,\alpha}\rangle \mid 0 \le i, j < m; \alpha \in \{0,1\}\} \\ & \cup \{\langle b_i^{2q_i-1,0}, \mathsf{null}\rangle \mid 0 \le i < m\} \end{aligned}$$

Define s_2 graph edges as follows.

Every c_j edge points to the position at the beginning of the word. Even numbered nodes point to the a^0-positions; odd numbered nodes point to b^1-positions.

The a_i and b_j word positions are connected so that an a-node points to a b-node for even indices, whereas a b-node points to an a-node for odd indices. The s_2-edges from a-nodes to b-nodes propagate the information that the a-node denotes the first position of some word through the value 1 of index α of the color $b^{2l,\alpha}$. The $b_k^{2l,1}$ nodes have an s_2-edge to root whereas $b_k^{2l,0}$ nodes have an s_2-edge to null. This distinction ensures that every $a_i^{0,0}$-colored node has an incoming edge from a C-node; which implies that every word occurring in the sequence of words that color U-nodes of a corresponder graph is selected by some C-node.

$$
\begin{aligned}
s_2 = \{ & \langle \mathsf{root}, \mathsf{null} \rangle \} \\
& \cup \{ \langle c_{2i}, a_i^{0,0} \rangle \mid 0 \le i < m \} \ \cup \ \{ \langle c_{2i+1}, b_i^{1,0} \rangle \mid 0 \le i < m \} \\
& \cup \{ \langle a_i^{0,0}, b_k^{2l,1} \rangle \mid 0 \le i, k < m; 0 \le l < q_k; v_i^0 = w_k^l \} \\
& \cup \{ \langle a_i^{2j,0}, b_k^{2l,0} \rangle \mid 0 \le i, k < m; 0 < j < p_i; 0 \le l < q_k; \\
& \qquad\qquad v_i^j = w_k^l \} \\
& \cup \{ \langle b_k^{2l,0}, \mathsf{null} \rangle \mid 0 \le k < m; 0 \le l < q_k \} \\
& \cup \{ \langle b_k^{2l,1}, \mathsf{root} \rangle \mid 0 \le k < m; 0 \le l < q_k \} \\
& \cup \{ \langle b_k^{1,0}, a_i^{2j+1,1} \rangle \mid 0 \le i, k < m; 0 \le j < p_i; v_i^j = w_k^l \} \\
& \cup \{ \langle b_k^{2l+1,0}, a_i^{2j+1,0} \rangle \mid 0 \le i, k < m; 0 \le j < p_i; \\
& \qquad\qquad\qquad 0 < l < q_k; v_i^j = w_k^l \} \\
& \cup \{ \langle a_i^{2j+1,0}, \mathsf{null} \rangle \mid 0 \le i < m; 0 \le j < p_i \} \\
& \cup \{ \langle a_i^{2j+1,1}, \mathsf{root} \rangle \mid 0 \le i < m; 0 \le j < p_i \}
\end{aligned}
$$

Claim. The PCP instance has a solution iff there exists a corresponder graph G_0 such that $G_0 \to G$.

This proof of this Claim is not very surprising because we have defined the notion of corresponder graphs to make it true. See [22] for details. ∎

3.4 Using Homomorphisms to Characterize Corresponder Graphs

In Section 3.2 we have defined corresponder graphs as a parameterized family $\mathsf{CG}(n, k, u_1, \ldots, u_{k-1}, l_1, \ldots, l_{k-1})$. In this section we give an alternative characterization of corresponder graphs, as a subclass of heaps that satisfies certain set of graph invariants. We have chosen these invariants so that each invariant is expressible as a homomorphism to some graph or as an absence of a homomorphism to some graph. These graphs show that the class of corresponder graphs is definable as the set of heaps that are counterexamples for the implication of two specific regular graph constraints.

Proposition 17. *There exist graphs P and Q such that for every heap H, $H \to (P \wedge \neg Q)$ iff H is a corresponder graph.*

Proof Sketch. We take P to be the graph in Figure 5 and let $Q = Q_0 + \cdots + Q_{16}$. See Appendix for the figures of graphs P, $Q_0, \ldots, Q_{16}$ and [22] for more proof details.

($\Longleftarrow$) : If G_0 is a corresponder graph we show that $G_0 \to P$ and for all $0 \leq i \leq 16$ it is not the case that $G_0 \to Q_i$. To show $G_0 \to P$ we find a homomorphism mapping C-nodes to c-nodes, U-nodes to a-nodes and L-nodes to b-nodes. Showing $\neg(G_0 \to Q_i)$ is not difficult either (e.g. for Q_0 consider a homomorphic image of the s_1-path from root to null).

