The First-Order Theory of Sets with Cardinality Constraints is Decidable

Viktor Kuncak and Martin Rinard
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
{vkuncak,rinard}@csail.mit.edu
MIT CSAIL Technical Report 958

VK0120, July 2004

Abstract

Data structures often use an integer variable to keep track of the number of elements they store. An invariant of such data structure is that the value of the integer variable is equal to the number of elements stored in the data structure. Using a program analysis framework that supports abstraction of data structures as sets, such constraints can be expressed using the language of sets with cardinality constraints. The same language can be used to express preconditions that guarantee the correct use of the data structure interfaces, and to express invariants useful for the analysis of the termination behavior of programs that manipulate objects stored in data structures. In this paper we show the decidability of valid formulas in one such language.

Specifically, we examine the first-order theory that combines 1) Boolean algebras of sets of uninterpreted elements and 2) Presburger arithmetic operations. Our language allows relating the cardinalities of sets to the values of integer variables. We use quantifier elimination to show the decidability of the resulting first-order theory. We thereby disprove a recent conjecture that this theory is undecidable. We describe a basic quantifier-elimination algorithm and its more sophisticated versions. From the analysis of our algorithms we obtain an elementary upper bound on the complexity of the resulting combination. Furthermore, our algorithm yields the decidability of a combination of sets of uninterpreted elements with any decidable extension of Presburger arithmetic. For example, we obtain decidability of monadic second-order logic of n-successors extended with sets of uninterpreted elements and their cardinalities, a result which is in contrast to the undecidability of extensions of monadic-second order logic over strings with equicardinality operator on sets of strings.

Version compiled November 3, 2004, 12:13am.

1 Introduction

Program analysis and verification tools can greatly contribute to software reliability, especially when used throughout the software development process. Such tools are even more valuable if their behavior is predictable, if they can be applied to partial programs, and if they allow the developer to communicate the design information in the form of specifications. Combining the basic idea of [22] with decidable logics leads to analysis tools that have these desirable properties, examples include [34, 26, 4, 41, 54, 30, 31]. These analyses are precise (because they represent loop-free code precisely) and predictable (because the checking of verification conditions terminates either with a realizable counterexample or with a sound claim that there are no counterexamples).

The key challenge in this approach to program analysis and verification is to identify a logic that captures an interesting class of program properties, but is nevertheless decidable. In [31, 30] we identify the first-order theory of Boolean algebras as a useful language for languages with dynamically allocated objects: this language allows expressing generalized typestate properties and reasoning about data structures as dynamically changing sets of objects.

The results of this paper are motivated by the fact that we often need to reason not only about the data structure content, but also about the size of the data structure. For example, we may want to express the fact that the number of elements stored in a data structure is equal to the value of an integer variable that is used to cache the data structure size, or we may want to introduce a decreasing integer measure on the data structure to show program termination. These considerations lead to a natural generalization of the first-order theory of Boolean algebra of sets, a generalization that allows integer variables in addition to set variables, and allows stating relations of the form $|A| = k$ meaning that the cardinality of the set A is equal to the value of the integer variable k. Once we have integer variables, a natural question arises: which relations and operations on integers should we allow? It turns out that, using only the Boolean algebra operations and the cardinality operator, we can already define all operations of Presburger arithmetic. This leads to the structure BAPA, which properly generalizes both Boolean algebras (BA) and Presburger arithmetic (PA). Our paper shows that the first-order theory of structure BAPA is decidable.

A special case of BAPA was recently shown decidable in [57], which allows only quantification over *elements* but not over *sets* of elements. (Note that quantification over sets of elements subsumes quantification over elements because singleton sets can represent elements.) In fact, [57] identifies the problem of decidability of BAPA and conjectures that it is *undecidable*. Our result proves this conjecture false by showing that BAPA is decidable. Moreover, we give a translation of BAPA sentences into PA sentences and derive an elementary upper bound on the worst-case complexity of the validity problem for BAPA.

Contributions and Overview. We can summarize our paper as follows.

1. We **motivate** the use of sets with cardinality constraints through an example (Section 2) and show how to reduce the validity of annotated recursive program schemas (which are a form of imperative programs) to the validity of logic formulas (Section 3).
2. We show the **decidability** of Boolean algebras with Presburger arithmetic (BAPA) using quantifier elimination in Section 5.2. This result immediately implies decidability of the verification problem for schemas whose specifications are expressed in BAPA.

   As a preparation for this result, we review the quantifier elimination technique in Section 4.1 and show its application to the decidability of Boolean algebras (Section 4.2) and Presburger arithmetic (Section 11.1). We also explain why adding the equicardinality operator to Boolean algebras allows defining Presburger arithmetic operations on equivalence classes of sets (Section 5.1).
3. We present an **algorithm** α (Section 5.4) that translates BAPA sentences into PA sentences by translating set quantifiers into integer quantifiers. This is the central result of this paper and shows a natural connection between Boolean algebras and Presburger arithmetic.
4. We analyze our algorithm α and show that it yields an **elementary upper bound** on the worst-case complexity of the validity problem for BAPA sentences that is close to the bound on PA sentences themselves (Section 6).
5. We show that PA sentences generated by translating pure BA sentences can be checked for validity in the space **optimal for Boolean algebras** (Section 6.2).
6. We extend our **algorithm** to **infinite sets** and predicates for distinguishing finite and infinite sets (Section 7).
7. We examine the relationship of our results to the monadic second-order logic (MSOL) of strings (Section 8). In contrast to the undecidability of MSOL with equicardinality operator (Section 11.2), we identify a combination of MSOL over trees with BA that is **decidable**. This result follows from the fact that our algorithm α enables adding BA operations to any extension of Presburger arithmetic, including decidable extensions such as MSOL over strings (Section 8.1).

2 Example

Figure 1 presents a procedure insert in a language that directly manipulates sets. Such languages can either be directly executed [15, 45] or can be derived from executable programs using an abstraction process [30, 31]. The program in Figure 1 manipulates a global set of objects content and an integer field size. The program maintains an invariant I that the size of the set content is equal to the value of the variable size. The insert procedure inserts an element e into the set and correspondingly updates the integer variable. The requires clause (precondition) of the insert procedure is that the parameter e is a non-null reference to an object that is not stored in the set content. The ensures clause (postcondition) of the procedure is that the size variable after the insertion is positive. Note that we represent references to objects (such as the procedure parameter e) as sets with at most one element. An empty set represents a null reference; a singleton set $\{o\}$ represents a reference to object o. The value of a variable after procedure execution is indicated by marking the variable name with a prime.

In addition to the explicit requires and ensures clauses, the insert procedure maintains an invariant, I, which captures the relationship between the size of the set content and the integer variable size. The invariant I is implicitly conjoined with the requires and the ensures clause of the procedure. The Hoare triple [19, 22] in Figure 2 summarizes the resulting correctness condition for the insert procedure.

$$
\begin{array}{l}
\textbf{var}\ \mathsf{content} : \textbf{set};\\
\textbf{var}\ \mathsf{size} : \textbf{integer};\\
\textbf{invariant}\ I \iff (\mathsf{size} = |\mathsf{content}|);\\
\\
\textbf{procedure}\ \mathsf{insert}(e : \mathsf{element})\ \textbf{maintains}\ I\\
\textbf{requires}\ |e| = 1 \wedge |e \cap \mathsf{content}| = 0\\
\textbf{ensures}\ \mathsf{size}' > 0\\
\{\\
\quad \mathsf{content} := \mathsf{content} \cup e;\\
\quad \mathsf{size} := \mathsf{size} + 1;\\
\}
\end{array}
$$

Figure 1: An Example Procedure

$$
\begin{array}{l}
\Big\{|e| = 1 \wedge |e \cap \mathsf{content}| = 0 \wedge \mathsf{size} = |\mathsf{content}|\Big\}\\
\quad \mathsf{content} := \mathsf{content} \cup e; \quad \mathsf{size} := \mathsf{size} + 1;\\
\Big\{\mathsf{size}' > 0 \wedge \mathsf{size}' = |\mathsf{content}'|\Big\}
\end{array}
$$

Figure 2: Hoare Triple for insert Procedure

$$
\begin{array}{l}
\forall e.\ \forall \mathsf{content}.\ \forall \mathsf{content}'.\ \forall \mathsf{size}.\ \forall \mathsf{size}'.\\
\quad (|e| = 1 \wedge |e \cap \mathsf{content}| = 0 \wedge \mathsf{size} = |\mathsf{content}| \wedge\\
\quad \mathsf{content}' = \mathsf{content} \cup e \wedge \mathsf{size}' = \mathsf{size} + 1) \Rightarrow\\
\quad\quad \mathsf{size}' > 0 \wedge \mathsf{size}' = |\mathsf{content}'|
\end{array}
$$

Figure 3: Verification Condition for Figure 2

Figure 3 presents a verification condition corresponding to the Hoare triple in Figure 2. Note that the verification condition contains both set and integer variables, contains quantification over these variables, and relates the sizes of sets to the values of integer variables. Our small example leads to a particularly simple formula; in general, formulas that arise in compositional analysis of set programs with integer variables may contain alternations of existential and universal variables over both integers and sets. This paper shows the decidability of such formulas.

3 First-Order-Logic Program Schemas

To formalize the verification of programs with specifications written in first-order logic, we introduce the notion of first-order-logic program schemas (or *schemas* for short). The schemas motivate the main result of this paper because the decidability of a class of logic formulas implies the decidability of the schema verification problem. The abstraction of programs in general-purpose languages into verifiable schemas can be used to verify partial correctness of programs, and is a particular instance of abstract interpretation [13]. Program schemas have been studied in the past, with the focus primarily on purely functional schemas [2, 9].

Figure 4 presents the syntax of schemas. A schema is a collection of annotated recursive procedures that manipulate global state given by finitely many variables. A recursive program schema is parameterized by a specification language which determines 1) a signature of the specification language, which is some variant of first-order logic and 2) the interpretation of the language, which is some family of multisorted first-order structures. The interpretations of types of global and local variables correspond to the interpretations of sorts in the multisorted language. We use the term "S-schema" for a schema parameterized by a specification language S. The language S is used to encode all basic statements of the schema and to write requires and ensures clauses. The only control structures in a schema are sequential composition ";", nondeterministic choice "$\Box$", and procedure call (denoted using procedure name). For simplicity, procedures in a schema have no parameters; parameter passing can be simulated using assignments to global and local variables.

$$
\begin{array}{rcl}
F & - & \text{first-order formula}\\
s & ::= & F \mid p \mid s\,;\,s \mid s \,\Box\, s \mid \textbf{var}\ x : T.\ s\\
\mathsf{spec}_p & ::= & \textbf{procedure}\ p\\
& & \textbf{requires}\ \mathsf{pre}_p\\
& & \textbf{ensures}\ \mathsf{post}_p\\
& & \{s_{\mathsf{body}(p)}\}\\
\mathsf{schema} & ::= & (\textbf{var}\ x : T)^*\ (\mathsf{spec})^*
\end{array}
$$

Figure 4: Syntax of First-Order Logic Program Schemas

$$
\begin{array}{c}
\text{the meaning of specifications::}\\
\mathsf{spec}_p \quad = \quad (\mathsf{pre}_p \Rightarrow \mathsf{post}_p)\\
\text{rules for reducing statements to formulas::}\\
p \quad \to \quad \mathsf{spec}_p\\
F_1\,;\,F_2 \quad \to \quad \exists \bar{x}_0.(F_1[\bar{x}' := \bar{x}_0] \wedge F_2[\bar{x} := \bar{x}_0])\\
\bar{x} \text{ - variables in pre-state}\\
\bar{x}' \text{ - variables in post-state}\\
F_1 \,\Box\, F_2 \quad \to \quad F_1 \vee F_2\\
\textbf{var}\ x : T.\ F \quad \to \quad \exists x : T.\ F\\
\text{correctness condition for } p\text{::}\\
\forall^*(F_{\mathsf{body}(p)} \Rightarrow \mathsf{spec}_p)\\
\text{where} \quad s_{\mathsf{body}(p)} \xrightarrow{*} F_{\mathsf{body}(p)} \quad \text{using rules above}
\end{array}
$$

Figure 5: Rules that Reduce Procedure Body to a Formula

Provided that variables in S range over sufficiently complex data types (such as integers or terms), schemas are a Turing-complete language. Indeed, the first-order logic can encode assignment statement ($x := t$ is represented by formula $x' = t \wedge \bigwedge_{y \not\equiv x} y' = y$), as well as assume statements ($\mathsf{assume}\ F$ is just $F \Rightarrow \mathsf{skip}$ where skip is $\bigwedge_y y' = y$); nondeterministic choice and assume-statements can encode the if statements; recursion with assume statements can encode while loops. As a consequence of Turing-completeness, the verification of schemas *without* specifications would be undecidable. Because we are assuming that procedures are annotated, the correctness of our recursive program schema reduces to the validity of a set of formulas in the logic, using standard technique of assume-guarantee reasoning. The idea of this reduction is to replace each call to procedure p with the specification given by requires and ensures clause of p, as in Figure 5. After this replacement, the body of each procedure contains only sequential composition, basic statements, and nondeterministic choice. The remaining rules in Figure 5 then reduce the body of a procedure to a single formula.[1] We check the correctness of the procedure by checking that the formula corresponding to the body of the procedure implies the specification of the procedure.

