# Reasoning About Set Comprehensions*

Edmund S. L. Lam[1] and Iliano Cervesato[1]

Carnegie Mellon University
sllam@qatar.cmu.edu, iliano@cmu.edu

**Abstract**

Set comprehension is a mathematical notation for defining sets on the basis of a property of their members. Although set comprehension is widely used in mathematics and some programming languages, direct support for reasoning about it is still not readily available in state-of-the-art SMT solvers. This paper presents a technique for translating formulas which express constraints involving set comprehensions into first-order formulas that can be verified by off-the-shelf SMT solvers. More specifically, we have developed a lightweight Python library that extends the popular Z3 SMT solver with the ability to reason about the satisfiability of set comprehension patterns. This technique is general and can be deployed in a broad range of SMT solvers.

## 1 Introduction

Reasoning about sets is a frequently occurring exercise when conducting automated verification of programs and algorithms. While some recent work has focused on automated reasoning about set (and multiset) cardinality constraints [6, 5] and about arithmetic aggregate computations [4], little research has been conducted on developing automated tools for reasoning about explicit *set comprehensions*. Set comprehension is a mathematical notation for defining sets on the basis of a property of their members. Although set comprehension is widely used in mathematics and some programming languages, direct support for reasoning about set comprehension is still not readily available in state-of-the-art SMT solvers. In this work, we consider the task of verifying the satisfiability of quantifier-free formulas in the language of set comprehension over some standard theory $Th$, supported by typical modern SMT solvers. We denote the resulting language as $SC(Th)$. Examples of such base theory $Th$ include linear arithmetic over integers or real numbers, and user-defined theories over constructed data types (e.g., tuples, finite lists). In this paper, we will demonstrate our techniques on the theory of linear arithmetic over the integers ($LIA$). A typical problem in $SC(LIA)$ is given by the following equality:

$$\{10, 20, 30\} \doteq \{x * 10 \mid x < 4\}_{x \dot{\in} X}$$

It asks whether there are instances of the free variables in this formula (here, just the set variable $X$) so that the sets on the two sides are equal (we use $\doteq$ to denote set equality). The left-hand side is the extensional set $\{10, 20, 30\}$. The expression on the right-hand side is a set comprehension: it specifies the set of all the values of the expression $x*10$ where the values of $x$ are drawn from $X$ and are constrained by $x < 4$. The variable $x$ is bound in the comprehension. Set comprehensions determine sets intensionally on the basis of transformations such as $x * 10$ and guards like $x < 4$. A solution to the satisfiability problem for this equality is a model $\mathcal{S}$ such that

$$\mathcal{S} \models_{SC(LIA)} \{10, 20, 30\} \doteq \{x * 10 \mid x < 4\}_{x \dot{\in} X}$$

*This paper was made possible by grant NPRP 09-667-1-100, *Effective Programming for Large Distributed Ensembles*, from the Qatar National Research Fund (a member of the Qatar Foundation). The statements made herein are solely the responsibility of the authors.

In particular, the model $\mathcal{S}$ carries information on an acceptable value of the set variable $X$ — in this example, $X = \{1, 2, 3\}$ is the smallest solution, with $X = \{1, 2, 3, 4\}$ being another one.

Our goal is to reduce satisfiability in $SC(Th)$ to satisfiability in $Th$ extended with an uninterpreted sort for sets and an uninterpreted binary predicate $\dot{\in}$, which encodes set membership. To achieve this, we translate a set formula of interest $S$ in $SC(Th)$ into a formula $[\![S]\!]$ of this resulting theory, which we denote by $U{+}Th$ ($\underline{Th}$ with $\underline{U}$ninterpreted symbols). Specifically, each appearance of a set comprehension or of a standard set operation in $S$, is mapped to a fresh variable together with formulas in the language of $U{+}Th$. These formulas ensure that this variable is associated with the exact elements of the original set structure. Then, we verify that $S$ is satisfiable in $SC(Th)$ by checking that $[\![S]\!]$ has a model $\mathcal{M}$ in $U{+}Th$, i.e., that $\mathcal{M} \models_{U+Th} [\![S]\!]$. For a wide range of formulas, this verification step can be carried out using a state-of-the-art high-performance SMT solver such as Z3 [1]. Finally, we lift $\mathcal{M}$ to a model $\mathcal{S}$ of $S$, such that $\mathcal{S} \models_{SC(Th)} S$. Even though this operation is still undecidable in general [2], it is nonetheless decidable for many instances of $SC(LIA)$ formulas and can be implemented as an effective satisfiability test operation. For instance, our work in [3] uses a conservative implementation of this test operation to determine satisfiability of finite set comprehension formulas that would guarantee safety of certain compiler optimizations.

To illustrate this technique, here is the (slightly simplified) encoding $[\![S]\!]$ where $S$ is the formula from our earlier example:

$$[\![S]\!] \;=\; \begin{cases} \forall z.\ z \mathrel{\dot\in} X_2 \leftrightarrow z \mathrel{\dot\in} X_3 & -\ F_1 : X_2 = X_3 \\ \forall y.\ y \mathrel{\dot\in} X_2 \leftrightarrow (y \doteq 10 \vee y \doteq 20 \vee y \doteq 30) & -\ F_2 : X_2 = \{10, 20, 30\} \\ \forall x.\ (x * 10 \mathrel{\dot\in} X_3) \leftrightarrow (x \mathrel{\dot\in} X \wedge x < 4) & -\ F_3 : X_3 = \{x * 10 \mid x < 4\}_{x \mathrel{\dot\in} X} \end{cases}$$

The first formula, $F_1$, states that $X_2$ and $X_3$ are extensionally equal. The second formula, $F_2$, constrains the members of $X_2$ to be the integers 10, 20 or 30. Hence $X_2$ is a precise representation of the extensional set $\{10, 20, 30\}$ in $U{+}LIA$. The third formula, $F_3$, restricts $X_3$ so that, for each element $x$ in $X$ that is larger than 4, $x * 10$ must be a member of $X_3$. Hence $F_3$ constrains the behavior of $X_3$ to that of $\{x * 10 \mid x < 4\}_{x \mathrel{\dot\in} X}$ in $U{+}LIA$.

