Solving Program Sketches with Large Integer Values

QINHEPING HU, University of Wisconsin—Madison
RISHABH SINGH, Google
LORIS D'ANTONI, University of Wisconsin—Madison

Program sketching is a program synthesis paradigm in which the programmer provides a partial program with holes and assertions. The goal of the synthesizer is to automatically find integer values for the holes so that the resulting program satisfies the assertions. The most popular sketching tool, Sketch, can efficiently solve complex program sketches but uses an integer encoding that often performs poorly if the sketched program manipulates large integer values. In this article, we propose a new solving technique that allows Sketch to handle large integer values while retaining its integer encoding. Our technique uses a result from number theory, the Chinese Remainder Theorem, to rewrite program sketches to only track the remainders of certain variable values with respect to several prime numbers. We prove that our transformation is sound and the encoding of the resulting programs are exponentially more succinct than existing Sketch encodings. We evaluate our technique on a variety of benchmarks manipulating large integer values. Our technique provides speedups against both existing Sketch solvers and can solve benchmarks that existing Sketch solvers cannot handle.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Theory of computation** → *Theory and algorithms for application domains;*

Additional Key Words and Phrases: Program synthesis, program sketching, chinese remainder theorem

ACM Reference format:
Qinheping Hu, Rishabh Singh, and Loris D'Antoni. 2022. Solving Program Sketches with Large Integer Values. *ACM Trans. Program. Lang. Syst.* 44, 2, Article 9 (July 2022), 28 pages.
https://doi.org/10.1145/3532849

1 INTRODUCTION

Program synthesis, the art of automatically generating programs that meet a user's intent, promises to increase the productivity of programmers by automating tedious, error-prone, and time-consuming tasks. **Syntax-guided Synthesis (SyGuS)** [2], where the search space of possible programs is defined using a grammar or a domain-specific language, has emerged as a common program synthesis paradigm for many synthesis domains. One of the earliest and successful syntax-guided program synthesis frameworks is program sketching [22], where (i) the search space of the

This work was supported, in part, by NSF under grants CNS-1763871, CCF-1750965, CCF-1744614, and CCF-1704117; and by the UW-Madison OVRGE with funding from WARF.
Authors' addresses: Q. Hu and L. D'Antoni, Department of Computer Sciences, University of Wisconsin Madison, 1210 West Dayton Street Madison, WI 53706-1685 USA; emails: qhu28@wisc.edu, ldantoni@wisc.edu; R. Singh, Google, 600 Amphitheatre Parkway Mountain View, CA 94043; email: rising@goggle.com.

0164-0925/2022/07-ART9 $15.00
https://doi.org/10.1145/3532849

synthesis problem is described using a partial program in which certain integer constants are left unspecified (represented as holes) and (ii) the specification is provided as a set of assertions describing the intended behavior of the program. The goal of the synthesizer is to automatically replace the holes in the program with integer values so that the resulting complete program satisfies all the assertions. Thanks to its simplicity, program sketching has found wide adoption in applications such as data-structure design [23], personalized education [21], program repair [7], and many others.

The most popular sketching tool, Sketch [24], can efficiently solve complex program sketches with hundreds of lines of code. However, Sketch often performs poorly if the sketched program manipulates large integer values. Sketch's synthesis is based on an algorithm called **counterexample-guided inductive synthesis (Cegis)** [24]. The Cegis algorithm iteratively considers a finite set I of inputs for the program and performs SAT queries to identify values for the holes so that the resulting program satisfies all the assertions for the inputs in I. Further SAT queries are then used to verify whether the generated solution is correct on all the possible inputs of the program. Sketch represents integers using a unary encoding (a variable for each integer value) so that arithmetic computations such as addition, multiplication, and so on, can be represented efficiently in the SAT formulas as lookup operations. This unary encoding, however, results in huge formulas for solving sketches with larger integer values as we also observe in our evaluation. Recently, a technique that extends the SAT solver with native integer variables and integer constraints was proposed to alleviate this issue in Sketch. It guesses values for the integer variables and propagates them through the integer constraints and learns from conflict clauses. However, this technique does not scale well when the sketches contain complex arithmetic operations, e.g., non-linear integer arithmetic.