We conclude that if the validity of first-order formulas in the language S is decidable, then the verification problem of an S-schema parameterized by those formulas is decidable. By considering different languages S whose first-order theory is decidable, we obtain different verifiable S-schemas. Example languages whose first-order theories are decidable are term algebras and their generalizations [27], Boolean algebras of sets [32] and Presburger arithmetic [37]. In this paper we establish the decidability of the first-order theory BAPA that combines the quantified formulas of Boolean algebras of sets and Presburger arithmetic. Our result therefore implies the verifiability of a new class of schemas, namely BAPA-schemas.

Schemas and Boolean programs. For a fixed set of predicates, Boolean programs used in predicate abstraction [4, 3, 21] can be seen as a particular form of schemas where the first-order variables range over finite domains. The assumption about finiteness of the domain has important consequences: in the finite domain case the first-order formulas reduce to quantified Boolean formulas, the schemas are not Turing-complete but reduce to pushdown automata, and procedure specifications are not necessary because finite-state properties can be checked using context-free reachability. In this paper we consider schemas where variables may range over infinite domains, yet the verification problem in the presence of specifications is decidable. The advantage of expressive program schemas is that they are closer to the implementation languages, which makes the abstraction of programs into schemas potentially simpler and more precise.

Verification using quantifier-free formulas. Note that the rules in Figure 5 do not introduce quantifier alternations. This means that we obtain verifiable S-schemas even if we restrict S to be a quantifier-free language whose formulas have decidable satisfiability problem. The advantage of using languages whose full first-order theory is decidable is that this approach allows specifications of procedures to use quantifiers to express parameterization (via universal quantifier) and information hiding (via existential quantifier). Moreover, the quantifier elimination technique which we use in this paper shows how to eliminate quantifiers from a formula while preserving its validity. This means that, instead of first applying rules in Figure 5 and then applying quantifier elimination, we may first eliminate all quantifiers from specifications, and then apply rules in Figure 5 yielding a quantifier-free formula. This approach may be more efficient because the decidability of quantifier-free formulas is easier to establish [56, 35, 42, 55, 49].

4 Overview of Quantifier Elimination

For completeness, this section introduces quantifier elimination; quantifier elimination is the central technique used in this paper. After reviewing the basic idea of quantifier elimination in Section 4.1, we explain how to use quantifier elimination to show the decidability of Boolean algebras in Section 4.2. We show the decidability of Presburger arithmetic in Section 11.1.

4.1 Quantifier Elimination

According to [23, Page 70, Lemma 2.7.4], to eliminate quantifiers from arbitrary formulas, it suffices to eliminate $\exists y$ from formulas of the form

$$\exists y. \bigwedge_{0 \le i < n} \psi_i(\bar{x}, y) \tag{1}$$

where $\bar{x}$ is a tuple of variables and $\psi_i(\bar{x}, y)$ is a literal whose all variables are among $\bar{x}, y$. The reason why eliminating formulas of the form (1) suffices is the following. Suppose that the formula is in prenex form and consider the innermost quantifier of a formula. Let ϕ be the subformula containing the quantifier along with the subformula that is the scope of that quantifier. If ϕ is of the form $\forall x.\ \phi_0$ we may replace ϕ with $\neg\exists x.\neg\phi_0$. Hence, we may assume that ϕ is of the form $\exists x.\ \phi_0$. We then transform ϕ_0 into disjunctive normal form and use the fact

$$\exists x.\ (\phi_2 \lor \phi_3) \iff (\exists x.\ \phi_2) \lor (\exists x.\ \phi_3) \tag{2}$$

[1] Note that our formulas encode transition relation as opposed to weakest precondition, so we use $\lor$ to encode non-deterministic choice and $\forall$ for uninitialized variables.

$$
\begin{array}{rcl}
F & ::= & A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid \exists x.F \mid \forall x.F \\
A & ::= & B_1 = B_2 \mid B_1 \subseteq B_2 \mid \\
 & & |B| = C \ \mid \ |B| \geq C \\
B & ::= & x \mid \mathbf{0} \mid \mathbf{1} \mid B_1 \cup B_2 \mid B_1 \cap B_2 \mid B^c \\
C & ::= & 0 \mid 1 \mid 2 \mid \ldots
\end{array}
$$

Figure 6: Formulas of Boolean Algebra (BA)

We conclude that elimination of quantifiers from formulas of form (1) suffices to eliminate the innermost quantifier. By repeatedly eliminating innermost quantifiers we can eliminate all quantifiers from a formula.

We may also assume that y occurs in every literal ψ_i, otherwise we would place the literal outside the existential quantifier using the fact

$$\exists y.\ (A \wedge B) \iff (\exists y.A) \wedge B$$

for y not occurring in B.

To eliminate variables we often use the following identity of theory with equality:

$$\exists x.x = t \wedge \phi(x) \iff \phi(t) \tag{3}$$

The quantifier elimination procedures we present imply the decidability of the underlying theories, because the interpretations of function and relation symbols on some domain A turn out to be effectively computable functions and relations on A. Therefore, the truth-value of every formula without variables is computable. The quantifier elimination procedures we present are all effective. To determine the truth value of a closed formula ϕ on a given model, it therefore suffices to apply the quantifier elimination procedure to ϕ, yielding a quantifier free formula ψ, and then evaluate the truth value of ψ.

4.2 Quantifier Elimination for BA

This section presents a quantifier elimination procedure for Boolean algebras of finite sets. We use the symbols for the set operations as the language of Boolean algebras. $b_1 \cap b_2$, $b_1 \cup b_2$, b_1^c, $\emptyset$, $\mathcal{U}$, correspond to set intersection, set union, set complement, empty set, and full set, respectively. We write $b_1 \subseteq b_2$ for $b_1 \cap b_2 = b_1$, and $b_1 \subset b_2$ for the conjunction $b_1 \subseteq b_2 \ \wedge \ b_1 \neq b_2$.

For every nonnegative integer constant k we introduce formulas of the form $|b| \geq k$ expressing that the set denoted by b has at least k elements, and formulas of the form $|b| = k$ expressing that the set denoted by b has exactly k elements. In this section, cardinality constraints always relate cardinality of a set to a *constant* integer. These properties are first-order definable within Boolean algebra itself:

$$
\begin{array}{rcl}
|b| \geq 0 & \equiv & \mathsf{true} \\
|b| \geq k{+}1 & \equiv & \exists x.\ x \subset b \ \wedge \ |x| \geq k \\
|b| = k & \equiv & |b| \geq k \ \wedge \ \neg |b| \geq k{+}1
\end{array}
$$

We call a language which contains terms $|b| \geq k$ and $|b| = k$ *the language of Boolean algebras with finite constant cardinality constraints.* Figure 6 summarizes the syntax of this language, which we denote BA. Because finite constant cardinality constraints are first-order definable, the language with finite constant cardinality constraints has the same expressive power as the language of Boolean algebras. Removing the restriction that integers are constants is, in fact, what leads to the generalization from Boolean algebras to Boolean algebras with Presburger arithmetic in Section 5, and is the main topic of this paper.

Preliminary observations. Every subset relation $b_1 \subseteq b_2$ is equivalent to $|b_1 \cap b_2^c| = 0$, and every equality $b_1 = b_2$ is equivalent to a conjunction of two subset relations. It is therefore sufficient to consider the first-order formulas whose only atomic formulas are of the form $|b| = k$ and $|b| \geq k$. Furthermore, because k denotes constants, we can eliminate negative literals as follows:

$$
\begin{array}{rcl}
\neg |b| = k & \iff & |b| = 0 \ \vee \cdots \vee \ |b| = k{-}1 \ \vee \ |b| \geq k{+}1 \\
\neg |b| \geq k & \iff & |b| = 0 \ \vee \cdots \vee \ |b| = k{-}1
\end{array}
\tag{4}
$$

Every formula in the language of Boolean algebras can therefore be written in prenex normal form where the matrix (quantifier-free part) of the formula is a disjunction of conjunctions of atomic formulas of the form $|b| = k$ and $|b| \geq k$, with no negative literals. If a term b contains at least one operation of arity one or more, we may assume that the constants $\emptyset$ and $\mathcal{U}$ do not appear in b, because $\emptyset$ and $\mathcal{U}$ can be simplified away. Furthermore, the expression $|\emptyset|$ denotes the integer zero, so all terms of form $|\emptyset| = k$ or $|\emptyset| \geq k$ evaluate to true or false. We can therefore simplify every term b so that either 1) b contains no occurrences of constants $\emptyset$ and $\mathcal{U}$, or 2) $b \equiv \mathcal{U}$.

The following lemma is the main idea behind the quantifier elimination for both BA in this section and BAPA in Section 5.

Lemma 1 *Let $b_1, \ldots, b_n$ be finite disjoint sets, and $l_1, \ldots, l_n, k_1, \ldots, k_n$ be natural numbers. Then the following two statements are equivalent:*

1. *There exists a finite set y such that*

$$\bigwedge_{i=1}^{n} |b_i \cap y| = k_i \wedge |b_i \cap y^c| = l_i \tag{5}$$

2.

$$\bigwedge_{i=1}^{n} |b_i| = k_i + l_i \tag{6}$$

Moreover, the statement continues to hold if for any subset of indices i the conjunct $|b_i \cap y| = k_i$ is replaced by $|b_i \cap y| \geq k_i$ or $|b_i \cap y^c| = l_i$ is replaced by $|b_i \cap y^c| \geq l_i$, provided that $|b_i| = k_i + l_i$ is replaced by $|b_i| \geq k_i + l_i$, as indicated in Figure 7.

Proof. ($\Rightarrow$) Suppose that there exists a set y satisfying (5). Because $b_i \cap y$ and $b_i \cap y^c$ are disjoint, $|b_i| = |b_i \cap y| + |b_i \cap y^c|$, so $|b_i| = k_i + l_i$ when the conjuncts are $|b_i \cap y| = k_i \wedge |b_i \cap y^c| = l_i$, and $|b_i| \geq k_i + l_i$ if any of the original conjuncts have inequality.

($\Leftarrow$) Suppose that (6) holds. First consider the case of equalities. Suppose that $|b_i| = k_i + l_i$ for each of the pairwise disjoint sets $b_1, \ldots, b_n$. For each b_i choose a subset $y_i \subseteq b_i$ such that $|y_i| = k_i$. Because $|b_i| = k_i + l_i$, we have $|b_i \cap y_i^c| = l_i$. Having chosen $y_1, \ldots, y_n$, let $y = \bigcup_{i=1}^{n} y_i$. For $i \neq j$ we have $b_i \cap y_j = \emptyset$ and $b_i \cap y_j^c = b_i$, so $b_i \cap y = y_i$ and $b_i \cap y^c = b_i \cap y_i^c$. By the choice of y_i, we conclude that y is the desired set for which (5) holds. The case of inequalities is analogous: for example, in the case $|b_i \cap y| \geq k_i \wedge |b_i \cap y^c| = l_i$, choose $y_i \subseteq b_i$ such that $|y_i| = |b_i| - l_i$. ∎

Quantifier elimination for BA. We next describe a quantifier elimination procedure for BA. This procedure motivates our algorithm in Section 5.

We first transform the formula into prenex normal form and then repeatedly eliminate the innermost quantifier. As argued in Section 4.1, it suffices to show that we can eliminate an existential quantifier from any existentially quantified conjunction of literals. Consider therefore an arbitrary existentially quantified conjunction of literals

$$\exists y. \bigwedge_{1 \leq i \leq n} \psi_i(\bar{x}, y)$$

where ψ_i is of the form $|b| = k$ or of the form $|b| \geq k$. We assume that y occurs in every formula ψ_i. It follows that no ψ_i contains $|\emptyset|$ or $|\mathcal{U}|$. Let $x_1, \ldots, x_m, y$ be the set of variables occurring in formulas ψ_i for $1 \leq i \leq n$.