The rest of the paper is organized as follows: Section 2 introduces our source and target languages, $SC(LIA)$ and $U{+}LIA$. Section 3 defines the encoding of $SC(LIA)$ formulas into $U{+}LIA$ formulas. Section 4 presents a conservative satisfiability test operation for $SC(LIA)$ formulas. Section 5 describes our prototype implementation of this operation, a lightweight Python library that extends Z3. Section 6 discusses our conclusions and future work.

## 2 Notations, Term Languages and Models

In this section, we introduce meta-notation that we will use throughout this paper and we define the source and target term languages $SC(LIA)$ and $U{+}LIA$.

In discussions, we write $\bar{o}$ to denote finite sets of syntactic objects $o$, with $\emptyset$ for the empty set. We write $\{\bar{o}, o\}$ for the extension of set $\bar{o}$ with syntactic object $o$, omitting the brackets when no ambiguity arises. Meta-level set membership is denoted as $o \in \bar{o}$. We write meta-level set comprehension as $\{o : \bar{o} \mid \Phi(o)\}$, which represents the set containing all objects $o$ from $\bar{o}$ that satisfy the condition $\Phi(o)$. We write $[o'/x]o$ for the simultaneous replacement within object $o$ of all occurrences of variable $x$ with $o'$. When traversing binding constructs and quantifiers, substitutions implicitly $\alpha$-rename variables to avoid capture. These are all meta-level notations, not to be confused with the syntactic objects in our source language, which also features sets.

Figure 1 defines our source language $SC(LIA)$ and the target language $U{+}LIA$. Terms in $SC(LIA)$ are either arithmetic expressions, written $t$, or set expressions, denoted $s$. Arithmetic expressions correspond to integers and set expressions to sets of integers. An arithmetic expression is either a base variable $x$, a number $v$ in $\mathbb{Z}$, or a binary arithmetic operation $t\ op\ t$

Variables $x, X$ Values $v$

$SC(LIA)$: Set Comprehensions over Linear Integer Arithmetic

| | | |
|---|---|---|
| Arithmetic Term | $t$ | $::= x \mid v \mid t \ op \ t$ |
| Arithmetic Formula | $T$ | $::= t \doteq t \mid t < t \mid \neg T \mid T \wedge T$ |
| Set Term | $s$ | $::= X \mid \{\bar{t}\} \mid \{t \mid T\}_{x \dot{\in} s} \mid s \cup s \mid s \cap s \mid s \setminus s$ |
| Set Formula | $S$ | $::= t \dot{\in} s \mid s \doteq s \mid s \subseteq s \mid \neg S \mid S \wedge S$ |

$U$+$LIA$: Linear Integer Arithmetic and Uninterpreted Sets

| | | |
|---|---|---|
| Arithmetic Term | $t$ | $::= x \mid v \mid t \ op \ t$ |
| Arithmetic Formula | $T$ | $::= t \doteq t \mid t < t$ |
| Uninterpreted Set Term | $s$ | $::= X$ |
| Uninterpreted Set Formula | $S$ | $::= t \dot{\in} s$ |
| Formula | $F, C$ | $::= S \mid T \mid \neg F \mid F \wedge F \mid \exists x.F \mid \forall x.F$ |

Figure 1: Object Languages: $SC(LIA)$ and $U$+$LIA$

supported in the theory of linear integer arithmetic (e.g., + or $*$). A set expression is either a set variable $X$, an extensional set $\{\bar{t}\}$, the union, intersection or difference between two sets, or a set comprehension $\{t \mid T\}_{x \dot{\in} s}$. The empty set $\{\}$ is the special case of an extensional set $\{\bar{t}\}$ where $\bar{t}$ is empty. In a set comprehension, we refer to $t$ as the *comprehension pattern*, to $T$ as the *comprehension guard*, to $x$ as the *binding variable*, and to $s$ as the *comprehension domain*. The scope of the binding variable $x$ is $t$ and $T$. Where useful for clarity, we explicitly annotate this dependency by writing $t$ and $T$ as $t(x)$ and $T(x)$. Formulas $T$ that appear within set comprehensions are quantifier-free arithmetic formulas over the integer domain. Atomic arithmetic formulas include equality, written $t_1 \doteq t_2$, and other standard predicates, for example $t_1 < t_2$. The arithmetic formulas $T$ in comprehension guards combine them using the standard Boolean connectives — we display a minimal set consisting of $\neg$ and $\wedge$ in Figure 1 but will freely use the full set of Boolean connectives in our examples. The set formulas of $SC(LIA)$, denoted $S$, are the Boolean combination of set membership $t \dot{\in} s$, set equality $s_1 \doteq s_2$ and the subset relation $s_1 \subseteq s_2$.

The language of $U$+$LIA$ resembles $SC(LIA)$, with two major syntactic differences: first, it dispenses with all set expressions except for set variables $X$; second, $U$+$LIA$ formulas allow the use of quantifiers over base variables.

A model is a structure $\mathcal{M}$ that satisfies the axioms of a theory $Th$ and a formula $F$. This is denoted by $\mathcal{M} \models_{Th} F$. A model $\mathcal{M}$ contains mappings from the variables of $F$ to their respective instantiations that satisfy $F$ in $Th$. The lookup operation $\mathcal{M}(x)$ denotes the instantiated value that $x$ is mapped to. We write $\mathcal{M}_{x \mapsto v}$ for the model that maps $x$ to $v$ and is otherwise identical to $\mathcal{M}$. For each predicate symbol $p$ that appears in $Th$ or $F$, the model $\mathcal{M}$ also contains mappings from $p$ to the set of all valid instances of the predicate relation. The lookup operation $\mathcal{M}(p)$ denotes the set of valid relation instances of $p$. For simplicity, we assume predicates have the same arity throughout $Th$ and $F$. Interpretations of the (sub)formula(s) of $F$ and terms $t$ that appear in $F$ are obtained from $\mathcal{M}$ by the operations $\mathcal{M}[\![F]\!]$ and $\mathcal{M}[\![t]\!]$. Unsatisfiability of a formula $F$ is denoted by $\not\models_{Th} F$.