In this article, we propose a program transformation technique that allows Sketch to solve program sketches involving large integer values while retaining the unary encoding used by the traditional Sketch solver. Our technique rewrites a Sketch program into an equivalent one that performs computations over smaller values. The technique is based on the well-known Chinese Remainder Theorem, which states that, given distinct prime numbers $p_1, \ldots, p_n$ such that $N = p_1 \cdot \ldots \cdot p_n$, for every two distinct numbers $0 \leq k_1, k_2 < N$, there exists a p_i such that $k_1 \bmod p_i \neq k_2 \bmod p_i$. Intuitively, this theorem states that tracking the modular values of a number smaller than N for each p_i is enough to uniquely recover the actual value of the number itself. We use this idea to replace a variable x in the program with n variables $x_{p_1}, \ldots, x_{p_n}$, so that for every i, $x_{p_i} = x \bmod p_i$. Using closure properties of modular arithmetic we show that, as long as the program uses the operators $+, -, *, ==$, tracking the modular values of variables and performing the corresponding operations on such values is enough to ensure correctness. For example, to reflect the variable assignment $x = y + z$, we perform the assignment $x_{p_i} = (y_{p_i} + z_{p_i}) \bmod p_i$, for every p_i. Similarly, the Boolean operation $x == y$ will only hold if $x_{p_i} = y_{p_i}$ for every p_i. To identify what variables and values in the program can be rewritten, we develop a dataflow analysis that computes what variables *may* flow into operations that are not sound in modular arithmetic, e.g., $<, >, \leq$, and $/$.

We provide a comprehensive theoretical analysis of the complexity of the proposed transformation. First, we derive how many prime numbers are needed to track values in a certain integer range. Second, we analyze the number of bits required to encode values in the original and rewritten program and show that, for the unary encoding used by Sketch, our technique offers an exponential saving in the number of required bits. We also present an incremental algorithm that lazily increases the number of primes (and therefore the integer range) used in the modular semantics.

We evaluate our technique on 181 benchmarks from various applications of program sketching. Our results show that our technique results in significant speedups over existing SKETCH solvers and is able to solve 48 benchmarks on which SKETCH times out.

Contributions. In summary, our contributions are as follows:

- A language IMP-MOD together with a *modular semantics* that represents integer values using their remainders for a given set of primes and a proof that this semantics is equivalent to the standard *integer semantics* (Section 4).
- A dataflow analysis for detecting variables that can be soundly executed in the modular semantics and an algorithm for translating IMP programs into IMP-MOD ones (Section 5).
- A synthesis algorithm for IMP-MOD programs and incremental synthesis algorithm that lazily increases the number of primes used in the modular semantics (Section 6).
- A complexity analysis that shows that synthesis for IMP-MOD programs requires exponentially smaller SAT queries than synthesis in IMP (Section 7).
- An evaluation of our technique on 181 benchmarks that manipulate large integer values. Our solver outperforms the default SKETCH unary solver, it can solve 48 new benchmarks that no SKETCH solver can solve, and it is 15.9× faster than the SKETCH NATIVE-INTS integer solver on the hard benchmarks that take more than 10 seconds to solve (Section 8).

This article is an extended version with proofs and additional examples of the short version of the article with the same title published at ESOP20 [17],

2 MOTIVATING EXAMPLE

In this section, we use a simple example to illustrate our technique and its effectiveness. Consider the SKETCH program `polyArray` presented in Figure 1(b). The goal of this synthesis problem is to synthesize a two-variable quadratic polynomial (lines 7 and 8) whose evaluation `p` on given inputs `x` and `y` is equal to a given expected-output array `z` (line 9),

```
assert p[i] == z[i];
```

Solving the problem amounts to finding non-negative integer values for the holes (??) and sign values, i.e., −1 or 1, for the holes ($??^s$) such that the assertion becomes true.[1] In this case, a possible solution is the polynomial:

```
p[i] = -17*y[i]^2-8*x[i]*y[i]-17*x[i]^2-3*x[i];
```

When attempting to solve this problem, the SKETCH synthesizer times out at 300 seconds. To solve this problem, SKETCH creates SAT queries where the variables are the holes. Due to the large numbers involved in the computation of this program (including intermediate expression computations), the unary encoding of SKETCH ends up with SAT formulas with approximately 45 *million* clauses.