First consider the more general case $m \geq 1$. Let for $i_1, \ldots, i_m \in \{0,1\}$, $s_{i_1 \ldots i_m} = x_1^{i_1} \cap \cdots \cap x_m^{i_m}$ where $x^0 = x^c$ and $x^1 = x$. The terms in the set

$$P = \{s_{i_1 \ldots i_m} \mid i_1, \ldots, i_m \in \{0,1\}\}$$

form a partition. Moreover, every Boolean algebra term whose variables are among x_i can be written as a disjoint union of some elements of the partition P. Any Boolean algebra term containing y can be written, for some $p, q \geq 0$ as

original formula	eliminated form
$\exists y. \ \ldots \ \lvert b \cap y\rvert \geq k \wedge \lvert b \cap y^c\rvert \geq l \ \ldots$	$\lvert b\rvert \geq k + l$
$\exists y. \ \ldots \ \lvert b \cap y\rvert = k \wedge \lvert b \cap y^c\rvert \geq l \ \ldots$	$\lvert b\rvert \geq k + l$
$\exists y. \ \ldots \ \lvert b \cap y\rvert \geq k \wedge \lvert b \cap y^c\rvert = l \ \ldots$	$\lvert b\rvert \geq k + l$
$\exists y. \ \ldots \ \lvert b \cap y\rvert = k \wedge \lvert b \cap y^c\rvert = l \ \ldots$	$\lvert b\rvert = k + l$

Figure 7: Rules for Eliminating Quantifiers from Boolean Algebra Expressions

$$(u_1 \cap y) \cup \cdots \cup (u_p \cap y) \cup$$
$$(t_1 \cap y^c) \cup \cdots \cup (t_q \cap y^c)$$

where $u_1, \ldots, u_p \in P$ are pairwise distinct elements from the partition and $t_1, \ldots, t_q \in P$ are pairwise distinct elements from the partition. Because

$$|(u_1 \cap y) \cup \cdots \cup (u_p \cap y) \cup (t_1 \cap y^c) \cup \cdots \cup (t_q \cap y^c)| =$$
$$|u_1 \cap y| + \cdots + |u_p \cap y| + |t_1 \cap y^c| + \cdots + |t_q \cap y^c|$$

a formula of the form $|b| = k$ can be written as

$$\bigvee_{k_1, \ldots, k_p, l_1, \ldots, l_q} \begin{array}{l} (|u_1 \cap y| = k_1 \wedge \cdots \wedge |u_p \cap y| = k_p \wedge \\ |t_1 \cap y^c| = l_1 \wedge \cdots \wedge |t_q \cap y^c| = l_p) \end{array}$$

where the disjunction ranges over nonnegative integers $k_1, \ldots, k_p, l_1, \ldots, l_q \geq 0$ that satisfy

$$k_1 + \cdots + k_p + l_1 + \cdots + l_q = k \tag{7}$$

From (4) it follows that we can perform a similar transformation for formulas of form $|b| \geq k$ (by representing $|b| \geq k$ as boolean combination of $|b| = k$ formulas, applying (7), and traslating the result back into $|b| \geq k$ formulas). After performing this transformation, we bring the formula into disjunctive normal form and continue eliminating the existential quantifier separately for each disjunct, as argued in Section 4.1. We may therefore assume that all conjuncts ψ_i are of one of the forms: $|s \cap y| = k$, $|s \cap y^c| = k$, $|s \cap y| \geq k$, and $|s \cap y^c| \geq k$ where $s \in P$.

If there are two conjuncts both of which contain $|s \cap y|$ for the same s, then either they are contradictory or one implies the other. We therefore assume that for any $s \in P$, there is at most one conjunct ψ_i containing $|s \cap y|$. For analogous reasons we assume that for every $s \in P$ there is at most one conjunct ψ_i containing $|s \cap y^c|$. The result of eliminating the variable y is then given in Figure 7. These rules are applied for all distinct partitions s for which $|s \cap y|$ or $|s \cap y^c|$ occurs. The case when one of the literals containing $|s \cap y|$ does not occur is covered by the case $|s \cap y| \geq k$ for $k = 0$, similarly for a literal containing $|s \cap y^c|$.

It remains to consider the case $m = 0$. Then y is the only variable occurring in conjuncts ψ_i. Every cardinality expression t containing only y reduces to one of $|y|$ or $|y^c|$. If there are multiple literals containing $|y|$, they are either contradictory or one implies the others. We may therefore assume there is at most one literal containing $|y|$ and at most one literal containing $|y^c|$. We eliminate quantifier by applying rules in Figure 7 putting formally $b = \mathcal{U}$, yielding quantifier-free cardinality constraint of the form $|\mathcal{U}| = k$ or of the form $|\mathcal{U}| \geq k$, which does not contain the variable y.

This completes the description of quantifier elimination from an existentially quantified conjunction. By repeating this process for all quantifiers we arrive at a quantifier-free formula ψ. Hence, we have the following fact.

Fact 1 *For every first-order formula ϕ in the language of Boolean algebras with finite cardinality constraints there exists a quantifier-free formula ψ such that ψ is a disjunction of conjunctions of literals of form $|b| \geq k$ and $|b| = k$ (for k denoting constant non-negative integers) where b are terms of Boolean algebra, the free variables of ψ are a subset of the free variables of ϕ, and ψ is equivalent to ϕ on all Boolean algebras of finite sets.*

5 First-Order Theory of BAPA is Decidable

This section presents the main result of this paper: the first-order theory of Boolean algebras with Presburger arithmetic (BAPA) is decidable. We first motivate the operations of the structure BAPA in Section 5.1. We prove the decidability of BAPA in Section 5.2 using a quantifier elimination algorithm that interleaves quantifier elimination for the Boolean algebra part with quantifier elimination for the Presburger arithmetic part. In Section 5.4 we present another algorithm (α) for deciding BAPA, based on the replacement of set quantifiers with integer quantifiers. The analysis of the algorithm α is the subject of Section 6, which derives a worst-case complexity bound on the validity problem for BAPA.

In this section, we interpret Boolean algebras over the family of all powersets of finite sets. Our quantifier elimination is uniform with respect to the size of the universal set. Section 7 extends the result to allow infinite universal sets and reasoning about finiteness of sets.

5.1 From Equicardinality to PA

To motivate the extension of Boolean algebra with all operations of Presburger arithmetic, we derive these operations from a single construct: the equicardinality of sets.

$$
\begin{array}{rcl}
F & ::= & A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid \\
 & & \exists x.F \mid \forall x.F \mid \exists k.F \mid \forall k.F \\
A & ::= & B_1 = B_2 \mid B_1 \subseteq B_2 \mid \\
 & & T_1 = T_2 \mid T_1 < T_2 \mid C \,\mathsf{dvd}\, T \\
B & ::= & x \mid \mathbf{0} \mid \mathbf{1} \mid B_1 \cup B_2 \mid B_1 \cap B_2 \mid B^c \\
T & ::= & k \mid C \mid \mathsf{MAXC} \mid T_1 + T_2 \mid T_1 - T_2 \mid C \cdot T \mid \; |B| \\
C & ::= & \ldots -2 \mid -1 \mid 0 \mid 1 \mid 2 \ldots
\end{array}
$$

Figure 8: Formulas of Boolean Algebras with Presburger Arithmetic (BAPA)

Define the equicardinality relation $\mathsf{eqcard}(b, b')$ to hold iff $|b| = |b'|$, and consider BA extended with relation $\mathsf{eqcard}(b, b')$. Define the ternary relation $\mathsf{plus}(b, b_1, b_2) \iff (|b| = |b_1| + |b_2|)$ by the formula

$$
\begin{array}{l}
\exists x_1.\ \exists x_2.\ \ x_1 \cap x_2 = \emptyset \ \wedge\ b = x_1 \cup x_2 \ \wedge \\
\qquad\qquad \mathsf{eqcard}(x_1, b_1) \wedge \mathsf{eqcard}(x_2, b_2)
\end{array}
$$

The relation $\mathsf{plus}(b, b_1, b_2)$ allows us to express addition using arbitrary sets as representatives for natural numbers. Moreover, we can represent integers as equivalence classes of pairs of natural numbers under the equivalence relation $(x, y) \sim (u, v) \iff x + v = u + y$. This construction allows us to express the unary predicate of being non-negative. The quantification over pairs of sets represents quantification over integers, and quantification over integers with the addition operation and the predicate "being non-negative" can express all operations in Figure 11.

This leads to our formulation of the language BAPA in Figure 8, which contains both the sets and the integers themselves. Note the language has two kinds of quantifiers: quantifiers over integers and quantifiers over sets; we distinguish between these two kinds by denoting integer variables with symbols such as k, l and set variables with symbols such as x, y. We use the shorthand $\exists^+ k.F(k)$ to denote $\exists k.k \geq 0 \wedge F(k)$ and, similarly $\forall^+ k.F(k)$ to denote $\forall k.k \geq 0 \Rightarrow F(k)$. Note that the language in Figure 8 subsumes the language in Figure 11. Furthermore, the language in Figure 8 contains the formulas of the form $|b| = k$ whose Boolean combinations can encode all atomic formulas in Figure 6, as in Section 4.2. This implies that the language in Figure 8 properly generalizes both the language in Figure 11 and the language in Figure 6. Finally, we note that the MAXC constant denotes the size of the finite universe, so we require $\mathsf{MAXC} = |\mathcal{U}|$ (see Section 7 for infinite universe case).

5.2 Basic Algorithm

We first present a simple quantifier-elimination algorithm for BAPA. As explained in Section 4.1, it suffices to eliminate an existential quantifier from a conjunction F of literals of Figure 8. We need to show how to eliminate an integer existential quantifier, and how to eliminate a set existential quantifier. By Section 4.2, assume that all occurrences of set expressions b are within expressions of the form $|b|$. Introduce an integer variable k_i for each such expression $|b_i|$, and write F in the form

$$F \equiv \exists^+ k_1, \ldots, k_p. \bigwedge_{i=1}^{p} |b_i| = k_i \wedge F_1(k_1, \ldots, k_p) \quad (8)$$

where F_1 is a PA formula.

To eliminate an existential integer quantifier $\exists k$ from the formula $\exists k.F$, observe that $\exists k.F(k)$ is equivalent to

$$\exists^+ k_1, \ldots, k_p. \bigwedge_{i=1}^{p} |b_i| = k_i \wedge \exists k.F_1(k, k_1, \ldots, k_p)$$

because k does not occur in the first part of the formula. Using quantifier elimination for Presburger arithmetic, eliminate $\exists k$ from $\exists k.F_1$ yielding a quantifier-free formula $F_2(k_1, \ldots, k_m)$. The formula $\exists k.F(k)$ is then equivalent to $F_2(|b_1|, \ldots, |b_m|)$ and the quantifier has been eliminated.

To eliminate an existential set quantifier $\exists y$ from the formula $\exists y.F$, proceed as follows. Start again from (8), and split each $|b_i|$ into sums of partitions as in Section 4.2. Specifically, let $x_1, \ldots, x_n$ where $y \in \{x_1, \ldots, x_n\}$ be all free set variables in $b_1, \ldots, b_p$, and let $s_1, \ldots, s_m$ for $m = 2^n$ be all set expressions of the form $\bigcap_{j=1}^{n} x_j^{\alpha_j}$ for $\alpha_1, \ldots, \alpha_n \in \{0, 1\}$. Every expression of the form $|b|$ is equal to an expression of the form $\sum_{j=1}^{q} |s_{i_j}|$ for some $i_1, \ldots, i_q$. Introduce an integer variable l_i for each $|s_i|$ where $1 \le i \le m$, and write F in the form

$$\begin{array}{l} \exists^+ l_1, \ldots, l_m. \exists^+ k_1, \ldots, k_p. \\ \quad \bigwedge_{i=1}^{m} |s_i| = l_i \;\wedge\; \bigwedge_{i=1}^{p} t_i = k_i \;\wedge\; F_1(k_1, \ldots, k_p) \end{array} \quad (9)$$

where each t_i is of the form $\sum_{j=1}^{q} l_{i_j}$ for some q and some $i_1, \ldots, i_q$ specific to t_i. Note that only the part $\bigwedge_{i=1}^{m} |s_i| = l_i$ contains set variables, so $\exists y.F$ is equivalent to

$$\begin{array}{l} \exists^+ l_1, \ldots, l_m. \exists^+ k_1, \ldots, k_p. \\ \quad (\exists y. \bigwedge_{i=1}^{m} |s_i| = l_i) \;\wedge\; \bigwedge_{i=1}^{p} t_i = k_i \;\wedge\; F_1(k_1, \ldots, k_p) \end{array} \quad (10)$$

Next, group each s_i of the form $|s \cap y|$ with the corresponding $|s \cap y^c|$ and apply Lemma 1 to replace each pair $|s \cap y| = l_a \wedge |s \cap y^c| = l_b$ with $|s| = l_a + l_b$. As a result, $\exists y. \bigwedge_{i=1}^{m} |s_i| = l_i$ is replaced by a quantifier-free formula of the form $\bigwedge_{i=1}^{m/2} |s'_i| = l_{a_i} + l_{b_i}$. The entire resulting formula is

$$\begin{array}{l} \exists^+ l_1, \ldots, l_m. \exists^+ k_1, \ldots, k_p. \\ \quad \bigwedge_{i=1}^{m/2} |s'_i| = l_{a_i} + l_{b_i} \;\wedge\; \bigwedge_{i=1}^{p} t_i = k_i \;\wedge\; F_1(k_1, \ldots, k_p) \end{array}$$

and contains no set quantifiers, but contains existential integer quantifiers. We have already seen how to eliminate existential integer quantifiers; by repeating the elimination for each of $l_1, \ldots, l_m, k_1, \ldots, k_p$, we obtain a quantifier-free formula. (We can trivially eliminate each k_i by replacing it with t_i, but it remains to eliminate the exponentially many variables $l_1, \ldots, l_m$.)