A model in our source language $SC(LIA)$ is denoted by $\mathcal{S}$ and its satisfiability judgment is denoted by $\mathcal{S} \models_{SC(LIA)} S$ where $S$ is an $SC(LIA)$ formula. We require that $SC(LIA)$ contain the theory of linear integer arithmetic and general set theory. A model $\mathcal{S}$ contains mappings for integer variables $x$ to integer values $v$ and set variables $X$ to extensional sets $\{\bar{t}\}$. The sets of all integers and sets that appear in $\mathcal{S}$ are denoted by $dom_{\mathbb{Z}}(\mathcal{S})$ and $dom_{\mathbb{S}}(\mathcal{S})$ respectively. We will

omit standard definitions of integer term interpretation $\mathcal{S}[\![t]\!]$ and integer formula satisfiability $\mathcal{S} \models_{SC(LIA)} T$. The interpretation $\mathcal{S}[\![s]\!]$ of set terms $s$ is defined as follows:

$$\begin{array}{rcl}
\mathcal{S}[\![X]\!] & = & \mathcal{S}(X) \\
\mathcal{S}[\![\{\bar{t}, t\}]\!] & = & \{\mathcal{S}[\![\bar{t}]\!], \mathcal{S}[\![t]\!]\} \\
\mathcal{S}[\![\{\}]\!] & = & \emptyset \\
\mathcal{S}[\![\{t(x) \mid T(x)\}_{x \dot{\in} s}]\!] & = & \{\mathcal{S}_{x \mapsto a}[\![t(x)]\!] : dom_{\mathbb{Z}}(\mathcal{S}) \mid a \in \mathcal{S}[\![s]\!] \text{ and } \mathcal{S}_{x \mapsto a} \models_{SC(LIA)} T(x)\} \\
\mathcal{S}[\![s_1 \cup s_2]\!] & = & \{a : dom_{\mathbb{Z}}(\mathcal{S}) \mid a \in \mathcal{S}[\![s_1]\!] \text{ or } a \in \mathcal{S}[\![s_2]\!]\} \\
\mathcal{S}[\![s_1 \cap s_2]\!] & = & \{a : dom_{\mathbb{Z}}(\mathcal{S}) \mid a \in \mathcal{S}[\![s_1]\!] \text{ and } a \in \mathcal{S}[\![s_2]\!]\} \\
\mathcal{S}[\![s_1 \setminus s_2]\!] & = & \{a : dom_{\mathbb{Z}}(\mathcal{S}) \mid a \in \mathcal{S}[\![s_1]\!] \text{ and } a \notin \mathcal{S}[\![s_2]\!]\}
\end{array}$$

This definition is inductive and makes use of $\mathcal{S}[\![t]\!]$ and $\mathcal{S} \models_{SC(LIA)} T$. The interpretation of a set comprehension $\mathcal{S}[\![\{t(x) \mid T(x)\}_{x \dot{\in} s}]\!]$ is defined as the set of all (and only) the interpretations of the comprehension pattern $t(x)$ with the binding variable $x$ mapped to the integers $a$ in the interpretation of the comprehension domain $s$ such that the corresponding instance of the comprehension guard $T(x)$ is satisfiable under $\mathcal{S}$. The other cases are standard.

Satisfiability for set formulas, $\mathcal{S} \models_{SC(LIA)} S$, is defined inductively over set formulas and on top of $\mathcal{S}[\![t]\!]$, $\mathcal{S}[\![S]\!]$ and $\mathcal{S} \models_{SC(LIA)} T$. The key cases of this definition are as follows:

$$\begin{array}{lll}
\mathcal{S} \models_{SC(LIA)} t \dot{\in} s & \textit{iff} & \mathcal{S}[\![t]\!] \in \mathcal{S}[\![s]\!] \\
\mathcal{S} \models_{SC(LIA)} s_1 \doteq s_2 & \textit{iff} & \text{for all } a \in dom_{\mathbb{Z}}(\mathcal{S}), \\
 & & \mathcal{S}_{x \mapsto a} \models_{SC(LIA)} x \dot{\in} s_1 \text{ iff } \mathcal{S}_{x \mapsto a} \models_{SC(LIA)} x \dot{\in} s_2 \\
\mathcal{S} \models_{SC(LIA)} s_1 \subseteq s_2 & \textit{iff} & \text{for all } a \in dom_{\mathbb{Z}}(\mathcal{S}), \\
 & & \mathcal{S}_{x \mapsto a} \models_{SC(LIA)} x \dot{\in} s_1 \text{ only if } \mathcal{S}_{x \mapsto a} \models_{SC(LIA)} x \dot{\in} s_2
\end{array}$$

We omitted the cases for the logical connectives whose interpretations are standard. Note that membership $\dot{\in}$ is an interpreted predicate symbol in *U+LIA*: $t \dot{\in} s$ is satisfiable in $\mathcal{S}$ if and only if the interpretation of $t$ is actually a member of the interpretation of $s$. Since $\dot{\in}$ is interpreted, $\mathcal{S}$ does not contain any mappings for $\dot{\in}$ (i.e., $\mathcal{S}(\dot{\in}) = \bot$). Satisfiability of equality, $s_1 \doteq s_2$ is defined in terms of the membership relation: $s_1$ and $s_2$ are equal if and only if they contain the same integers. In a similar manner, satisfiability of the subset relation $s_1 \subseteq s_2$ is defined in terms of membership $\dot{\in}$.

Models of our target language *U+LIA* are denoted by $\mathcal{M}$, and the satisfiability judgment is denoted by $\mathcal{M} \models_{U+LIA} F$. We require that *U+LIA* contains the theory of linear integer arithmetic and an uninterpreted domain for sets. Unlike $\mathcal{S}$, $\mathcal{M}$ contains no mappings for set variables, since the set domain is uninterpreted. However, $\mathcal{M}(\dot{\in})$ maps to the set of pairs $\langle \mathcal{M}[\![t]\!], X \rangle$ for every valid relation $t \dot{\in} X$ from $F$. The definition of $\mathcal{M}[\![t]\!]$ and $\mathcal{M} \models_{U+LIA} F$ are standard and therefore omitted.