Sketch Program with Modular Arithmetic. The technique we propose in this article has the goal of reducing the complexity of the synthesis problem by transforming the program into an equivalent one that manipulates smaller integer values and that yields easier SAT queries. Given the SKETCH program in Figure 1(b), our technique produces the modified SKETCH program `pAPrime` in Figure 1(a). The new SKETCH program has the same control flow graph as the original one, but instead of computing the actual values of the expressions `x[·]` and `y[·]`, it tracks their remainders for the set of prime numbers $\{2, 3, 5, 7, 11, 13, 17\}$ using new variables, e.g., `x2[i]` tracks the remainder of `x[i]` modulo 2.

[1] In SKETCH, holes can only assume positive values. This is why we need the sign holes, which are implemented using regular holes as follows: `if(??) then 1 else -1`.

```
1 // n=4, x=[24,-1,0,-19], y=[-7,11,-3,13]
2 // z=[-9353,-1983,-153,-6977]
3 polyArray(int n, int[n] x, int[n] y, int[n] z){
4   int[n] p;
5   int i=0;
6   while (i<n){
7     p[i]=??^s_1*??_1*y[i]^2+??^s_2*??_2*x[i]^2+??^s_3*??_3*x[i]*y[i]
8          +??^s_4*??_4*y[i]+??^s_5*??_5*x[i]+??^s_6*??_6;
9    assert p[i] == z[i];
10   i++; }
11 }
```

(a) Original sketch program polyArray.

```
1 // n=4, x=[24,-1,0,-19], y=[-7,11,-3,13]
2 // z=[-9353,-1983,-153,-6977]
3 pAPrime(int n, int[n] x, int[n] y, int[n] z){
4   int[n] x2,x3,x5,x7,x11,x13,x17;
5   while (i<n){   // Initialize modular variables
6     x2[i]=x[i]%2;
7     x3[i]=x[i]%3;
8     ... i++; }
9   int i=0;
10  int[n] p2,p3,p5,p7,p11,p13,p17;
11  while (i<n){
12    p2[i]=(??^s_1*(??_1%2)*(y2[i]^2%2)%2
13           +??^s_2*(??_2%2)*(x2[i]^2%2)%2
14           +??^s_3*(??_3%2)*(x2[i]%2)*(y2[i]%2)%2
15           +??^s_4*(??_4%2)*(y2[i]%2)%2
16           +??^s_5*(??_5%2)*(x2[i]%2)%2
17           +??^s_6*(??_6%2)%2)%2;
18    ...
19    assert p2[i] = z2[i];
20    assert p3[i] = z3[i];
21    ...
22    i++; }
23 }
```

(b) Rewritten sketch program pAPrime.

Fig. 1. Sketch program (a) and rewritten version with values tracked for different moduli (b).

The program pAPrime initializes the modular variables with the corresponding modular values (lines 5–8). When rewriting a computation over modular variables, the same computation is performed modularly (lines 12–17). For example, the term $??^s_1 * ??_1$*y[i]^2 when tracked modulo 2 is rewritten as

$$(??^s_1*(??_1\%2)*((y2[i]\%2)^2\%2))\%2$$

In the rewritten program, the variables i and n are not tracked modularly, since such a transformation would incorrectly access array indices. Finally, the assertions for different moduli share the same holes as the solution to the Sketch has to be correct for all modular values. In the rest of the article, we develop a dataflow analysis that detects when variables can be tracked modularly.