This completes the description of the basic quantifier elimination algorithm. This quantifier-elimination algorithm is a decision procedure for formulas in Figure 8. We have therefore established the decidability of the language BAPA that combines Boolean algebras and Presburger arithmetic, solving the question left open in [57] for the finite universe case.

Theorem 2 *The validity of* BAPA *sentences over the family of all models with finite universe of uninterpreted elements is decidable.*

Comparison with Quantifier Elimination for BA. Note the difference in the use of Lemma 1 in the quantifier elimination for BA in Section 4.2 compared to the use of Lemma 1 in this section: Section 4.2 uses the statement of the lemma when the cardinalities of sets are known constants, whereas this section uses the statement of the lemma in a more general way, creating the appropriate symbolic sum expression for the cardinality of the resulting sets. On the other hand, the algorithm in this section does not need to consider the case of inequalities for cardinality constraints, because the handling of negations of cardinality constraints is hidden in the subsequent quantifier elimination of integer variables. This simplification indicates that the first-order theories BA and PA naturally fit together; the algorithm in Section 5.4 further supports this impression.

5.3 Reducing the Number of Introduced Integer Variables

This section presents two observations that may reduce the number of integer variables introduced in the elimination of set quantifier in Section 5.2. The algorithm in Section 5.2 introduces 2^n integer variables where n is the number of set variables in the formula F of (8).

First, we observe that it suffices to eliminate the quantifier $\exists y$ from the conjunction of the conjuncts $|b_i| = k_i$ where y occurs in b_i. Let $a_1(y), \ldots, a_q(y)$ be those terms among $b_1, \ldots, b_p$ that contain y, and let $x_1, \ldots, x_{n_1}$ be the free variables in $a_1(y), \ldots, a_q(y)$. Then it suffices to introduce 2^{n_1} integer variables corresponding to the the partitions with respect to

$x_1, \ldots, x_{n_1}$, which may be an improvement because $n_1 \leq n$.

The second observation is useful if the number q of terms $a_1(y), \ldots, a_q(y)$ satisfies the property $2q+1 < n_1$, i.e. there is a large number of variables, but a small number of terms containing them. In this case, consider all Boolean combinations $t_1, \ldots, t_u$ of the $2q$ expressions $a_1(\emptyset), a_1(\mathcal{U}), a_2(\emptyset), a_2(\mathcal{U}), \ldots, a_q(\emptyset), a_q(\mathcal{U})$. For each a_i, we have

$$a_i(y) = (y \cap a_i(\emptyset)) \cup (y^c \cap a_i(\mathcal{U}))$$

Each $a_i(\emptyset)$ and each $a_i(\mathcal{U})$ is a disjoint union of the Boolean combinations of $t_1, \ldots, t_u$, so each $a_i(y)$ is a disjoint union of Boolean combinations of y and the expressions $t_1, \ldots, t_u$ that do not contain y. It therefore suffices to introduce 2^{2q+1} integer variables denoting all terms of the form $y \cap t_i$ and $y^c \cap t_i$, as opposed to 2^{n_1} integer variables.

5.4 Reduction to Quantified PA Sentences

This section presents an algorithm, denoted α, which reduces a BAPA sentence to an equivalent PA sentence with the same number of quantifier alternations and an exponential increase in the total size of the formula. Although we have already established the decidability of BAPA in Section 5.2, the algorithm α of this section is important for several reasons.

1. Given the space and time bounds for Presburger arithmetic sentences [40], the algorithm α yields reasonable space and time bounds for BAPA sentences.

2. Unlike the algorithm in Section 5.2, the algorithm α does not perform any elimination of integer variables, but instead produces an equivalent quantified PA formula. The resulting PA formula can be decided using any decision procedure for PA, including the decision procedures based on automata and model-checking [24, 20].

3. The algorithm α can eliminate set quantifiers from any extension of Presburger arithmetic. We thus obtain a technique for adding a particular form of set reasoning to every extension of Presburger arithmetic, and the technique preserves the decidability of the extension. An example extension where our construction applies is second-order linear arithmetic i.e. monadic second-order logic of one successors, as well monadic second order logic of n-successors, as we note in Section 8.

We next describe the algorithm α for transforming a BAPA sentence F_0 into a PA sentence. The algorithm α is similar to the algorithm in Section 5.2, but, instead of eliminating the integer quantifiers, it accumulates them in a PA formula.

As the first step of the algorithm, transform F_0 into prenex form

$$Q_p v_p. \ldots Q_1 v_1.\ F(v_1, \ldots, v_p) \tag{11}$$

where F is quantifier-free, and each quantifier $Q_i v_i$ is of one the forms $\exists k$, $\forall k$, $\exists y$, $\forall y$ where k denotes an integer variable and y denotes a set variable. As in Section 5.2, separate F into the set part and the purely Presburger arithmetic part by expressing all set relations in terms of $|b|$ terms and by naming each $|b|$, obtaining a formula of the form (8). Next, split all sets into disjoint union of cubes $s_1, \ldots, s_m$ for $m = 2^n$ where n is the number of all set variables, obtaining a formula of the form $Q_p v_p. \ldots Q_1 v_1.F$ where F is of the form (9). Letting $G_1 = F_1(t_1, \ldots, t_p)$, we obtain a formula of the form

$$\begin{array}{l} Q_p v_p. \ldots Q_1 v_1. \\ \quad \exists^+ l_1, \ldots, l_m.\ \bigwedge_{i=1}^{m} |s_i| = l_i \ \land\ G_1 \end{array} \tag{12}$$

where G_1 is a PA formula and $m = 2^n$. Formula (12) is the starting point of the main phase of algorithm α. The main phase of the algorithm successively eliminates quantifiers $Q_1 v_1, \ldots, Q_p v_p$ while maintaining a formula of the form

$$\begin{array}{l} Q_p v_p \ldots Q_r v_r. \\ \quad \exists^+ l_1 \ldots l_q.\ \bigwedge_{i=1}^{q} |s_i| = l_i \ \land\ G_r \end{array} \tag{13}$$

where G_r is a PA formula, r grows from 1 to $p+1$, and $q = 2^e$ where e for $0 \leq e \leq n$ is the number of set variables among $v_p, \ldots, v_r$. The list $s_1, \ldots, s_q$ is the list of all 2^e partitions formed from the set variables among $v_p, \ldots, v_r$.

We next show how to eliminate the innermost quantifier $Q_r v_r$ from the formula (13). During this process, the algorithm replaces the formula G_r with a formula G_{r+1} which has more integer quantifiers. If v_r is an integer variable then the number of sets q remains the same, and if v_r is a set variable, then q reduces from 2^e to 2^{e-1}. We next consider each of the four possibilities $\exists k$, $\forall k$, $\exists y$, $\forall y$ for the quantifier $Q_r v_r$.

Consider first the case $\exists k$. Because k does not occur in $\bigwedge_{i=1}^{q} |s_i| = l_i$, simply move the existential quantifier to G_r and let $G_{r+1} = \exists k.G_r$, which completes the step.

For universal quantifiers, observe that

$$\neg(\exists^+ l_1 \ldots l_q.\ \bigwedge_{i=1}^{q} |s_i| = l_i \ \land\ G_r)$$

is equivalent to

$$\exists^+ l_1 \ldots l_q.\ \bigwedge_{i=1}^{q} |s_i| = l_i \ \land\ \neg G_r$$

because the existential quantifier is used as a let-binding, so we may first substitute all values l_i into G_r, then perform the negation, and then extract back the definitions of all values l_i. Given that the universal quantifier $\forall k$ can be represented as a sequence of unary operators $\neg\exists k\neg$, from the elimination of $\exists k$ we immediately obtain the elimination of $\forall k$; it turns out that it suffices to let $G_{r+1} = \forall k.G_r$.

We next show how to eliminate an existential set quantifier $\exists y$ from

$$\exists y.\ \exists^+ l_1 \ldots l_q.\ \bigwedge_{i=1}^{q} |s_i| = l_i \ \wedge\ G_r \tag{14}$$

which is equivalent to

$$\exists^+ l_1 \ldots l_q.\ (\exists y.\ \bigwedge_{i=1}^{q} |s_i| = l_i) \ \wedge\ G_r \tag{15}$$

Without loss of generality assume that the set variables $s_1, \ldots, s_q$ are numbered such that $s_{2i-1} \equiv s'_i \cap y^c$ and $s_{2i} \equiv s'_i \cap y$ for some cube s'_i. Then apply again Lemma 1 and replace each pair of conjuncts

$$|s'_i \cap y^c| = l_{2i-1} \ \wedge\ |s'_i \cap y| = l_{2i} \tag{16}$$

with the conjunct $|s'_i| = l_{2i-1} + l_{2i}$, yielding formula

$$\exists^+ l_1 \ldots l_q.\ \bigwedge_{i=1}^{q'} |s'_i| = l_{2i-1} + l_{2i} \ \wedge\ G_r \tag{17}$$

for $q' = 2^{e-1}$. Finally, to obtain a formula of the form (13) for $r+1$, introduce fresh variables l'_i constrained by $l'_i = l_{2i-1} + l_{2i}$, rewrite (17) as

$$\exists^+ l'_1 \ldots l'_{q'}.\ \bigwedge_{i=1}^{q'} |s'_i| = l'_i \wedge (\exists l_1 \ldots l_q.\ \bigwedge_{i=1}^{q'} l'_i = l_{2i-1} + l_{2i} \wedge G_r)$$

and let

$$G_{r+1} \equiv \exists^+ l_1 \ldots l_q.\ \bigwedge_{i=1}^{q'} l'_i = l_{2i-1} + l_{2i} \ \wedge\ G_r \tag{18}$$

This completes the description of elimination of an existential set quantifier $\exists y$.

To eliminate a set quantifier $\forall y$, proceed analogously: introduce fresh variables $l'_i = l_{2i-1} + l_{2i}$ and let $G_{r+1} \equiv \forall^+ l_1 \ldots l_q.\ (\bigwedge_{i=1}^{q'} l'_i = l_{2i-1} + l_{2i}) \ \Rightarrow\ G_r$, which can be verified by expressing $\forall y$ as $\neg\exists y\neg$.

After eliminating all quantifiers as described above, we obtain a formula of the form $\exists^+ l.\ |\mathcal{U}| = l \wedge G_{p+1}(l)$. We define the result of the algorithm, denoted $\alpha(F_0)$, to be the PA sentence $G_{p+1}(\mathsf{MAXC})$.

This completes the description of the algorithm α. Given that the validity of PA sentences is decidable, the algorithm α is a decision procedure for BAPA sentences.

$$
\begin{array}{l}
\forall^+ l_1.\forall^+ l_0.\ \mathsf{MAXC} = l_1 + l_0 \Rightarrow \\
\forall^+ l_{11}.\forall^+ l_{01}.\forall^+ l_{10}.\forall^+ l_{00}. \\
l_1 = l_{11} + l_{01} \wedge l_0 = l_{10} + l_{00} \Rightarrow \\
\quad \forall^+ l_{111}.\ \forall^+ l_{011}.\ \forall^+ l_{101}.\ \forall^+ l_{001}. \\
\quad \forall^+ l_{110}.\ \forall^+ l_{010}.\ \forall^+ l_{100}.\ \forall^+ l_{000}. \\
\quad\quad l_{11} = l_{111} + l_{011}\ \wedge l_{01} = l_{101} + l_{001} \wedge \\
\quad\quad l_{10} = l_{110} + l_{010}\ \wedge l_{00} = l_{100} + l_{000} \Rightarrow \\
\quad\quad\quad \forall size.\forall size'. \\
\quad\quad\quad\quad (l_{111} + l_{011} + l_{101} + l_{001} = 1 \wedge \\
\quad\quad\quad\quad l_{111} + l_{011} = 0 \wedge \\
\quad\quad\quad\quad l_{111} + l_{011} + l_{110} + l_{010} = size \wedge \\
\quad\quad\quad\quad l_{100} = 0 \wedge \\
\quad\quad\quad\quad l_{011} + l_{001} + l_{010} = 0 \wedge \\
\quad\quad\quad\quad size' = size + 1) \Rightarrow \\
\quad\quad\quad\quad\quad (0 < size'\ \wedge l_{111} + l_{101} + l_{110} + l_{100} = size')
\end{array}
$$

Figure 9: The translation of the BAPA sentence from Figure 3 into a PA sentence

Theorem 3 *The algorithm α described above maps each* BAPA*-sentence F_0 into an equivalent* PA*-sentence $\alpha(F_0)$.*

Formalization of the algorithm α. To formalize the algorithm α, we have implemented it in the functional programming language O'Caml (Section 11.3).[2] As an illustration, when we run the implementation on the BAPA formula in Figure 3 which represents a verification condition, we immediately obtain the PA formula in Figure 9. Note that the structure of the resulting formula mimics the structure of the original formula: every set quantifier is replaced by the corresponding block of quantifiers over non-negative integers constrained to partition the previously introduced integer variables. Figure 10 presents the correspondence between the set variables of the BAPA formula and the integer variables of the translated PA formula. Note that the relationship $\mathsf{content}' = \mathsf{content} \cup e$ translates into the conjunction of the constraints $|\mathsf{content}' \cap (\mathsf{content} \cup e)^c| = 0 \wedge |(\mathsf{content} \cup e) \cap \mathsf{content}'^c| = 0$, which reduces to the conjunction $l_{100} = 0 \wedge l_{011} + l_{001} + l_{010} = 0$ using the translation of set expressions into the disjoint union of partitions, and the correspondence in Figure 10.