# 3 Encoding $SC(LIA)$ into $U{+}LIA$

In this section, we define an encoding of $SC(LIA)$ terms and formulas into *U+LIA* terms and formulas. This encoding is given by two translation functions, the set term encoding $\llbracket \cdot \rrbracket^{set}$ and set formula encoding functions $\llbracket \cdot \rrbracket^{form}$.

The *set term encoding* of a set expression $s$ is given by a triple $(X, \mathcal{V}, C)$ that we write $\llbracket s \rrbracket^{term} = X \ \triangleright \ \mathcal{V} \ ; \ C$. Here, $X$ is a fresh variable associated with $s$ (unless $s$ is already a variable), $\mathcal{V}$ is a collection of uninterpreted set variables, and $C$ is a *U+LIA* formula. We refer to $X$ as the *representative variable* of $s$, $C$ as the *extensional context formula* and $\mathcal{V}$ as the *set variable signature*. The context formula $C$ constrains $X$ to capture the extensional behavior of $s$, while $\mathcal{V}$ contains all local set variables in $C$ created while encoding $s$. We require

$$
\begin{aligned}
[\![X]\!]^{set} &= X \ \rhd\ \emptyset \ ;\ \top \\
[\![\{\bar{t}\}]\!]^{set} &= \begin{cases} X \ \rhd\ \{X\} \ ;\ \forall x.\, x \mathrel{\dot\in} X \leftrightarrow mem(\bar{t}, x) \\ \quad \text{where} \quad mem(\{\bar{t}, t\}, x) \ = \ x \doteq [\![t]\!] \vee mem(\bar{t}, x) \\ \qquad\qquad\ mem(\emptyset, x) \ = \ \bot \end{cases} \\
[\![s_1 \ op \ s_2]\!]^{set} &= \begin{cases} X \ \rhd\ \mathcal{V}_1, \mathcal{V}_2, X \ ;\ C_1 \wedge C_2 \wedge constr(op) \\ \quad \text{where} \quad [\![s_1]\!]^{set} = xs_1 \ \rhd\ \mathcal{V}_1 \ ;\ C_1 \\ \qquad\qquad\ [\![s_2]\!]^{set} = xs_2 \ \rhd\ \mathcal{V}_2 \ ;\ C_2 \\ \qquad\qquad\ constr(\cup) = \forall x.\, x \mathrel{\dot\in} X \leftrightarrow x \mathrel{\dot\in} X_1 \vee x \mathrel{\dot\in} X_2 \\ \qquad\qquad\ constr(\cap) = \forall x.\, x \mathrel{\dot\in} X \leftrightarrow x \mathrel{\dot\in} X_1 \wedge x \mathrel{\dot\in} X_2 \\ \qquad\qquad\ constr(\setminus) = \forall x.\, x \mathrel{\dot\in} X \leftrightarrow x \mathrel{\dot\in} X_1 \wedge \neg(x \mathrel{\dot\in} X_2) \end{cases} \\
[\![\{t \mid T\}_{x \mathrel{\dot\in} s}]\!]^{set} &= \begin{cases} X \ \rhd\ \mathcal{V}, X \ ;\ C_{dom} \wedge C_{max} \wedge C_{rg} \\ \quad \text{where} \quad [\![s]\!]^{set} \ = \ X' \ \rhd\ \mathcal{V} \ ;\ C_{dom} \\ \qquad\qquad\ C_{max} \ = \ \forall x.\, (x \mathrel{\dot\in} X' \wedge [\![T]\!]) \rightarrow [\![t]\!] \mathrel{\dot\in} X \\ \qquad\qquad\ C_{rg} \ = \ \forall z.\, z \mathrel{\dot\in} X \rightarrow \exists x.\, (z \doteq [\![t]\!] \wedge x \mathrel{\dot\in} X' \wedge [\![T]\!]) \end{cases} \\
& \qquad\qquad [\![t]\!] = t \qquad\qquad [\![T]\!] = T
\end{aligned}
$$

Figure 2: Set Term Encoding into *U+LIA*: $[\![s]\!]^{set} = X \ \rhd\ \mathcal{V} \ ;\ C$

that $\mathcal{V}$ contain no duplicates and we rely on implicit $\alpha$-renaming to enforce this constraint. This provides an implicit mechanism to guarantee unique assignments of set variables during encoding.

Figure 2 defines the set term encoding operation. In the case of an extensional set $\bar{t} = \{t_1, \ldots, t_n\}$, the extensional context formula $\forall x.\, x \mathrel{\dot\in} X \leftrightarrow (x \doteq t_1 \vee \ldots \vee x \doteq t_n)$ constrains representative variable $X$ to contain all and only the elements that appear in $\bar{t}$. We encode all $SC(LIA)$ terms $t_i$ in $\bar{t}$ by means of the encoding operation $[\![t_i]\!]$. For the element terms we consider in this paper, it is sufficient to define their encoding as the identity function. The entry $[\![s_1 \ op \ s_2]\!]^{set}$ covers the cases for set union, intersection and difference. For each, we first derive encodings for $s_1$ and $s_2$ to obtain representative variables $X_1$ and $X_2$. The variable $X$ is assigned to represent the set $s_1 \ op \ s_2$. This is done with the help of the context formula $constr(op)$: If $op$ is the union operator $\cup$, the context formula state that any member $x$ of $X$ must be a member of either $X_1$ or $X_2$ and vice-versa. The cases where $op$ is $\cap$ and $\setminus$ are similar. Each of these context formulas constrains the extensional behavior $X$ to that of the union, intersection or set difference of $s_1$ and $s_2$ respectively. Finally, $[\![\{t \mid T\}_{x \mathrel{\dot\in} s}]\!]^{set}$ encodes the set comprehension as follows: The comprehension domain $s$ is encoded into $X'$ with the constraints of $s$ captured by $C_{dom}$. The comprehension $\{t \mid T\}_{x \mathrel{\dot\in} s}$ is encoded into $X$ with its constraints captured by context formulas $C_{max}$ and $C_{rg}$. The *comprehension maximality* condition $C_{max}$ imposes the constraint that every member $x$ of the comprehension domain $X'$ that satisfies the comprehension guard $T$ has its corresponding comprehension pattern $t$ as a member of $X$. The *comprehension range restriction* $C_{rg}$ imposes the constraint that for every member $z$ of the set comprehension $X$, there exists some member $x$ of the comprehension domain $X'$ whose corresponding comprehension pattern $t$ is equal to $z$ and satisfies the comprehension guard $T$.