Sketch can solve the rewritten program in less than 2 seconds and produce hole values that are correct solutions for the original program. This speedup is due to the small integer values

$$
\begin{array}{rcl}
\text{Arith Expr } a & := & ?? \mid c \mid v \mid a_0\ op_a\ a_1 \\
\text{Arith Op } op_a & := & + \mid - \mid * \mid / \\
\text{Bool Expr } b & := & \texttt{not}\ b \mid a_0\ op_c\ a_1 \mid b_0\ \texttt{and}\ b_1 \\
\text{Comp Op } op_c & := & == \mid < \mid > \mid \leq \mid \geq \\
\text{Stmt } s & := & v = a \mid s_0; s_1 \mid \texttt{while}(b)\ \{s\} \\
 & \mid & \texttt{if}(b)\ s_0\ \texttt{else}\ s_1 \mid \texttt{assert}\ b \\
\text{Program } P & := & f(v_1, \cdots, v_n, ??_1, \ldots ??_m)\ \{s\}
\end{array}
$$

Fig. 2. The syntax for a simple imperative language IMP with integer hole values.

$$
[\![??]\!]^{\texttt{IMP}}_{\sigma,\sigma_H} := \sigma_H(??) \qquad [\![v]\!]^{\texttt{IMP}}_{\sigma,\sigma_H} := \sigma(v)
$$

$$
\begin{array}{rcl}
[\![c]\!]^{\texttt{IMP}}_{\sigma,\sigma_H} & := & c \\
[\![a_1\ op_a\ a_2]\!]^{\texttt{IMP}}_{\sigma,\sigma_H} & := & [\![a_1]\!]^{\texttt{IMP}}_{\sigma,\sigma_H}\ op_a\ [\![a_2]\!]^{\texttt{IMP}}_{\sigma,\sigma_H} \\
[\![v = a]\!]^{\texttt{IMP}}_{\sigma,\sigma_H} & := & (\sigma[v \mapsto [\![a]\!]^{\texttt{IMP}}_{\sigma,\sigma_H}],\ \sigma_H) \\
[\![\texttt{assert}\ b]\!]^{\texttt{IMP}}_{\sigma,\sigma_H} & := & \begin{cases} \bot & [\![b]\!]^{\texttt{IMP}}_{\sigma,\sigma_H} = \texttt{false} \\ (\sigma,\ \sigma_H) & [\![b]\!]^{\texttt{IMP}}_{\sigma,\sigma_H} = \texttt{true} \end{cases} \\
[\![s_0; s_1]\!]^{\texttt{IMP}}_{\sigma,\sigma_H} & := & [\![s_1]\!]^{\texttt{IMP}}_{\sigma',\sigma'_H}\ \text{s.t.}\ [\![s_0]\!]^{\texttt{IMP}}_{\sigma,\sigma_H} = (\sigma',\ \sigma'_H)
\end{array}
$$

Fig. 3. Semantics of IMP. Valuations σ and σ_H assign integer values to variables and holes, respectively.

manipulated by the modular computations. In fact, the intermediate SAT formulas generated by SKETCH for the program pAPrime have approximately 120 *thousand* clauses instead of the 45 *million* clauses for polyArray. Due to the complex arithmetic in the formulas, even if SKETCH uses the SMT-like native integer encoding, it still requires more than 300 seconds to solve this problem.

While this technique is quite powerful, it does have some limitations. In particular, the solution to the rewritten SKETCH is guaranteed to be a correct solution only for inputs that cause intermediate values of the program to be in a range $[d_1, d_2]$ such that $d_2 - d_1 \leq 2 \times 3 \times 5 \times 7 \times 11 \times 13 \times 17 = 510,510$. We will prove this result in Section 4.

3 PRELIMINARIES

In this section, we describe the IMP language that we will consider throughout the article and briefly recall the counter-example guided inductive synthesis algorithm employed by the SKETCH solver.

3.1 IMP Language for Sketching

For simplicity, we consider a simple imperative language IMP with integer holes for defining the hypothesis space of programs (Figure 2).[2] Without loss of generality, we assume the programs consists of a single program $f(v_1, \ldots, v_n, ??_1, \ldots ??_m)$ with n integer variables and m integer holes. The body of the program f consists of a sequence of statements, where a statement s can either be a variable assignment, a while loop statement, an if conditional statement, or an assert statement. For simplicity, we assume the function has no local variables. The holes ?? denote integer constant values that are unknown and the goal of the synthesis process is to compute these values such that a set of desired program assertions are satisfied for every possible input values to f. The semantics of IMP is standard (some of the rules are shown in Figure 3).