The subsequent sections explore further consequences of the existence of the algorithm α, including an upper bound on the computational complexity of BAPA sentences and the combination of BA with proper extensions of PA.

6 Complexity

In this section we analyze the algorithm α from Section 5.4 and obtain space and time bounds on BAPA from the corresponding space and time bounds for PA.

[2] The implementation is available from `http://www.cag.lcs.mit.edu/~vkuncak/artifacts/bapa/`.

general relationship:

$$l_{i_1,\dots,i_k} = |\mathsf{set}_q^{i_1} \cap \mathsf{set}_{q+1}^{i_2} \cap \dots \cap \mathsf{set}_S^{i_k}|$$

$q = S - (k-1)$, $\quad S$ – number of set variables

in this example:

$$\begin{aligned}
\mathsf{set}_1 &= \mathsf{content}' \\
\mathsf{set}_2 &= \mathsf{content} \\
\mathsf{set}_3 &= e \\
l_{000} &= |\mathsf{content}'^c \cap \mathsf{content}^c \cap e^c| \\
l_{001} &= |\mathsf{content}'^c \cap \mathsf{content}^c \cap e| \\
l_{010} &= |\mathsf{content}'^c \cap \mathsf{content} \cap e^c| \\
l_{011} &= |\mathsf{content}'^c \cap \mathsf{content} \cap e| \\
l_{100} &= |\mathsf{content}' \cap \mathsf{content}^c \cap e^c| \\
l_{101} &= |\mathsf{content}' \cap \mathsf{content}^c \cap e| \\
l_{110} &= |\mathsf{content}' \cap \mathsf{content} \cap e^c| \\
l_{111} &= |\mathsf{content}' \cap \mathsf{content} \cap e|
\end{aligned}$$

Figure 10: The Correspondence between Integer Variables in Figure 9 and Set Variables in Figure 3

We then show that the new decision procedure meets the optimal worst-case bounds for Boolean algebras if applied to purely Boolean algebra formulas. Moreover, by construction, our procedure reduces to the procedure for Presburger arithmetic formulas if there are no set quantifiers. In summary, our decision procedure is optimal for BA, does not impose any overhead for pure PA formulas, and the complexity of the general BAPA validity is not much worse than the complexity of PA itself.

6.1 An Elementary Upper Bound

We next show that the algorithm in Section 5.4 transforms a BAPA sentence F_0 into a PA sentence whose size is at most one exponential larger and which has the same number of quantifier alternations.

If F is a formula in prenex form, let $\mathsf{size}(F)$ denote the size of F, and let $\mathsf{alts}(F)$ denote the number of quantifier alternations in F. Define the iterated exponentiation function $\mathsf{exp}_k(x)$ by $\mathsf{exp}_0(x) = x$ and $\mathsf{exp}_{k+1}(x) = 2^{\mathsf{exp}_k(x)}$. We have the following lemma.

Lemma 4 *For the algorithm α from Section 5.4 there is a constant $c > 0$ such that*

$$\mathsf{size}(\alpha(F_0)) \leq 2^{c \cdot \mathsf{size}(F_0)}$$

$$\mathsf{alts}(\alpha(F_0)) = \mathsf{alts}(F_0)$$

Moreover, the algorithm α runs in $2^{O(\mathsf{size}(F_0))}$ space.

Proof. To gain some intuition on the size of $\alpha(F_0)$ compared to the size of F_0, compare first the formula in Figure 9 with the original formula in Figure 3. Let n denote the size of the initial formula F_0 and let S be the number of set variables. Note that the following operations are polynomially bounded in time and space: 1) transforming a formula into prenex form, 2) transforming relations $b_1 = b_2$ and $b_1 \subseteq b_2$ into the form $|b| = 0$. Introducing set variables for each partition and replacing each $|b|$ with a sum of integer variables yields formula G_1 whose size is bounded by $O(n2^S S)$ (the last S factor is because representing a variable from the set of K variables requires space $\log K$). The subsequent transformations introduce the existing integer quantifiers, whose size is bounded by n, and introduce additionally $2^{S-1} + \ldots + 2 + 1 = 2^S - 1$ new integer variables along with the equations that define them. Note that the defining equations always have the form $l'_i = l_{2i-1} + l_{2i}$ and have size bounded by S. We therefore conclude that the size of $\alpha(F_0)$ is $O(nS(2^S + 2^S))$ and therefore $O(nS2^S)$, which is certainly $O(2^{cn})$ for any $c > 1$. Moreover, note that we have obtained a more precise bound $O(nS2^S)$ indicating that the exponential explosion is caused only by set variables. Finally, the fact that the number of quantifier alternations is the same in F_0 and $\alpha(F_0)$ is immediate because the algorithm replaces one set quantifier with a block of corresponding integer quantifiers. ■

We next consider the worst-case space bound on BAPA. Recall first the following bound on space complexity for PA.

Fact 2 *[17, Chapter 3] The validity of a* PA *sentence of length n can be decided in space $\mathsf{exp}_2(O(n))$.*

From Lemma 4 and Fact 2 we conclude that the validity of BAPA formulas can be decided in space $\mathsf{exp}_3(O(n))$. It turns out, however, that we obtain better bounds on BAPA validity by analyzing the number of quantifier alternations in BA and BAPA formulas.

Fact 3 *[40] The validity of a* PA *sentence of length n and the number of quantifier alternations m can be decided in space $2^{n^{O(m)}}$.*

From Lemma 4 and Fact 3 we obtain our space upper bound, which implies the upper bound on deterministic time.

Theorem 5 *The validity of a* BAPA *sentence of length n and the number of quantifier alternations m can be decided in space $\mathsf{exp}_2(O(mn))$, and, consequently, in deterministic time $\mathsf{exp}_3(O(mn))$.*

If we approximate quantifier alternations by formula size, we conclude that BAPA validity can be decided in space $\mathsf{exp}_2(O(n^2))$ compared to $\mathsf{exp}_2(O(n))$ bound for Presburger arithmetic from Fact 2. Therefore, despite the exponential explosion in the size of the formula in the algorithm α, thanks to the same number of quantifier alternations, our bound is not very far from the bound for Presburger arithmetic.

6.2 Boolean Algebras as a Special Case

We next analyze the result of applying the algorithm α to a pure BA sentence F_0. By a pure BA sentence we mean a BA sentence without cardinality constraints, containing only the standard operations $\cap, \cup, ^c$ and the relations $\subseteq, =$. At first, it might seem that the algorithm α is not a reasonable approach to deciding pure BA formulas given that the best upper bounds for PA are worse than the corresponding bounds for BA. However, we identify a special form of PA sentences $\mathsf{PA}_{\mathsf{BA}} = \{\alpha(F_0) \mid F_0 \text{ is in pure } \mathsf{BA}\}$ and show that such sentences can be decided in $2^{O(n)}$ space, which is optimal for Boolean algebras [25]. Our analysis shows that using binary representations of integers that correspond to the sizes of sets achieves a similar effect to representing these sets as bitvectors, although the two representations are not identical.

Let S be the number of set variables in the initial formula F_0 (recall that set variables are the only variables in F_0). Let $l_1, \ldots, l_q$ be the set of free variables of the formula $G_r(l_1, \ldots, l_q)$; then $q = 2^e$ for $e = S+1-r$. Let $w_1, \ldots, w_q$ be integers specifying the values of $l_1, \ldots, l_q$. We then have the following lemma.

Lemma 6 *For each r where $1 \leq r \leq S$ the truth value of $G_r(w_1, \ldots, w_q)$ is equal to the the truth value of $G_r(\bar{w}_1, \ldots, \bar{w}_q)$ where $\bar{w}_i = \min(w_i, 2^{r-1})$.*

Proof. We prove the claim by induction. For $r = 1$, observe that the translation of a quantifier-free part of the pure BA formula yields a PA formula F_1 whose all atomic formulas are of the form $l_{i_1} + \ldots + l_{i_k} = 0$, which are equivalent to $\bigvee_{j=1}^{k} l_{i_j} = 0$. Therefore, the truth-value of F_1 depends only on whether the integer variables are zero or non-zero, which means that we may restrict the variables to interval $[0, 1]$.

For the inductive step, consider the elimination of a set variable, and assume that the property holds for G_r and for all q tuples of non-negative integers $w_1, \ldots, w_q$. Let $q' = q/2$ and $w'_1, \ldots, w'_{q'}$ be a tuple of non-negative integers. We show that $G_{r+1}(w'_1, \ldots, w'_{q'})$ is equivalent to $G_{r+1}(\bar{w}'_1, \ldots, \bar{w}'_{q'})$.

Suppose first that $G_{r+1}(\bar{w}'_1, \ldots, \bar{w}'_{q'})$ holds. Then for each w'_i there are w_{2i-1} and w_{2i} such that $\bar{w}'_i = u_{2i-1} + u_{2i}$ and $G_r(u_1, \ldots, u_q)$. We define witnesses $w_1, \ldots, w_q$ as follows. If $w'_i \leq 2^r$, then let $w_{2i-1} = u_{2i-1}$ and $w_{2i} = u_{2i}$. If $w'_i > 2^r$ then either $u_{2i-1} > 2^{r-1}$ or $u_{2i} > 2^{r-1}$ (or both). If $u_{2i-1} > 2^{r-1}$, then let $w_{2i-1} = w'_i - u_{2i}$ and $w_{2i} = u_{2i}$. Note that $G_r(\ldots, w_{2i-1}, \ldots) \iff G_r(\ldots, u_{2i-1}, \ldots) \iff G_r(\ldots, 2^{r-1}, \ldots)$ by induction hypothesis because both $u_{2i-1} > 2^{r-1}$ and $w_{2i-1} > 2^{r-1}$. For $w_1, \ldots, w_q$ chosen as above we therefore have $w'_i = w_{2i-1} + w_{2i}$ and $G_r(w_1, \ldots, w_q)$, which by definition of G_{r+1} means that $G_{r+1}(w'_1, \ldots, w'_{q'})$ holds.

Conversely, suppose that $G_{r+1}(w'_1, \ldots, w'_{q'})$ holds. Then there are $w_1, \ldots, w_q$ such that $G_r(w_1, \ldots, w_q)$ and $w'_i = w_{2i-1} + w_{2i}$. If $w_{2i-1} \leq 2^{r-1}$ and $w_{2i} \leq w_{2i}$ then $w'_i \leq 2^r$ so let $u_{2i-1} = w_{2i-1}$ and $u_{2i} = w_{2i}$. If $w_{2i-1} > 2^{r-1}$ and $w_{2i} > w_{2i}$ then let $u_{2i-1} = 2^{r-1}$ and $u_{2i} = 2^{r-1}$. If $w_{2i-1} > 2^{r-1}$ and $w_{2i} \leq 2^{r-1}$ then let $u_{2i-1} = 2^r - w_{2i}$ and $u_{2i} = w_{2i}$. By induction hypothesis we have $G_r(u_1, \ldots, u_q) = G_r(w_1, \ldots, w_q)$. Furthermore, $u_{2i-1} + u_{2i} = \bar{w}'_i$, so $G_{r+1}(\bar{w}'_1, \ldots, \bar{w}'_{q'})$ by definition of G_{r+1}. ■

Now consider a formula F_0 of size n with S free variables. Then $\alpha(F_0) = G_{S+1}$. By Lemma 4, $\mathsf{size}(\alpha(F_0))$ is $O(nS2^S)$. By Lemma 6, it suffices for the outermost variable k to range over the integer interval $[0, 2^S]$, and the range of subsequent variables is even smaller. Therefore, the value of each of the $2^{S+1}-1$ variables can be represented in $O(S)$ space, which is the same order of space used to represent the names of variables themselves. This means that evaluating the formula $\alpha(F_0)$ can be done in the same space $O(nS2^S)$ as the size of the formula. Representing the valuation assigning values to variables can be done in $O(S2^S)$ space, so the truth value of the formula can be evaluated in $O(nS2^S)$ space, which is certainly $2^{O(n)}$. We obtain the following theorem.