While it is tempting to replace the context formulas $C_{max}$ and $C_{rg}$ with the single implication formula $C_{comp} = \forall x.\, (x \mathrel{\dot\in} X' \wedge T(x)) \leftrightarrow t(x) \mathrel{\dot\in} X$ (as done in our example in Section 1), $C_{comp}$ would not accurately capture the behavior of set comprehensions with expressions $t(x)$ that

$$
\begin{aligned}
[\![t \mathrel{\dot\in} s]\!]^{form} &= [\![t]\!] \mathrel{\dot\in} X \ \rhd\ \mathcal{V}\ ;\ C \qquad \text{where} \quad [\![s]\!]^{set} = X \ \rhd\ \mathcal{V}\ ;\ C \\
[\![s_1 \doteq s_2]\!]^{form} &= \begin{cases} \forall x.\ (x \mathrel{\dot\in} X_1) \leftrightarrow (x \mathrel{\dot\in} X_2) \ \rhd\ \mathcal{V}_1, \mathcal{V}_2\ ;\ C_1 \wedge C_2 \\ \quad \text{where} \quad [\![s_1]\!]^{set} = X_1 \ \rhd\ \mathcal{V}_1\ ;\ C_1 \\ \qquad\quad\ [\![s_2]\!]^{set} = X_2 \ \rhd\ \mathcal{V}_2\ ;\ C_2 \end{cases} \\
[\![s_1 \subseteq s_2]\!]^{form} &= \begin{cases} \forall x.\ (x \mathrel{\dot\in} X_1) \rightarrow (x \mathrel{\dot\in} X_2) \ \rhd\ \mathcal{V}_1, \mathcal{V}_2\ ;\ C_1 \wedge C_2 \\ \quad \text{where} \quad [\![s_1]\!]^{set} = X_1 \ \rhd\ \mathcal{V}_1\ ;\ C_1 \\ \qquad\quad\ [\![s_2]\!]^{set} = X_2 \ \rhd\ \mathcal{V}_2\ ;\ C_2 \end{cases} \\
[\![S_1 \wedge S_2]\!]^{form} &= \begin{cases} F_1 \wedge F_2 \ \rhd\ \mathcal{V}_1, \mathcal{V}_2\ ;\ C_1 \wedge C_2 \\ \quad \text{where} \quad [\![S_1]\!]^{form} = F_1 \ \rhd\ \mathcal{V}_1\ ;\ C_1 \\ \qquad\quad\ [\![S_2]\!]^{form} = F_2 \ \rhd\ \mathcal{V}_2\ ;\ C_2 \end{cases} \\
[\![\neg S]\!]^{form} &= \neg F \ \rhd\ \mathcal{V}\ ;\ C \qquad \text{where} \qquad [\![S]\!]^{form} = F \ \rhd\ \mathcal{V}\ ;\ C
\end{aligned}
$$

Figure 3: Set Formula Encoding into *U+LIA*: $[\![S]\!]^{form} = F \ \rhd\ \mathcal{V}\ ;\ C$

are not injective functions. For instance, consider the comprehension $s_1 = \{x\%3 \mid \top\}_{x \mathrel{\dot\in} s_2}$, where % is the modulus operator. Assuming that $X$ and $X'$ are the representatives of set comprehension $s_1$ and set term $s_2$ respectively, the formula $C_{comp} = \forall x.\, x \mathrel{\dot\in} X' \leftrightarrow (x\%3) \mathrel{\dot\in} X$ is incorrect in the '$\leftarrow$' case: it demands any $x$ such that $x\%3$ is a member of $X$ must be a member of $X'$, which is clearly not the behavior of the set comprehension. The combination of $C_{max}$ and $C_{rg}$ on the other hand, is sound as long as $t(x)$ is a total function.

Figure 3 defines the encoding of a set formula $S$, denoted by $[\![S]\!]^{form} = F \ \rhd\ \mathcal{V}\ ;\ C$. It encodes a $SC(LIA)$ formula $S$ into an *U+LIA* formula $F$ under the extensional context formula $C$ with the set variable signatures $\mathcal{V}$. We call $F$ the *main encoding formula*. Similarly to the term encoding operation, the formula $C$ specifies the constraints on uninterpreted set variables and $\mathcal{V}$ is the set of local set variables created by the encoding. The case $[\![t \mathrel{\dot\in} s]\!]^{form}$ encodes the membership relation by simply using the term encoding $[\![s]\!]^{set}$ of $s$. We then simply check that the encoding of $t$ is in the representative variable $X$ of $[\![s]\!]^{set}$. For $[\![s_1 \doteq s_2]\!]^{form}$, we encode $s_1$ and $s_2$ by applying the term encoding operation of Figure 3 to each, obtaining $X_1$ and $X_2$ as representatives of $s_1$ and $s_2$ respectively with extensional context $C_1$ and $C_2$. Following this, the equality constraint is represented in *U+LIA* as the formula $\forall x.\, x \mathrel{\dot\in} X_1 \leftrightarrow x \mathrel{\dot\in} X_2$, stating that all members of $X_1$ are members of $X_2$ and vice-versa. Encoding set equality in this manner exercises the axiom of extensionality of set theory, enforcing the extensional equality of sets with respect to the membership relation ($\mathrel{\dot\in}$). The case $[\![s_1 \subseteq s_2]\!]^{form}$ is similar, except that it is encoded into a one-sided extensional membership 'equality' ($\rightarrow$). The case $[\![S_1 \wedge S_2]\!]^{form}$ defines the cases for the logical connectives $\wedge$, during which we traverse the respective sub-formulas and terms containing set terms and derive their encodings. The main formula is simply the $\wedge$ logical connective applied to the respective encoded formulas.