[2]Our implementation also supports for-loops, recursion, C-style arrays (i.e., non-infinite arrays), and complex types.

Example 3.1. An example IMP sketch denoting a partial program is shown below.

```
triple(n,h,??){ h=??; assert h*n==n+n+n; }
```

The goal of the synthesizer is to compute the value of the hole ?? such that the assertion is true for all possible input values of n and h. For this example, ?? = 3 is a valid solution.

3.2 Solving IMP Sketches

The SKETCH solver uses the counter-example guided inductive synthesis algorithm (CEGIS) to find hole values such that the desired assertions hold for all input values. Formally, the SKETCH synthesizer solves the following constraint:

$$\exists \vec{??} \equiv (??_1, \ldots, ??_m) \in \mathbb{Z}^m. \ \forall in \in \mathcal{I}. \ [\![f(in, \vec{??})]\!]^{\texttt{IMP}} \neq \bot,$$

where $\mathbb{Z}$ denotes the domain of all integer values, $\vec{??}$ denotes the list of unknown hole values $(??_1, \ldots, ??_m) \in \mathbb{Z}^m$, $\mathcal{I}$ denotes the domain of all input argument values to the function f, and $[\![f(in, \vec{??})]\!]^{\texttt{IMP}} \neq \bot$ denotes that the program satisfies all assertions. The synthesis problem is in general undecidable for a language with complex operations such as the IMP language because of the infinite size of possible hole and input values. To make the synthesis process more tractable, SKETCH imposes a bound on the sizes of both the input domain ($\mathcal{I}_b$) and the domain of hole values ($\mathbb{Z}_b$) to obtain the following constraint:

$$\exists \vec{??} \equiv (??_1, \ldots, ??_m) \in \mathbb{Z}_b^m. \ \forall in \in \mathcal{I}_b. \ [\![f(in, \vec{??})]\!]^{\texttt{IMP}} \neq \bot.$$

The bounded domains make the synthesis problem decidable, but the formula with nested quantifiers results in a search space of hole values that is still huge for any reasonable bounds. To solve such bounded equations efficiently, SKETCH uses the CEGIS algorithm to incrementally add inputs from the domain until obtaining hole values $\vec{??}$ that satisfy the assertion predicates for all the input values in the bounded domain. The algorithm solves the formula with two quantifiers by iteratively solving a series of first-order queries with a single quantifier. It first encodes the existential query (synthesis query) over a randomly selected input value in_0 to find the hole values $\vec{H}$ that satisfy the predicate for in_0 using a SAT solver in the backend,

$$\exists \vec{??} \equiv (??_1, \ldots, ??_m) \in \mathbb{Z}_b^m. \ [\![f(in_0, \vec{??})]\!]^{\texttt{IMP}} \neq \bot.$$

It then encodes another existential query (verification) to now find a counter-example in_1 for which the predicate is not satisfied for the previously found hole values,

$$\exists in \in \mathcal{I}_b. \ \neg [\![f(in, \vec{H})]\!]^{\texttt{IMP}} \neq \bot.$$

If no counter-example input can be found, then the hole values are returned as the desired solution. Otherwise, the algorithm computes a new hole value that satisfies the assertion for all the counter-example inputs found so far. This process continues iteratively until either a desired hole value is found (i.e., no counter-example input exists), no satisfiable hole value is found (i.e., the synthesis problem is infeasible), or the SAT solver times out.

Integer Encoding. The SKETCH solver can efficiently solve the synthesis constraint in many domains, but it does not scale well for sketches manipulating large numbers. SKETCH uses a unary encoding to represent integers, where the encoded formula consists of a variable for each integer value. The unary encoding allows for simplifying the representation of complex non-linear arithmetic operations. For example, a multiplication operation can be represented as simply a lookup table using this encoding. In practice, the unary encoding is efficient for many practical problems. However, this also results in huge SAT formulas in presence of large integers. Recently, a new technique based on extending the SAT solver with native integer variables and constraints was

proposed to alleviate this issue in SKETCH (flag `--slv-nativeints`). Similarly to the Boolean variables, this technique extends the solver to also guess integer values and propagates them in the constraints while also learning from conflict clauses. Note that SKETCH uses these SAT extensions and encodings instead of an SMT solver. While it is possible in principle to implement a SKETCH solver that directly uses an SMT solver (e.g., using the theory of bit-vectors), SKETCH uses these encodings for more efficient solving of constraints especially for non-linear arithmetic, where the unary encoding may instead produce a large number of clauses. Our new technique for handling computations over large numbers still maintains the efficient unary encoding of integers and computations over them while improving scalability for problems involving large integer values.