Theorem 7 *If F_0 is a pure BA formula with S variables and of size n, then the truth value of $\alpha(B_0)$ can be computed in $O(nS2^S)$ and therefore $2^{O(n)}$ space.*

7 Allowing Infinite Sets

We next sketch the extension of our algorithm α (Section 5.4) to the case when the universe of the structure may be infinite, and the underlying language has the ability to distinguish between finite and infinite sets. Infinite sets are useful in program analysis for modelling pools of objects such as those arising in dynamic object allocation.

We generalize the language of BAPA and the interpretation of BAPA operations as follows.

1. Introduce unary predicate $\mathsf{fin}(b)$ which is true iff b is a finite set. The predicate $\mathsf{fin}(b)$ allows us to generalize our algorithm to the case of infinite universe, and additionally gives the expressive power to distinguish between finite and infinite sets. For example, using $\mathsf{fin}(b)$ we can express bounded quantification over finite or over infinite sets.
2. Define $|b|$ to be the integer zero if b is infinite, and the cardinality of b if b is finite.
3. Introduce propositional variables denoted by letters such as p, q, and quantification over propositional variables. Extend also the underlying PA

formulas with propositional variables, which is acceptable because a variable p can be treated as a shorthand for an integer from $\{0,1\}$ if each use of p as an atomic formula is interpreted as the atomic formula $(p = 1)$. Our extended algorithm uses the equivalences $\mathsf{fin}(b) \Leftrightarrow p$ to represent the finiteness of sets just as it uses the equations $|b| = l$ to represent the cardinalities of finite sets.

4. Introduce a propositional constant FINU such that $\mathsf{fin}(\mathcal{U}) \Leftrightarrow \mathsf{FINU}$. This propositional constant enables equivalence preserving quantifier elimination over the set of models that includes both models with finite universe $\mathcal{U}$ and the models with infinite universe $\mathcal{U}$.

Denote the resulting extended language BAPA^∞.

The following lemma generalizes Lemma 1 for the case of equalities.

Lemma 8 *Let $b_1, \ldots, b_n$ be disjoint sets, $l_1, \ldots, l_n, k_1, \ldots, k_n$ be natural numbers, and $p_1, \ldots, p_n, q_1, \ldots, q_n$ be propositional values. Then the following two statements are equivalent:*

1. *There exists a set y such that*

$$\bigwedge_{i=1}^{n} \begin{array}{l} |b_i \cap y| = k_i \wedge (\mathsf{fin}(b_i \cap y) \Leftrightarrow p_i) \wedge \\ |b_i \cap y^c| = l_i \wedge (\mathsf{fin}(b_i \cap y^c) \Leftrightarrow q_i) \end{array} \tag{19}$$

2.

$$\bigwedge_{i=1}^{n} \begin{array}{l} (p_i \wedge q_i \Rightarrow |b_i| = k_i + l_i) \wedge \\ (\mathsf{fin}(b_i) \Leftrightarrow (p_i \wedge q_i)) \end{array} \tag{20}$$

Proof. $(\Rightarrow)$ Suppose that there exists a set y satisfying (19). From $b_i = (b_i \cap y) \cup (b_i \cap y^c)$, we have $\mathsf{fin}(b_i) \Leftrightarrow (p_i \wedge q_i)$. Furthermore, if p_i and q_i hold, then both $b_i \cap y$ and $b_i \cap y^c$ are finite so the relation $|b_i| = |b_i \cap y| + |b_i \cap y^c|$ holds.

$(\Leftarrow)$ Suppose that (20) holds. For each i we choose a subset $y_i \subseteq b_i$, depending on the truth values of p_i and q_i, as follows.

1. If both p_i and q_i are true, then $\mathsf{fin}(b_i)$ holds, so b_i is finite. Choose y_i as any subset of b_i with k_i elements, which is possible since b_i has $k_i + l_i$ elements.
2. If p_i does not hold, but q_i holds, then $\mathsf{fin}(b_i)$ does not hold, so b_i is infinite. Choose y'_i as any finite set with l_i elements and let $y_i = b_i \setminus y'_i$ be the corresponding cofinite set.
3. Analogously, if p_i holds, but q_i does not hold, then b_i is infinite; choose y_i as any finite subset of b_i with k_i elements.
4. If p_i and q_i are both false, then b_i is also infinite; every infinite set can be written as a disjoint union of two infinite sets, so let y_i be one such set.

Let $y = \bigcup_{i=1}^{n} y_i$. As in the proof of Lemma 1, we have $b_i \cap y = y_i$ and $b_i \cap y^c = y_i^c$. By construction of $y_1, \ldots, y_n$ we conclude that (19) holds. ■

The algorithm α for BAPA^∞ is analogous to the algorithm for BAPA. In each step, the new algorithm maintains a formula of the form

$$\begin{array}{l} Q_p v_p \ldots Q_r v_r. \\ \quad \exists^+ l_1 \ldots l_q.\ \exists p_1 \ldots p_q. \\ \qquad (\bigwedge_{i=1}^{q} |s_i| = l_i \wedge (\mathsf{fin}(s_i) \Leftrightarrow p_i)) \ \wedge \ G_r \end{array}$$

As in Section 5.4, the algorithm eliminates an integer quantifier $\exists k$ by letting $G_{r+1} = \exists k.G_r$ and eliminates an integer quantifier $\forall k$ by letting $G_{r+1} = \forall k.G_r$. Furthermore, just as the algorithm in Section 5.4 uses Lemma 1 to reduce a set quantifier to integer quantifiers, the new algorithm uses Lemma 8 for this purpose. The algorithm replaces

$$\begin{array}{l} \exists y.\ \exists^+ l_1 \ldots l_q.\ \exists p_1 \ldots p_q. \\ \qquad (\bigwedge_{i=1}^{q} |s_i| = l_i \wedge (\mathsf{fin}(s_i) \Leftrightarrow p_i)) \ \wedge \ G_r \end{array}$$

with

$$\begin{array}{l} \exists^+ l'_1 \ldots l'_{q'}.\ \exists p'_1 \ldots p'_{q'}. \\ \qquad (\bigwedge_{i=1}^{q'} |s'_i| = l'_i \wedge (\mathsf{fin}(s'_i) \Leftrightarrow p'_i)) \ \wedge \ G_{r+1} \end{array}$$

for $q' = q/2$, and

$$\begin{array}{rcl} G_{r+1} & \equiv & \exists^+ l_1 \ldots l_q.\ \exists p_1, \ldots, p_q. \\ & & \Big(\bigwedge_{i=1}^{q'} \begin{array}{l} (p_{2i-1} \wedge p_{2i} \Rightarrow l'_i = l_{2i-1} + l_{2i}) \wedge \\ (p'_i \Leftrightarrow (p_{2i-1} \wedge p_{2i})) \end{array} \Big) \\ & & \wedge \ G_r \end{array}$$

For the quantifier $\forall y$ the algorithm analogously generates

$$\begin{array}{rcl} G_{r+1} & \equiv & \forall^+ l_1 \ldots l_q.\ \forall p_1, \ldots, p_q. \\ & & \Big(\bigwedge_{i=1}^{q'} \begin{array}{l} (p_{2i-1} \wedge p_{2i} \Rightarrow l'_i = l_{2i-1} + l_{2i}) \wedge \\ (p'_i \Leftrightarrow (p_{2i-1} \wedge p_{2i})) \end{array} \Big) \\ & & \Rightarrow \ G_r \end{array}$$

After eliminating all quantifiers, the algorithm obtains a formula of the form $\exists^+ l. \exists p.\ |\mathcal{U}| = l \wedge (\mathsf{fin}(\mathcal{U}) \Leftrightarrow p) \wedge G_{p+1}(l,p)$. We define the result of the algorithm to be the PA sentence $G_{p+1}(\mathsf{MAXC}, \mathsf{FINU})$.

This completes our description of the generalized algorithm α for BAPA^∞. The complexity analysis from Section 6 also applies to the generalized version. We also note that our algorithm yields an equivalent formula over any family of models. A sentence is valid in a set of models iff it is valid on each model. Therefore, the validity of a BAPA^∞ sentence F_0 is given by applying to the formula $\alpha(F_0)(\mathsf{MAXC}, \mathsf{FINU})$ a form of universal quantifier over all pairs $(\mathsf{MAXC}, \mathsf{FINU})$ that determine the characteristics of the models in question. For example, for the validity over the models with infinite universe we use $\alpha(F_0)(0, \mathsf{false})$, for validity over all finite models we use $\forall k.\alpha(F_0)(k, \mathsf{true})$, and for the validity over all models we use the PA formula

$$\alpha(F_0)(0, \mathsf{false}) \land \forall k.\alpha(F_0)(k, \mathsf{true}).$$

We therefore have the following result, which answers a generalized version of the question left open in [57].

Theorem 9 *The algorithm above effectively reduces the validity of* BAPA^∞ *sentences to the validity of Presburger arithmetic formulas with the same number of quantifier alternations, and the increase in formula size exponential in the number of set variables; the reduction works for each of the following: 1) the set of all models, 2) the set of models with infinite universe only, and 3) the set of all models with finite universe.*

8 Relationship with MSOL over Strings

The monadic second-order logic (MSOL) over strings is a decidable logic that can encode Presburger arithmetic by encoding addition using one successor symbol and quantification over sets. This logic therefore simultaneously supports sets and integers, so it is natural to examine its relationship with BAPA. It turns out that there are two important differences between MSOL over strings and BAPA:

1. BAPA can express relationships of the form $|A| = k$ where A is a set variable and k is an integer variable; such relation is not definable in MSOL over strings.

2. In MSOL over strings, the sets contain *integers* as elements, whereas in BAPA the sets contain *uninterpreted elements*.

Given these differences, a natural question is to consider the decidability of an extension of MSOL that allows stating relations $|A| = k$ where A is a set of integers and k is an integer variable. Note that by saying $\exists k.|A| = k \land |B| = k$ we can express $|A| = |B|$, so we obtain MSOL with equicardinality constraints. However, extensions of MSOL over strings with equicardinality constraints are known to be undecidable; we review some reductions in Section 11.2. Undecidability results such as these are what perhaps led to the conjecture that BAPA itself is undecidable [57, Page 12]. In this paper we have shown that BAPA is, in fact, decidable and has an elementary decision procedure. Moreover, we next present a combination of BA with MSOL over n-successors that is still decidable.

8.1 Decidability of MSOL with Cardinalities on Uninterpreted Sets

Consider the multisorted language BAMSOL defined as follows. First, BAMSOL contains all relations of monadic second-order logic of n-successors, whose variables range over strings over an n-ary alphabet and sets of such strings. Second, BAMSOL contains sets of uninterpreted elements and boolean algebra operations on them. Third, BAMSOL allows stating relationships of the form $|x| = k$ where x is a set of uninterpreted elements and k is a string representing a natural number. Because all PA operations are definable in MSOL of 1-successor, the algorithm α applies in this case as well. Indeed, the algorithm α only needs a "lower bound" on the expressive power of the theory of integers that BA is combined with: the ability to state constraints of the form $l'_i = l_{2i-1} + l_{2i}$, and quantification over integers. Therefore, applying α to a BAMSOL formula results in an MSOL formula. This shows that BAMSOL is decidable and can be decided using a combination of algorithm α and tool such as [24]. By Lemma 4, the decision procedure for BAMSOL based on translation to MSOL has upper bound of $\mathsf{exp}_n(O(n))$ using a decision procedure such as [24] based on tree automata [11]. The corresponding non-elementary lower bound follows from the lower bound on MSOL itself [48].

9 Related Work

Presburger arithmetic. The original result on decidability of Presburger arithmetic is [37] (see [51, Page 24] for review). This decision procedure was improved in [12] and subsequently in [36]. The best known bound on formula size is obtained using bounded model property techniques [17]. An analysis based on the number of quantifier alternations is presented in [40]. [8] presents a proof-generating version of [12]. The omega test as a decision procedure for Presburger arithmetic is described in [39]. [38] describes how to compute the number of satisfying assignments to free variables in a Presburger arithmetic formula, and describes the applications for computing those numbers for the purpose of program analysis and optimization. Some bounds on quantifier-elimination procedures for Presburger arithmetic are presented in [52]. Automata-theoretic [24, 6] and model checking approaches [20, 46] can also be used to decide Presburger arithmetic and its fragments.

Boolean Algebras. The first results on decidability of Boolean algebras are from [47, 32, 50], [1, Chapter 4] and use quantifier elimination, from which one can derive small model property; [25] gives the complexity of the satisfiability problem. [7] gives an overview of several fragments of set theory including theories with quantifiers but no cardinality constraints and theories with cardinality constraints but no quantification over sets.