The encoding of negation requires some care, and is the reason for separating the main encoding formula $F$ and the extensional context formula $C$. Our definition in Figure 3 encodes the negated $SC(LIA)$ formula $\neg(\{1\} \subseteq \{1,2\})$ as follows:

$$[\![\neg(\{1\} \subseteq \{1,2\})]\!]^{form} = \neg(\forall x.\, x \mathrel{\dot\in} X_1 \rightarrow x \mathrel{\dot\in} X_2) \ \rhd\ X_1, X_2\ ;\ X_1 = \{1\} \wedge X_2 = \{1,2\}$$

Here, we abbreviated the actual encoding of the extensional sets as $X_1 = \{1\}$ and $X_2 = \{1,2\}$ for to be succinct. The conjunction of the main formula and context formula (i.e., $\neg F \wedge C$) concisely captures the negated formula $\neg(\{1\} \subseteq \{1,2\})$, namely:

$$\neg(\{1\} \subseteq \{1,2\}) \quad \equiv \quad \neg(\forall x.\, x \mathrel{\dot\in} X_1 \rightarrow x \mathrel{\dot\in} X_2) \wedge X_1 = \{1\} \wedge X_2 = \{1,2\}$$

$$satCheck(S) = \begin{cases} \lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor & \text{if } \lceil\!\lceil S \rceil\!\rceil^{form} = F \ \rhd\ \mathcal{V}\ ;\ C \text{ and } \mathcal{M} \models_{U+LIA} C \wedge F \\ \bot & \text{otherwise} \end{cases}$$

$$\begin{array}{rrcl} \text{where} & \lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor(x) & = & \mathcal{M}(x) \\ & \lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor(X) & = & \begin{cases} \{a : dom_{\mathbb{Z}}(\mathcal{M}) \mid \langle a, X\rangle \in \mathcal{M}(\dot\in)\} & \text{if } X \notin \mathcal{V} \\ \bot & \text{otherwise} \end{cases} \\ & \lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor(\dot\in) & = & \bot \end{array}$$

Figure 4: Satisfiability Checking for $SC(LIA)$ Formulas

Assume we had instead collapsed $F$ and $C$ into a single formula component, so that $\lceil\!\lceil S \rceil\!\rceil^{form} = F \wedge C \rhd \mathcal{V}$. Then we would have to translate the above formula as

$$\neg((\forall x.\, x \mathrel{\dot\in} X_1 \rightarrow x \mathrel{\dot\in} X_2) \wedge X_1 \doteq \{1\} \wedge X_2 \doteq \{1,2\}) \rhd X_1, X_2$$

which is not the intended meaning of the negation as, propagating the negation,

$$\neg(\{1\} \subseteq \{1,2\}) \quad \not\equiv \quad \neg(\forall x.\, x \mathrel{\dot\in} X_1 \rightarrow x \mathrel{\dot\in} X_2) \vee X_1 \neq \{1\} \vee X_2 \neq \{1,2\}$$

# 4 Satisfiability Testing for $SC(LIA)$

In this section, we define a satisfiability test operation of $SC(LIA)$ formulas on the encoding defined in the previous section. This operation, denoted by $satCheck(S)$, is defined in Figure 4. If the $SC(LIA)$ set formula $S$ is satisfiable, this operation extracts a model for $S$ from a model of its encoding $\lceil\!\lceil S \rceil\!\rceil^{form}$ in *U+LIA*. Such a model of $\lceil\!\lceil S \rceil\!\rceil^{form}$ can be obtained using an off-the-shelf SMT solver that supports the built-in theory of linear integer arithmetic and uninterpreted domains. More specifically, given $\lceil\!\lceil S \rceil\!\rceil^{form} = F \ \rhd\ \mathcal{V}\ ;\ C$, the satisfiability of $S$ is determined by checking the satisfiability of its *U+LIA* encoding: i.e., $\mathcal{M} \models_{U+LIA} C \wedge F$. Decidability of this operation therefore depends on the decidability of $\models_{U+LIA}$. If $S$ is determined to be unsatisfiable, $satCheck(S)$ returns $\bot$. If $S$ is determined to be satisfiable, it returns a *decoded model* $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor$, which is an $SC(LIA)$ model inferred from $\mathcal{M}$. Since this operation is undecidable in general, it may not return a value at all for some formulas. The decode operator $\lfloor\!\lfloor \cdot \rfloor\!\rfloor$ is defined as follows: for integer variables $x$, $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor$ simply maps $x$ to $\mathcal{M}(x)$. For a set variable $X$ however, $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor(X)$ returns the set of all integers $a$ such that $\langle a, X\rangle \in \mathcal{M}(\dot\in)$. The membership relation of $\mathcal{M}$ is stripped away in $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor$, i.e., $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor(\dot\in) = \bot$.

Lemma 1 determines the soundness of the term encoding: for any set term $s$, we can extract a corresponding interpretation of $s$ from its encoding. Soundness for the formula encoding is given by Theorem 2. It states that for some $SC(LIA)$ formula $S$ with the encoding $\lceil\!\lceil S \rceil\!\rceil^{form} = F \ \rhd\ \mathcal{V}\ ;\ C$, if $S$ is satisfiable under $SC(LIA)$, then $C \wedge F$ is satisfiable under *U+LIA*.

**Lemma 1** (Soundness of the Term Encoding)**.** *Let $s$ be a $SC(LIA)$ set term and $\lceil\!\lceil s \rceil\!\rceil^{term} = X \ \rhd\ \mathcal{V}\ ;\ C$ its encoding in U+LIA. For any $\mathcal{S}$ that contains an interpretation of $s$ (i.e., $\mathcal{S}[\![s]\!]$), there exists $\mathcal{M}$ such that $\mathcal{M} \models_{U+LIA} C$ and $\mathcal{S} = \lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor$.*

*Proof.* The proof proceeds by structural induction on the encoding operation of the term $s$, with base cases $X$ and $\{\bar t\}$. For the base case $\{\bar t\}$, we can infer that $X$ contains all and only the values in $\bar t$ for the membership relation constraints ($x \mathrel{\dot\in} X$, such that $x \in \bar t$) imposed by the context formula of the encoding of $\{\bar t\}$ (i.e., $\mathcal{S}[\![s]\!] = \bar t$). Hence, by definition of $\lfloor\!\lfloor \cdot \rfloor\!\rfloor$, we show that $\mathcal{S} = \lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor$. For the inductive case $s_1\ op\ s_2$ and $\{t \mid T\}_{x \dot\in s}$, we similarly show $\mathcal{S} = \lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor$ by observing that their respective interpretations in $\mathcal{S}$ correspond to sets built from the membership relations ($x \mathrel{\dot\in} X$) captured by $C$. □