($\Longrightarrow$) : (This is the more difficult direction.) Assume $G_0 \to P$ and for all $0 \leq i \leq 16$ it is not the case that $G_0 \to Q_i$. We show that G_0 is a corresponder graph. While P ensures that G_0 has roughly the desired shape, the graphs Q_i ensure the remaining invariants that characterize corresponder graphs. The graphs Q_0 (Figure 6) Q_1 (Figure 7), Q_2 (Figure 8), and Q_3 (Figure 9) eliminate models of P that contain cycles of certain from. For example, if following s_1-edges in G_0 starting from root leads to a cycle, then G_0 must be homomorphic to Q_0 in Figure 9. In this way Q_0 ensures the property of corresponder graphs that following s_1-edges from root eventually leads to null.

The graphs Q_4 (Figure 10), Q_5 (Figure 11), Q_6 (Figure 12), and Q_9 (Figure 15) ensure that certain distinct paths in the graph G_0 commute (i.e. lead to the same node). The graphs Q_7 (Figure 13) and Q_8 (Figure 14) ensure that there is the same number of U and L-nodes in a model of P. The graphs Q_{10} (Figure 16) and Q_{11} (Figure 17) ensure that U or L nodes have an s_2 edge to root iff the U or L node in the same column has an s_2-edge from a C-node. The graph Q_{12} (Figure 18) ensures that if a node C_{2i} has an s_2-edge to a node U_{j_1}, and the node C_{2i+2} has an s_2-edge to U_{j_2}, then U_{j_1} occurs before U_{j_2} in the list of U-nodes. Similarly, Q_{13} (Figure 19) ensures that s_2-edges from C_{2i+1}-nodes to L-nodes are in the proper order.

Finally, graphs Q_{14} (Figure 20), Q_{15} (Figure 21) and Q_{16} (Figure 22) ensure that C-nodes have s_2 edges only to U and L-nodes, and that an L or U node can only have an edge to root, null, a U-node, or an L-node. ∎

Having shown Proposition 17 and Proposition 16, Theorem 14 follows.

4 Conclusion

We have proposed regular graph constraints as an abstraction of mutually recursive properties of objects in potentially cyclic graphs. Regular graph constraints are a natural generalization of tree automata and domino systems. We have shown that satisfiability of regular graph constraints is decidable over the domain of heaps. As a main result, we have shown that the implication of regular graph constraints is undecidable. The consequence of this result is that verifying that procedure preconditions are satisfied is undecidable for any system of constraints that subsumes regular graph constraints.

The fact that decidability of problems with regular constraints is sensitive to the choice of the class of graphs is summarized in Table 1. The table indicates that techniques for reasoning about different classes of graphs may be substantially different. We therefore expect that a good support for mechanized reasoning about data structures will likely contain a set of specialized techniques for different classes of graphs corresponding to commonly used data structures.

class	satisfiability decidable	source	entailment decidable	source
graphs	yes	trivial	yes	easy
trees	yes	[35]	yes	[35]
grids	no	[17]	no	[17]
heaps	yes	present paper	no	present paper

Table 1. Decidability of Regular Graph Constraints

Acknowledgements We thank Chandrasekhar Boyapati, Yuri Gurevich, Patrick Lam, Andreas Podelski, and the participants of the Dagstuhl Seminar 03101 "Reasoning About Shape" for useful discussions. We thank Patrick Lam, Darko Marinov, Chandrasekhar Boyapati, and anonymous reviewers for useful comments on an earlier version of this paper.

References

1. Michael Benedikt, Thomas Reps, and Mooly Sagiv. A decidable logic for linked data structures. In *Proc. 8th ESOP*, 1999.
2. Egon Börger, Erich Gräedel, and Yuri Gurevich. *The Classical Decision Problem.* Springer-Verlag, 1997.
3. Cristiano Calcagno, Luca Cardelli, and Andrew D. Gordon. Deciding validity in a spatial logic for trees. In *ACM TLDI'02*, 2002.
4. Venkatesan T. Chakaravarthy and Susan B. Horwitz. On the non-approximability of points-to analysis. *Acta Informatica*, 38(8):587–598, June 2002.
5. David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proc. ACM PLDI*, 1990.
6. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. release 1 October 2002.
7. Bruno Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In *Handbook of graph grammars and computing by graph transformations, Vol. 1 : Foundations*, chapter 5. World Scientific, 1997.
8. Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. ACM PLDI*, 2002.
9. Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM PLDI*, 2001.
10. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *Proc. 15th ECOOP*, LNCS 2072, pages 130–149. Springer, 2001.
11. Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. 4th USENIX OSDI*, 2000.
12. Ronald Fagin, Larry J. Stockmeyer, and Moshe Y. Vardi. On monadic NP vs monadic co-NP. *Information and Computation*, 120(1), 1995.
13. Cormac Flanagan, K. Rustan M. Leino, Mark Lilibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proc. ACM PLDI*, 2002.

14. Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proc. ACM PLDI*, 2002.
15. Pascal Fradet and Daniel Le Métayer. Shape types. In *Proc. 24th ACM POPL*, 1997.
16. Rakesh Ghiya and Laurie Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proc. 23rd ACM POPL*, 1996.
17. Dora Giammarresi and Antonio Restivo. Two-dimensional languages. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages Vol.3: Beyond Words*. Springer-Verlag, 1997.
18. Wilfrid Hodges. *Model Theory*, volume 42 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1993.
19. Neil Immerman. *Descriptive Complexity*. Springer-Verlag, 1998.
20. Daniel Jackson. Alloy: A lightweight object modelling notation. Technical Report 797, MIT Laboratory for Computer Science, 2000.
21. Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proc. 29th POPL*, 2002.
22. Viktor Kuncak and Martin Rinard. Typestate checking and regular graph constraints. Technical Report 863, MIT Laboratory for Computer Science, 2002.
23. James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proc. ACM PLDI*, Atlanta, GA, June 1988.
24. Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. ACM PLDI*, 2001.
25. Greg Nelson. Techniques for program verification. Technical report, XEROX Palo Alto Research Center, 1981.
26. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
27. Ganesan Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.
28. H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 89:25–29, 1953.
29. Neil Robertson and Paul D. Seymour. Graph minors: A survey. In Ian Anderson, editor, *Surveys in Combinatorics Papers for the London Math. Soc. Lecture Note Series*, 1985.
30. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, 1998.
31. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
32. Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.
33. Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *Proc. 9th ESOP*, Berlin, Germany, March 2000.
34. Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, January 1986.
35. Wolfgang Thomas. Languages, automata, and logic. In *Handbook of Formal Languages Vol.3: Beyond Words*. Springer-Verlag, 1997.
36. David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, 2000.

Appendix: Regular Constraints that Characterize Corresponder Graphs

In this appendix we present the graphs $P, Q_0, \ldots, Q_{16}$ that characterize the class of corresponder graphs CG.

When presenting the graphs we use the following conventions. We use the label r to denote the root of the graph. We label the edges of the relation s_1 relation by 1 and the edges of s_2 by 2. Note that if a node has no outgoing edges, it would be useless in the graph in terms of specifying a set of models G_0. Every graph node in our graphs thus has least one outgoing edge for every label. However, to make the graph presentation clearer, if a node x has an outgoing edge with label a to *every* node in the graph, we simply *omit all a* edges of node x from the sketch. In particular, if a node has no outgoing edges in the graph sketch, it means that its outgoing edges are unconstrained. A double-headed arrow from node x to node y with label a denotes two single arrows, one from x to y and one from y to x, both labeled with a. We do not show the edge $\langle \mathsf{root}, \mathsf{null} \rangle \in s_2$ that is always present in an orable graph. We similarly do not show the edges originating from null. We sometimes display null several times in the same picture; all these occurrences denote the unique null node in the graph.