Combinations of Decidable Theories. The techniques for combining *quantifier-free* theories [35, 42] and their generalizations such as [55, 56] are of great importance for program verification. This paper shows a particular combination result for *quantified formulas*, which add additional expressive power in writing specifications. Among the general results for quantified formulas are the Feferman-Vaught theorem for products [16], and term powers [27, 28].

Our decidability result is inspired by [57] which gives a solution for the combination of Presburger arithmetic with a notion of sets and quantification of elements, and conjectures that adding the quantification over sets leads to an undecidable theory. The results of this paper prove that the conjecture is false and give an elementary upper bound on the complexity of the combined theory.

We note that [16, Section 8, Page 90] presents the decidability of the first-order theory of a single-sorted variant of BAPA, where the domain is the set of all subsets of some (potentially infinite) set, and the structure contains the equicardinality operator, as suggested in Section 5.1. [3] Compared to [16], we identify the importance of the problem for static program analysis, provide a formalized algorithm in a functional programming language, and analyze the complexity of the algorithm.

Analyses of Dynamic Data Structures. Our new decidability result enables verification tools to reason about sets and their sizes. This capability is particularly important for analyses that handle dynamically allocated data structures where the number of objects is statically unbounded [30, 31, 29, 54, 53, 43, 44]. Recently, these approaches were extended to handle the combinations of the constraints representing data structure contents and constraints representing numerical properties of data structures [43, 10]. Our result provides a systematic mechanism for building precise and predictable versions of such analyses.

10 Conclusion

Motivated by static analysis and verification of relations between data structure content and size, we have introduced the first-order theory of Boolean algebras with Presburger arithmetic (BAPA), established its decidability, presented a decision procedure via reduction to Presburger arithmetic, and showed an elementary upper bound on the worst-case complexity. We expect that our decidability result will play a significant role in verification of programs [35, 14, 18, 5], especially for programs that manipulate dynamically changing sets of objects [30, 31, 29, 54, 53, 43, 44].

[3] We thank Alexis Bes for pointing out the relevance of the Section 8 of [16] in November 2004.

Acknowledgements. The first author thanks Alexis Bes for pointing out to the relevance of [16, Section 8], Chin Wei-Ngan for useful discussions on the analysis of data structure size constraints and useful comments on a version of this paper, the members of the Stanford REACT group and the members of the Berkeley CHESS group on useful discussions on decision procedures and program analysis, and Bruno Courcelle on remarks regarding undecidability of MSOL with equicardinality constraints.

References

[1] W. Ackermann. *Solvable Cases of the Decision Problem.* North Holland, 1954.

[2] Edward Ashcroft, Zohar Manna, and Amir Pnueli. Decidable properties of monadic functional schemas. *J. ACM*, 20(3):489–499, 1973.

[3] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.

[4] Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, 2000.

[5] Nikolaj Bjørner, Anca Browne, Eddie Chang, Michael Colón, Arjun Kapur, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *8th CAV*, volume 1102, pages 415–418, 1996.

[6] Alexandre Boudet and Hubert Comon. Diophantine equations, presburger arithmetic and finite automata. In *21st International Colloquium on Trees in Algebra and Programming - CAAP'96*, volume 1059 of *LNCS*. Springer, 1996.

[7] Domenico Cantone, Eugenio Omodeo, and Alberto Policriti. *Set Theory for Computing.* Springer, 2001.

[8] Amine Chaieb and Tobias Nipkow. Generic proof synthesis for presburger arithmetic. Technical report, Technische Universität München, October 2003.

[9] Ashok K. Chandra. On the decision problems of program schemas with commutative and invertible functions. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 235–242. ACM Press, 1973.

[10] Wei-Ngan Chin, Siau-Cheng Khoo, and Dana N. Xu. Extending sized types with with collection analysis. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics Based Program Manipulation (PEPM'03)*, 2003.

[11] 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 1999.

[12] D. C. Cooper. Theorem proving in arithmetic without multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7, pages 91–100. Edinburgh University Press, 1972.

[13] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proc. 6th POPL*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[14] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, COMPAQ Systems Research Center, 1998.

[15] Robert K. Dewar. Programming by refinement, as exemplified by the SETL representation sublanguage. *Transactions on Programming Languages and Systems*, July 1979.

[16] S. Feferman and R. L. Vaught. The first order properties of products of algebraic systems. *Fundamenta Mathematicae*, 47:57–103, 1959.

[17] Jeanne Ferrante and Charles W. Rackoff. *The Computational Complexity of Logical Theories*, volume 718 of *Lecture Notes in Mathematics*. Springer-Verlag, 1979.

[18] 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.

[19] Robert W. Floyd. Assigning meanings to programs. In *Proc. Amer. Math. Soc. Symposia in Applied Mathematics*, volume 19, pages 19–31, 1967.

[20] Vijay Ganesh, Sergey Berezin, and David L. Dill. Deciding presburger arithmetic by model checking and comparisons with other methods. In *Formal Methods in Computer-Aided Design*. Springer-Verlag, November 2002.

[21] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *31st POPL*, 2004.

[22] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[23] Wilfrid Hodges. *Model Theory*, volume 42 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1993.

[24] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. In *Proc. 5th International Conference on Implementation and Application of Automata*. LNCS, 2000.

[25] Dexter Kozen. Complexity of boolean algebras. *Theoretical Computer Science*, 10:221–247, 1980.

[26] Viktor Kuncak and Martin Rinard. On role logic. Technical Report 925, MIT CSAIL, 2003.

[27] Viktor Kuncak and Martin Rinard. On the theory of structural subtyping. Technical Report 879, Laboratory for Computer Science, Massachusetts Institute of Technology, 2003.

[28] Viktor Kuncak and Martin Rinard. Structural subtyping of non-recursive types is decidable. In *Eighteenth Annual IEEE Symposium on Logic in Computer Science*, 2003.

[29] Viktor Kuncak and Martin Rinard. Generalized records and spatial conjunction in role logic. In *11th Annual International Static Analysis Symposium (SAS'04)*, Verona, Italy, August 26–28 2004.

[30] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Notices*, 39:46–55, March 2004.

[31] Patrick Lam, Viktor Kuncak, and Martin Rinard. Modular pluggable analyses, 2004. Submitted to POPL'05.

[32] L. Loewenheim. Über mögligkeiten im relativkalkül. *Math. Annalen*, 76:228–251, 1915.

[33] Yuri V. Matiyasevich. Enumerable sets are Diophantine. *Soviet Math. Doklady*, 11(2):354–357, 1970.

[34] Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. ACM PLDI*, 2001.

[35] Greg Nelson. Techniques for program verification. Technical report, XEROX Palo Alto Research Center, 1981.

[36] Derek C. Oppen. Elementary bounds for presburger arithmetic. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 34–37. ACM Press, 1973.

[37] M. Presburger. über die vollständigkeit eines gewissen systems der aritmethik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Comptes Rendus du premier Congrès des Mathématiciens des Pays slaves, Warsawa*, pages 92–101, 1929.

[38] William Pugh. Counting solutions to presburger formulas: how and why. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 121–134. ACM Press, 1994.

[39] William Pugh and David Wonnacott. Going beyond integer programming with the omega test to eliminate false data dependences. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):204–211, 1995.

[40] C. R. Reddy and D. W. Loveland. Presburger arithmetic with bounded quantifier alternation. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 320–325. ACM Press, 1978.

[41] Thomas Reps, Mooly Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation*, 2004.

[42] Harald Ruess and Natarajan Shankar. Deconstructing shostak. In *Proc. 16th IEEE LICS*, 2001.

[43] Radu Rugina. Quantitative shape analysis. In *Static Analysis Symposium (SAS'04)*, 2004.

[44] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[45] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for selection of data representations in Setl programs. *Transactions on Programming Languages and Systems*, 3(2):126–143, 1991.

[46] Sanjit A. Seshia and Randal E. Bryant. Deciding quantifier-free presburger formulas using parameterized solution bounds. In *19th IEEE LICS*, 2004.

[47] Thoralf Skolem. Untersuchungen über die Axiome des Klassenkalküls and über "Produktations- und Summationsprobleme", welche gewisse Klassen von Aussagen betreffen. Skrifter utgit av Vidnskapsselskapet i Kristiania, I. klasse, no. 3, Oslo, 1919.

[48] Larry Stockmeyer and Albert R. Meyer. Cosmological lower bound on the circuit complexity of a small problem in logic. *J. ACM*, 49(6):753–784, 2002.

[49] A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.

[50] Alfred Tarski. Arithmetical classes and types of boolean algebras. *Bull. Amer. Math. Soc.*, 55, 64, 1192, 1949.

[51] Ralf Treinen. First-order theories of concrete domains. `http://www.lsv.ens-cachan.fr/~treinen/publications.html`, January 2002.

[52] Volker Weispfenning. Complexity and uniformity of elimination in presburger arithmetic. In *Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, pages 48–53. ACM Press, 1997.

$$
\begin{array}{rcl}
F & ::= & A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid \exists x.F \mid \forall x.F \\
A & ::= & T_1 = T_2 \mid T_1 < T_2 \mid C \,\mathsf{dvd}\, T \\
T & ::= & C \mid T_1 + T_2 \mid T_1 - T_2 \mid C \cdot T \\
C & ::= & \ldots -2 \mid -1 \mid 0 \mid 1 \mid 2 \ldots
\end{array}
$$

Figure 11: Formulas of Presburger Arithmetic PA

[53] Eran Yahav and Ganesan Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *PLDI*, 2004.

[54] Greta Yorsh, Thomas Reps, and Mooly Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *10th TACAS*, 2004.

[55] Calogero G. Zarba. *The Combination Problem in Automated Reasoning*. PhD thesis, Stanford University, 2004.

[56] Calogero G. Zarba. Combining sets with elements. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 762–782. Springer, 2004.

[57] Calogero G. Zarba. A quantifier elimination algorithm for a fragment of set theory involving the cardinality operator. In *18th International Workshop on Unification*, 2004.

11 Appendix

11.1 Quantifier Elimination for PA

For completeness, this section reviews a procedure for quantifier elimination in Presburger arithmetic. For expository purposes we present a version of the quantifier elimination procedure that first transforms the formula into disjunctive normal form. The transformation to disjunctive normal form can be avoided, as observed in [12, 36, 40]. However, our results in Section 5 can be used with other variations of the quantifier-elimination for Presburger arithmetic, and can be formulated in such a way that they not only do not depend on the technique for quantifier elimination for Presburger arithmetic, but do not depend on the technique for deciding Presburger arithmetic at all, allowing the use of automata-theoretic [24] and model checking techniques [20].

Figure 11 presents the syntax of Presburger arithmetic formulas. We interpret formulas over the structure of integers, with the standard interpretation of logical connectives, quantifiers, irreflexive total order on integers, addition, subtraction, and constants. We allow multiplication by a constant only (the case $C \cdot T$ in Figure 11), which is expressible using addition and subtraction. If c is a constant and t is a term, the notation $c \,\mathsf{dvd}\, t$ denotes that c divides t i.e., $t \bmod c = 0$. We assume that $c > 0$ in each formula $c \,\mathsf{dvd}\, t$.