**Theorem 2** (Soundness of the Formula Encoding)**.** *Let $S$ be a $SC(LIA)$ formula $S$ and $\llbracket S \rrbracket^{form} = F \rhd \mathcal{V}\ ;\ C$ its encoding in U+LIA. For any $\mathcal{S}$ such that $\mathcal{S} \models_{SC(LIA)} S$, there exists $\mathcal{M}$ such that $\mathcal{M} \models_{U+LIA} C \wedge F$ and $\mathcal{S} = \lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor$.*

*Proof.* The proof proceeds by structural induction on the encoding operation of the formula $S$, with base cases $t \mathrel{\dot\in} s$, $s_1 \doteq s_2$ and $s_1 \subseteq s_2$, whose proofs follow from the soundness of term encoding (Lemma 1). □

Lemma 3 and Theorem 4 state the converse of Lemma 1 and Theorem 2 respectively, hence providing a completeness guarantee for our encoding.

**Lemma 3** (Completeness of the Term Encoding)**.** *Let $s$ be a $SC(LIA)$ set term and $\llbracket s \rrbracket^{term} = X \rhd \mathcal{V}\ ;\ C$ its encoding in U+LIA. For any $\mathcal{M}$ such that $\mathcal{M} \models_{U+LIA} C$, we have that $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor \models_{SC(LIA)} X \doteq s$.*

*Proof.* This proof is similar to that of Lemma 1. It proceeds by structural induction on the encoding of $s$. The main difference is that in each case, we show that interpretations of $s$ obtained from $\mathcal{S}$ (i.e., $\mathcal{S}(X)$) corresponds to that of obtained from $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor$, hence $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor = \mathcal{S}$. □

**Theorem 4** (Completeness of the Formula Encoding)**.** *Let $S$ be a $SC(LIA)$ formula and $\llbracket S \rrbracket^{form} = F \rhd \mathcal{V}\ ;\ C$ its encoding in U+LIA. For all $\mathcal{M}$ such that $\mathcal{M} \models_{U+LIA} C \wedge F$, we have that $\lfloor\!\lfloor \mathcal{M} \rfloor\!\rfloor \models_{SC(LIA)} S$.*

*Proof.* Like the proof of Theorem 2, this proof proceeds by structural induction on the encoding of formula $S$. The proof for the base cases follows from Lemma 3. □

Corollary 1 states the soundness and completeness of the satisfiability test operation *satCheck*, and it follows from properties 1, 2, 3 and 4.

**Corollary 1** (Soundness and Completeness of *satCheck*)**.** *For any $SC(LIA)$ formula $S$, we have the following:*

- *$satCheck(S) = \mathcal{S}$ if and only if $\mathcal{S} \models_{SC(LIA)} S$.*
- *$satCheck(S) = \bot$ if and only if $\not\models_{SC(LIA)} S$.*

# 5 Implementation

We implemented the above technique as a lightweight library with Z3 as the underlying SMT solver. This prototype is available for download at `https://github.com/sllam/pysetcomp`. Given an $SC(Th)$ formula, where $Th$ is some theory that the Z3 SMT solver supports, our Python library implements the encoding operation $\llbracket S \rrbracket^{form} = F \rhd \mathcal{V}\ ;\ C$. Formulas $F$ and $C$ are native Z3 formulas, hence $\mathcal{M} \models_{U+Th} F \wedge C$ is implemented by simply passing $F \wedge C$ to the Z3 satisfiability checking interfaces. Our prototype also includes a simple combinator library that provides the programmer with a convenient way to write $SC(Th)$ formulas.

The code that implement our example in Section 1 is as follows:

```
1  ## Initialize sorts and variables
2  I = z3.IntSort()
3  IntSet = mkSetSort( I )
4  x = z3.Int('x')
5  X = z3.Const('X', IntSet)
6
7  ## {10,20,30} ≐ { x*10 | x<4 }_{x ∈̇ X}
8  S = VSet(I,10,20,30) |Eq| Compre(x*10,x<4,x,X)
```

```
9
10  ## ⟦S⟧^form = F  ▷  V  ;  C
11  F, C, V = transForm( S )
12
13  ## M ⊨_U+Th F  ∧  C
14  s = z3.Solver()
15  s.add(z3.And([F]+C))
16  print s.check()     -- Prints 'sat'
```

Z3's built-in operations are explicitly prepended by `z3`. Lines 2–5 initializes the Z3 sorts (data domains) and variables that we need in this example: `I` abbreviates the Z3 integer sort, `mkSetSort(I)` returns a representation of the sort of sets of integer. In line 4, a Z3 integer variable `x` is declared, while in line 5 a Z3 variable of the sort of sets of integer is declared. Line 8 implements the actual $SC(LIA)$ formula: `VSet(I,10,20,30)` builds the extensional set $\{$`10,20,30`$\}$, while `Compre(x*10,x<4,x,X)` builds the set comprehension $\{x * 10 \mid x < 4\}_{x \dot{\in} X}$. The infix operator `Eq` corresponds to the equality predicate $\doteq$. Line 11 implements the encoding of `S` into *U+LIA* formulas `F` and `C`, with the set of local variables `V` created by the encoding procedure. Finally, lines 14–15 implements the satisfiability test by feeding `F` $\wedge$ `C` to the Z3 satisfiability checker.