4 MODULAR ARITHMETIC SEMANTICS

In this section, we present the language IMP-MOD in which variables can be tracked using modular arithmetic. We start by recalling the Chinese Remainder Theorem, then define both a modular and integer semantics for the IMP-MOD language, and show that the two semantics are equivalent.

4.1 The Chinese Remainder Theorem

The Chinese Remainder Theorem is a powerful number theory result that shows the following: Given a set of distinct primes $\mathbb{P} = \{p_1, \ldots, p_k\}$, any number n in an interval of size $p_1 \cdot \ldots \cdot p_k$ can be uniquely identified from the remainders $[n \bmod p_1, \ldots, n \bmod p_k]$. In Section 4.2, we will use this idea to define the semantics of the IMP-MOD language. The main benefit of this idea is that the remainders could be much smaller than actual program values.

Example 4.1. For $\mathbb{P} = [3, 5, 7]$ and an integer 101, its remainders $[2, 1, 3]$ are much smaller than 101. However, any number of the form $101 + 105 \times n$ also has remainders $[2, 1, 3]$ with respect to the same prime set.

In general, one cannot uniquely determine an arbitrary integer value from its remainders for some set $\mathbb{P}$, i.e., the mapping from a number to its remainders is an abstraction in the sense of abstract interpretation [6]. However, if we are interested in a limited range of integer values $[L, U)$, then one can choose a set of primes $\mathbb{P} = \{p_1, \ldots, p_k\}$ such that, for values $L \leq x < U$, the map $[r_1, \ldots, r_k] \mapsto x$, where $x \equiv r_i \bmod p_i$, is an injection.

THEOREM 4.2 (CHINESE REMAINDER THEOREM [4]). *Let p_1, ..., p_k be positive integers that are pairwise co-prime, i.e., no two numbers share a divisor larger than 1. Denote $N = \prod_{i=1}^{k} p_i$, and let d, r_1, r_2, ..., r_k be any integers. Then there is one and only one integer $d \leq x < d + N$ such that $x \equiv r_i \bmod p_i$ for every $1 \leq i \leq k$.*

We define the translation function $m_{\mathbb{P}}(x) := [x \bmod p_i, \ldots, x \bmod p_k]$ that maps an integer to its set of remainders with respect to $\mathbb{P}$. The following corollary follows from Theorem 4.2.

COROLLARY 4.3. *Let $\mathbb{P} = [p_1, \ldots, p_k]$ be a set of distinct primes such that $p_1 \cdot \ldots \cdot p_k = N$. For every integer d, the map $m_{\mathbb{P}}(x) : [d, d + N) \to [0, p_1) \times \cdots \times [0, p_k)$ is a bijection.*

In Section 4.2, we will use Corollary 4.3 to relate the semantics of IMP and IMP-MOD. When $m_{\mathbb{P}}(x)$ is bijective on some set R, we denote with $m_{\mathbb{P}}^{-1,R} : [0, p_1) \times \cdots \times [0, p_k) \to R$ its inverse function.

Example 4.4. Let x be a integer in the range $[0, 105)$ (note that $105 = 3 \times 5 \times 7$). If we know that the value of x is congruent to $[2, 1, 3]$ modulo $\{3, 5, 7\}$, then we can uniquely identify the value of x to be 101 by observing that $101 \equiv 2 \bmod 3$, $101 \equiv 1 \bmod 5$, and $101 \equiv 3 \bmod 7$.