We review a simple algorithm for deciding Presburger arithmetic inspired by [37], [51, Page 24], [12]. The algorithm we present eliminates an existential quantifier from a conjunction of literals in the language of Figure 11, which suffices by Section 4.1. Note first that we may eliminate all equalities $t_1 = t_2$ because

$$t_1 = t_2 \iff (t_1 < t_2 + 1) \vee (t_2 < t_1 + 1)$$

Next, we have $\neg(t_1 < t_2) \iff t_2 < t_1 + 1$ and

$$\neg(c \,\mathsf{dvd}\, t) \iff \bigvee_{i=1}^{c-1} c \,\mathsf{dvd}\, t{+}i$$

which means that it suffices to consider the elimination of an existential quantifier from the formula of the form $\bigwedge_{i=1}^{n} A$ where each A is an atomic formula of the form $t_1 < t_2$ or of the form $c \,\mathsf{dvd}\, t$. Each of the terms t_1, t_2, t is linear, so we can write it in the form $c_0 + \sum_{i=1}^{k} c_i x_i$. Consequently, we may transform the atomic formulas into forms $0 < c_0 + \sum_{i=1}^{k} c_i x_i$ and $c \,\mathsf{dvd}\, c_0 + \sum_{i=1}^{k} c_i x_i$. Consider an elimination of an existential quantifier $\exists x$ from a conjunction of such atomic formulas. Let $c_1, \ldots, c_p$ be the coefficients next to x in the conjuncts and let $M > 0$ be the least common multiple of $c_1, \ldots, c_p$. Multiply each atomic formula of the form $0 < c_i x + t$ by $M/|c_i|$, and multiply each atomic formula of the form $c \,\mathsf{dvd}\, c_i x + t$ by M/c_i (yielding $Mc \,\mathsf{dvd}\, Mx + (M/c_i)t$). The result is an equivalent conjunction of formulas with the property that, in each conjunct, the coefficient next to x is M or $-M$. The conjunction is therefore of the form $F_0(Mx)$ for some formula F_0. The formula $\exists x.F_0(Mx)$ is equivalent to the formula $\exists y.(F_0(y) \wedge M \,\mathsf{dvd}\, y)$. By moving x to the left-hand side if its coefficient is -1 in the term t of each atomic formula $0 < t$, replacing $c \,\mathsf{dvd}\, -y + t$ by $c \,\mathsf{dvd}\, y - t$, and renaming y as x, it remains to eliminate an existential quantifier from $\exists x.F(x)$ where

$$F(x) \;\equiv\; \bigwedge_{i=1}^{q} x < a_i \;\wedge\; \bigwedge_{i=1}^{p} b_i < x \;\wedge\; \bigwedge_{i=1}^{r} c_i \,\mathsf{dvd}\, x + t_i$$

where x does not occur in any of a_i, b_i, t_i. Let N be the least common multiple of $c_1, \ldots, c_r$. Clearly, if $x = u$ is a solution of $F_1(x) \equiv \bigwedge_{i=1}^{r} c_i \,\mathsf{dvd}\, x + t_i$, then so is $x = u + Nk$ for every integer k. If $p = 0$ and $q = 0$ then $\exists y.F(y)$ is equivalent to e.g. $\bigwedge_{i=1}^{N} F(i)$, which eliminates the quantifier. Otherwise, suppose that $p > 0$ (the case $q > 0$ is analogous, and if $p > 0$ and $q > 0$ then both are applicable). Suppose for a moment that we are given an assignment to free variables of $\exists x.F(x)$. Then the formula $\exists x.F(x)$ is equivalent to $\bigvee_u F_1(u)$ where u ranges over the elements u such that

$$\max(b_1, \ldots, b_p) < u < \min(a_1, \ldots, a_q)$$

Let $b = \max(b_1, \ldots, b_p)$. Then $\exists x.F(x)$ is equivalent to $\bigvee_{i=1}^{N} F(b+i)$. Namely, if a solution exists, it must be of the form $b + i$ for some $i > 0$, and it suffices to check N

consecutive numbers as argued above. Of course, we do not know the assignment to free variables of $\exists x.F(x)$, so we do not know for which b_i we have $b = b_i$. However, we can check all possibilities for b_i. We therefore have that $\exists y.F(y)$ is equivalent to

$$\bigvee_{j=1}^{p} \bigvee_{i=1}^{N} F(b_j + i)$$

This completes the sketch of the quantifier elimination for Presburger arithmetic. We obtain the following result.

Fact 4 *For every first-order formula ϕ in the language of Presburger arithmetic of Figure 11 there exists a quantifier-free formula ψ such that ψ is a disjunction of conjunctions of literals, the free variables of ψ are a subset of the free variables of ϕ, and ψ is equivalent to ϕ over the structure of integers.*

11.2 Undecidability of MSOL of Integer Sets with Cardinalities

We first note that there is a reduction from the Post Correspondence Problem that shows the undecidability of MSOL with equicardinality constraints. Namely, we can represent binary strings by finite sets of natural numbers. In this encoding, given a position, MSOL itself can easily express the local property that, at a given position, a string contains a given finite substring. The equicardinality gives the additional ability of finding an n-th element of an increasing sequence of elements. To encode a PCP instance, it suffices to write a formula checking the existence of a string (represented as set A) and the existence of two increasing sequences of equal length (represented by sets U and D), such that for each i, there exists a pair (a_j, b_j) of PCP instance such that the position starting at U_i contains the constant string a_j, and $U_{i+1} = U_i + |a_j|$, and similarly the position starting at D_i contains b_j and $D_{i+1} = D_i + |b_j|$.

The undecidability of MSOL over strings extended with equicardinality can also be shown by encoding multiplication of natural numbers. Given $A = \{1, 2, ..., x\}$ and $B = \{1, 2, ..., y\}$, we can define a set the set $C = \{x, 2x, ...y \cdot x\}$ as the set with the same number of elements as B, that contains x, and that is closed under unary operation $z \mapsto z + y$. Therefore, if we represent a natural number n as the set $\{1, \ldots, n\}$, we can define both multiplication and addition of integers. This means that we can write formulas whose satisfiability answers the existence of solutions of Diophantine equations, which is undecidable by [33]. A similar reduction to a logic that does not even have quantification over sets is presented in [57].

11.3 O'Caml source code of algorithm α

```
(* ------------------------------------------- *)
(*             datatype of formulas            *)
(* ------------------------------------------- *)

type ident = string

type binder = | Forallset | Existsset
              | Forallint | Existsint
              | Forallnat | Existsnat

type form =
  | Not of form
  | And of form list
  | Or of form list
  | Impl of form * form
  | Bind of binder * ident * form
  | Less of intTerm * intTerm
  | Inteq of intTerm * intTerm
  | Seteq of setTerm * setTerm
  | Subseteq of setTerm * setTerm
and intTerm =
  | Intvar of ident
  | Const of int
  | Plus of intTerm list
  | Minus of intTerm * intTerm
  | Times of int * intTerm
  | Card of setTerm
and setTerm =
  | Setvar of ident
  | Emptyset
  | Fullset
  | Complement of setTerm
  | Union of setTerm list
  | Inter of setTerm list

let maxcard = "MAXC"

(* ------------------------------------------- *)
(*             algorithm \alpha                *)
(* ------------------------------------------- *)

  (* replace Seteq and Subseteq with Card(...)=0 *)
let simplify_set_relations (f:form) : form =
  let rec sform f = match f with
  | Not f -> Not (sform f)
  | And fs -> And (List.map sform fs)
  | Or fs -> Or (List.map sform fs)
  | Impl(f1,f2) -> Impl(sform f1,sform f2)
  | Bind(b,id,f1) -> Bind(b,id,sform f1)
  | Less(it1,it2) -> Less(siterm it1, siterm it2)
  | Inteq(it1,it2) -> Inteq(siterm it1,siterm it2)
  | Seteq(st1,st2) -> And[sform (Subseteq(st1,st2));
                          sform (Subseteq(st2,st1))]
  | Subseteq(st1,st2) -> Inteq(Card(Inter[st1;Complement st2]),Const 0)
  and siterm it = match it with
  | Intvar _ -> it
  | Const _ -> it
  | Plus its -> Plus (List.map siterm its)
  | Minus(it1,it2) -> Minus(siterm it1, siterm it2)
  | Times(k,it1) -> Times(k,siterm it1)
  | Card st -> it
  in sform f

  (* split f into quantifier sequence and body *)
let split_quants_body f =
  let rec vl f acc = match f with
  | Bind(b,id,f1) -> vl f1 ((b,id)::acc)
  | f -> (acc,f)
  in vl f []

  (* extract set variables from quantifier sequence *)
let extract_set_vars quants =
  List.map (fun (b,id) -> id)
    (List.filter (fun (b,id) -> (b=Forallset || b = Existsset))
       quants)

type partition = (ident * setTerm) list

  (* make canonical name for integer variable naming a cube *)
let make_name sts =
  let rec mk sts = match sts with
  | [] -> ""
  | (Setvar _)::sts1 -> "1" ^ mk sts1
  | (Complement (Setvar _))::sts1 -> "0" ^ mk sts1
  | _ -> failwith "make_name: unexpected partition form"
  in "l_" ^ mk sts

  (* make all cubes over vs *)
let generate_partition (vs : ident list) : partition =
  let add id ss = (Setvar id)::ss in
  let addc id ss = Complement (Setvar id)::ss in
  let add_set id inters =
    List.map (add id) inters @
    List.map (addc id) inters in
  let mk_nm is = (make_name is, Inter is) in
  List.map mk_nm
    (List.map List.rev
       (List.fold_right add_set vs [[]]))

  (* is the set term true in the set valuation
     -- reduces to propositional reasoning *)
let istrue (st:setTerm) (id,ivaluation) : bool =
  let valuation = match ivaluation with
  | Inter v -> v
  | _ -> failwith "wrong valuation" in
```

```
  let lookup v =
    if List.mem (Setvar v) valuation then true
    else if List.mem (Complement (Setvar v)) valuation then false
    else failwith "istrue: unbound var in valuation" in
  let rec check st = match st with
  | Setvar v -> lookup v
  | Emptyset -> false
  | Fullset -> true
  | Complement st1 -> not (check st1)
  | Union sts -> List.fold_right (fun st1 t -> check st1 || t) sts false
  | Inter sts -> List.fold_right (fun st1 t -> check st1 && t) sts true
  in check st

  (* compute cardinality of set expression
     as a sum of cardinalities of cubes *)
let get_sum (p:partition) (st:setTerm) : intTerm list =
  let get_list (id,inter) = match inter with
  | Inter ss -> ss
  | _ -> failwith "failed inv in get_sum"
  in
  List.map (fun (id,inter) -> Intvar id)
    (List.filter (istrue st) p)

  (* replace cardinalities of sets with sums of
     variables denoting cube cardinalities *)
let replace_cards (p:partition) (f:form) : form =
  let rec repl f = match f with
  | Not f -> Not (repl f)
  | And fs -> And (List.map repl fs)
  | Or fs -> Or (List.map repl fs)
  | Impl(f1,f2) -> Impl(repl f1,repl f2)
  | Bind(b,id,f1) -> Bind(b,id,repl f1)
  | Less(it1,it2) -> Less(irepl it1,irepl it2)
  | Inteq(it1,it2) -> Inteq(irepl it1,irepl it2)
  | Seteq(_,_)|Subseteq(_,_) -> failwith "failed inv in replace_cards"
  and irepl it = match it with
  | Intvar _ -> it
  | Const _ -> it
  | Plus its -> Plus (List.map irepl its)
  | Minus(it1,it2) -> Minus(irepl it1, irepl it2)
  | Times(k,it1) -> Times(k, irepl it1)
  | Card st -> Plus (get_sum p st)
  in repl f

let apply_quants quants f =
  List.fold_right (fun (b,id) f -> Bind(b,id,f)) quants f

let make_defining_eqns id part =
  let rec mk ps = match ps with
  | [] -> []
  | (id1,Inter (st1::sts1)) :: (id2,Inter (st2::sts2)) :: ps1
    when (st1=Setvar id && st2=Complement (Setvar id) && sts1=sts2) ->
      (Inter sts1,make_name sts1,id1,id2) :: mk ps1
  | _ -> failwith "make_triples: unexpected partition form" in
  let rename_last lss = match lss with
  | [(s,l,l1,l2)] -> [(s,maxcard,l1,l2)]
  | _ -> lss in
  rename_last (mk part)

  (* -------------------------- *)
  (* main loop of the algorithm *)
  (* -------------------------- *)
let rec eliminate_all quants part gr = match quants with
  | [] -> gr
  | (Existsint,id)::quants1 ->
      eliminate_all quants1 part (Bind(Existsint,id,gr))
  | (Forallint,id)::quants1 ->
      eliminate_all quants1 part (Bind(Forallint,id,gr))
  | (Existsnat,id)::quants1 ->
      eliminate_all quants1 part (Bind(Existsnat,id,gr))
  | (Forallnat,id)::quants1 ->
      eliminate_all quants1 part (Bind(Forallnat,id,gr))
  | (Existsset,id)::quants1 ->
      let eqns = make_defining_eqns id part in
      let newpart = List.map (fun (s,l',_,_) -> (l',s)) eqns in
      let mk_conj (_,l',l1,l2) = Inteq(Intvar l',Plus[Intvar l1;Intvar l2]) in
      let conjs = List.map mk_conj eqns in
      let lquants = List.map (fun (l,_) -> (Existsnat,l)) part in
      let gr1 = apply_quants lquants (And (conjs @ [gr])) in
      eliminate_all quants1 newpart gr1
  | (Forallset,id)::quants1 ->
      let eqns = make_defining_eqns id part in
      let newpart = List.map (fun (s,l',_,_) -> (l',s)) eqns in
      let mk_conj (_,l',l1,l2) = Inteq(Intvar l',Plus[Intvar l1;Intvar l2]) in
      let conjs = List.map mk_conj eqns in
      let lquants = List.map (fun (l,_) -> (Forallnat,l)) part in
      let gr1 = apply_quants lquants (Impl(And conjs, gr)) in
      eliminate_all quants1 newpart gr1

(* putting everything together *)

let alpha (f:form) : form =
  (* assumes f in prenex form *)
  let (quants,fm) = split_quants_body f in
  let fm1 = simplify_set_relations fm in
  let setvars = List.rev (extract_set_vars quants) in
  let part = generate_partition setvars in
  let g1 = replace_cards part fm1 in
  eliminate_all quants part g1
```