Our prototype works on more than just integers. The following example involves tuples of integers.

$$X \doteq \{\langle 1,8\rangle, \langle 2,5\rangle, \langle 3,2\rangle, \langle 3,4\rangle, \langle 4,8\rangle\} \wedge Y \doteq \{x * 10 \mid x \leq 3\}_{\langle x,y\rangle \dot{\in} X} \wedge x * 10 \dot{\in} Y \wedge \langle x,y\rangle \dot{\in} X$$

The corresponding code in our implementation is as follows:

```
1   IntPair = mkTupleSort( I, I )
2   tup = IntPair.tup
3   pi1 = IntPair.pi1
4
5   PairSet = mkSetSort( IntPair )
6   IntSet = mkSetSort( I )
7
8   X = z3.Const('X', PairSet)
9   Y = z3.Const('Y', IntSet)
10  xy = z3.Const('xy', IntPair)
11  x, y = z3.Ints('x y')
12
13  cs = [X |Eq| VSet(IntPair, tup(1,8), tup(2,5), tup(3,2), tup(3,4), tup(4,8))
14        ,Y |Eq| Compre(pi1(xy)*10, pi1(xy)<=3, xy, X)
15        ,(x * 10) |In| Y
16        ,tup(x,y) |In| X]
```

In this example, the set comprehension $Y$ maps a set containing pairs of integers, into a set of integers, thus requiring our system to handle many-sorted encodings. Lines 1–3 define a new tuple sort of pairs of integers `IntPair`, with `tup` as its data constructor and `pi1` as a projection operator for the left tuple argument. Lines 5–6 declare new set sorts: `PairSet` is the sort of sets of integer pairs, while `IntSet` the sort of sets of integers. Lines 8–11 declare the variables of the respective sorts, while lines 13–16 define the actual formula. The infix binary relation `In` implements the membership relation $\dot{\in}$. Encoding `cs` with the `transForm` operation and feeding its output to the Z3 solver yields a satisfiable result, where a satisfiable instance of $x$ and $y$ can be extracted.

The implementation of the `In` operator is non-trivial and requires explanation. A call to `x |In| X` creates an instance of a Z3 function, named *mem*, that is interpreted by our encoding

as the binary relation $x \mathrel{\dot\in} X$ in Z3: this function maps the pair `x` and `X` to a Z3 Boolean value. Note that the two instances of `In` at lines 15–16 are of different sort: At line 15 we have an instance with the sort `Int*IntSet`, while at line 16 we have `IntPair*PairSet`. Since Z3 does not support parametric data types, our implementation must therefore map each instance to a different Z3 function symbol (here *mem_IntSet* and *mem_PairSet*) that are interpreted appropriately by our Python library. In order to provide a convenient interface that hides this mapping, we implement a lookup procedure inside the `In` operator that is akin to a simplified run-time version of type dictionary passing during type checking of Haskell type classes.

We have tested our implementation on a suite of set comprehension formulas of varying complexity. While most are rather small, in the future we intend to provide empirical results on more practical examples.

# 6 Conclusion and Future Work

In this paper, we reduced the satisfiability problem for formulas featuring comprehension and other set operations over a standard theory (linear integer arithmetic in our examples) to solving satisfiability constraints over this same theory augmented with a single uninterpreted sort. This technique allows the satisfiability of set-based formulas to be verified by a wide range of off-the-shelf SMT solvers that support the base theory. We have implemented a lightweight Python library that utilizes this encoding technique on the popular Z3 SMT solver. This implementation generalizes the encoding described here to a broad range of data types supported by Z3, which includes integer, real numbers, tuples, and finite lists.

In the future, we are interested in expanding our results to comprehensions over *multisets*. A commonly used representation for multisets relies on array maps, that associate each elements in the support domain to its multiplicity [5]. An adaptation of the technique presented in this paper that simply swaps our encoding of sets with this representation of multisets does not work however. To appreciate the added challenge, consider the task of verifying the following formula about multiset comprehensions:

$$M \quad = \quad X_1 \doteq \wr x\%3 \mid \top \int_{x \mathrel{\dot\in} X_2}$$

where $\wr \cdot \int$ delimits multisets in the same way as $\{\cdot\}$ delimited sets in the rest of the paper. In the suggested encoding, the variables $X_1$ and $X_2$ would be implemented as array maps: for every element $x$ in the domain, $X_1[x]$ gives us the number of times that $x$ occurs in $X_1$ (its multiplicity), and similarly for $X_2$. A naive adaptation of our technique suggests the following encoding for the above formula:

$$[\![M]\!] = \begin{cases} \forall x, m.\ (X_2[x] = m \wedge m > 0) \rightarrow X_1[x\%3] = m \\ \forall z, m.\ (X_1[z] = m \wedge m > 0) \rightarrow \exists x.(z = x\%3 \wedge X_2[x] = m) \end{cases}$$

Although it would be adequate for injective operations on $x$ (e.g., $x + 3$), this encoding fails for non-injective operations such as $x\%3$. For instance, the formula $M$ is satisfied for $X_1 = \wr 0, 0, 0, 2\int$ and $X_2 = \wr 3, 3, 6, 8\int$, yet the above encoding yields an unsatisfiable formula. The problem is that we have multiple elements in $X_2$ that are mapped to the same element in $X_1$. Since $3\%3 = 0$ and $X_2[3] = 2$, the above encoding entails that $X_1[0] = 2$. However, $6\%3 = 0$ also, which means that, given that $X_2[6] = 1$, we have that $X_1[0] = 1$ too — an absurdity. We would need to combine these results by adding the multiplicities of all $x$ that map to the same value in $X_1$: doing so for all $x$ in $X_2$ such that $x\%3 = 0$ in this example would yield the expected result, $X_1[0] = 3$. This has proved to be non-trivial in Z3. A possible approach to reasoning about such arithmetic aggregate operations could be adapted from [4].

# References


[1] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[2] Joseph Y. Halpern. Presburger Arithmetic with Unary Predicates is $\Pi_1^1$ Complete. *Journal of Symbolic Logic*, 56:56–2, 1991.

[3] Edmund S. L. Lam and Iliano Cervesato. Constraint Handling Rules with Multiset Comprehension Patterns. In *11th Workshop on Constraint Handling Rules*, 2014.

[4] K. R. M. Leino and R. Monahan. Reasoning about Comprehensions with First-Order SMT Solvers. In *In Proc. of the 2009 ACM symposium on Applied Computing*, pages 615–622. ACM, 2009.

[5] R. Piskac and V. Kuncak. MUNCH — Automated Reasoner for Sets and Multisets. In *IJCAR'10*, volume 6173 of *Lecture Notes in Computer Science*, pages 149–155. Springer, 2010.

[6] P. Suter, R. Steiger, and V. Kuncak. Sets with Cardinality Constraints in Satisfiability Modulo Theories. In *Verification, Model Checking, and Abstract Interpretation*, pages 403–418. Springer, 2011.