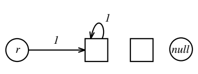

Fig. 6. Graph Q_0

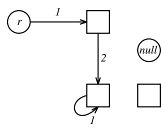

Fig. 7. Graph Q_1

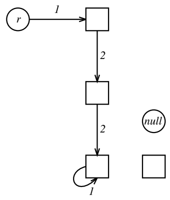

Fig. 8. Graph Q_2

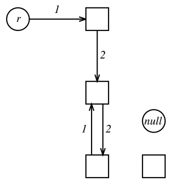

Fig. 9. Graph Q_3

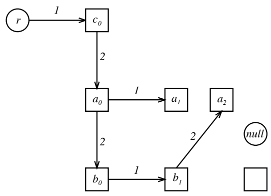

Fig. 10. Graph Q_4

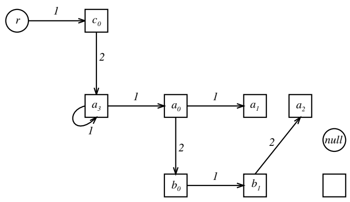

Fig. 11. Graph Q_5

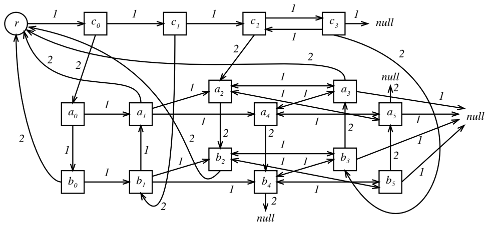

Fig. 5. Graph P

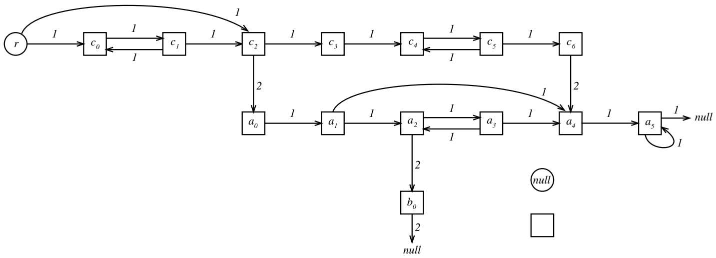

Fig. 18. Graph Q_{12}

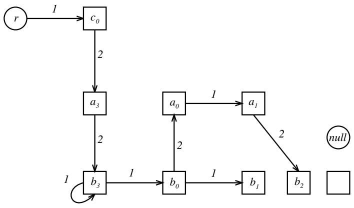

Fig. 12. Graph Q_6

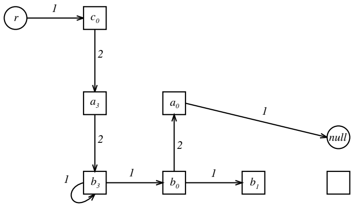

Fig. 14. Graph Q_8

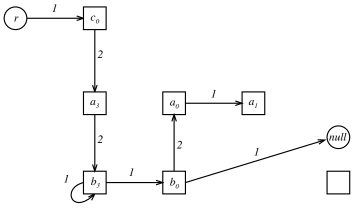

Fig. 13. Graph Q_7

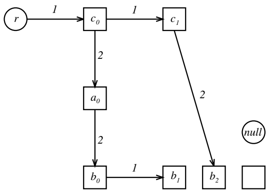

Fig. 15. Graph Q_9

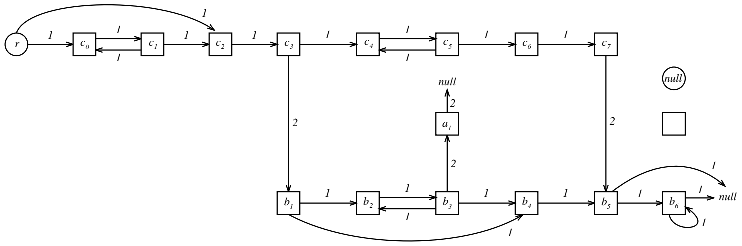

Fig. 19. Graph Q_{13}

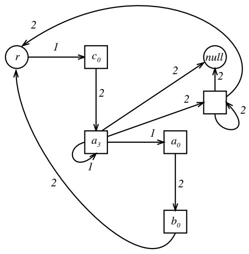

Fig. 16. Graph Q_{10}

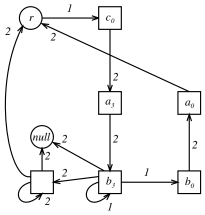

Fig. 17. Graph Q_{11}

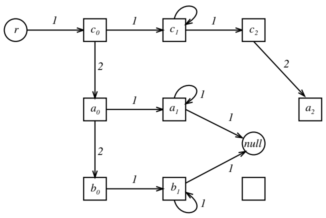

Fig. 20. Graph Q_{14}

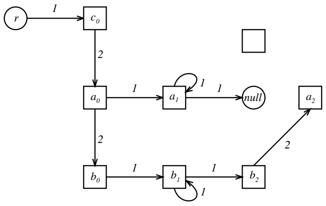

Fig. 21. Graph Q_{15}

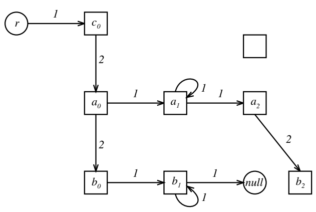

Fig. 22. Graph Q_{16}