The following lemma shows that the function $m_{\mathbb{P}}$ is closed under addition, subtraction, and multiplication of integers.

$$
\begin{array}{rcl}
\text{Modular Expr } a^{\mathbb{P}} & := & c^{\mathbb{P}} \mid v^{\mathbb{P}} \mid a_1^{\mathbb{P}}\ op_a^{\mathbb{P}}\ a_2^{\mathbb{P}} \mid \textsc{toPrime}(a) \\
\text{Modular Op } op_a^{\mathbb{P}} & := & + \mid - \mid * \\
\\
\text{Arith Expr } a & := & ?? \mid c \mid v \mid a_1\ op_a\ a_2 \\
\text{Arith Op } op_a & := & + \mid - \mid * \mid / \\
\text{Bool Expr } b & := & \texttt{not}\ b \mid a_1\ op_c\ a_2 \mid b_1\ \texttt{and}\ b_2 \mid b_1\ \texttt{or}\ b_2 \mid a_1^{\mathbb{P}} == a_2^{\mathbb{P}} \\
\text{Comp Op } op_c & := & < \mid > \mid \leq \mid \geq \\
\\
\text{Stmt } s & := & v = a \mid v^{\mathbb{P}} = a^{\mathbb{P}} \mid s_1; s_2 \\
 & \mid & \texttt{while}(b)\ \{s\} \mid \texttt{if}(b)\ s_1\ \texttt{else}\ s_2 \mid \texttt{assert}\ b \\
\text{Program } P & := & f(v_1, \cdots, v_n, v_1^{\mathbb{P}}, \cdots, v_m^{\mathbb{P}}, ??_1, \ldots, ??_l)\ \{s\}
\end{array}
$$

Fig. 4. Syntax of the IMP-MOD language.

Lemma 4.5. *For every set of primes $\mathbb{P}$, integers x and y, and $op \in \{+, -, *\}$, the following holds:* $m_{\mathbb{P}}(x\ op\ y) = m_{\mathbb{P}}(x)\ op\ m_{\mathbb{P}}(y)$.

4.2 The IMP-MOD Language

In this section, we define the IMP-MOD language (syntax in Figure 4), a variant of the IMP language for which the semantics can be defined using modular arithmetic.[3] An IMP-MOD program is parametric on a set $\mathbb{P} = \{p_1, \ldots, p_k\}$ of distinct prime numbers. The structure of an IMP-MOD program is similar to an IMP program, but IMP-MOD supports two types of variables and arithmetic expressions: the regular IMP ones (i.e., v, a, and b), which operate over an integer semantics, and the modular ones (i.e., $v^{\mathbb{P}}$, $a^{\mathbb{P}}$, and $b^{\mathbb{P}}$), which take as an additional parameter the set of primes $\mathbb{P}$ and operate over a modular semantics. The semantics of some of the key constructs of IMP-MOD is shown in Figure 5.

The key idea of the modular semantics is that the value of each program variable in $v^{\mathbb{P}}$ and arithmetic expressions in $a^{\mathbb{P}}$ is denoted by a tuple of values, one for each prime number $p_i \in \mathbb{P}$. For example, the value of the constant $c^{\mathbb{P}}$ is represented by the tuple $[c \bmod p_1, \ldots, c \bmod p_k]$, where each individual value denotes the remainder of c when divided by the prime number $p_i \in \mathbb{P}$. Formally, the program f has two sets of variables $V^{\mathbb{Z}} = \{v_1, \ldots, v_n\}$ and $V^{\mathbb{P}} = \{v_1^{\mathbb{P}}, \ldots, v_m^{\mathbb{P}}\}$, which contain all the integer and prime variables, respectively, and a set of holes $H = \{??_1, \ldots, ??_k\}$. The denotation function, uses two valuation functions: (i) $\sigma : V^{\mathbb{Z}} \cup H \to \mathbb{Z}$, which maps variables and holes to integer values, (ii) $\sigma^{\mathbb{P}} : V^{\mathbb{P}} \to [0, p_1) \times \cdots \times [0, p_k)$, which maps primed variables to modular values. The expression $\textsc{toPrime}(a)$ converts the integer value of an integer expression a to a modular tuple. Arithmetic expressions in $a^{\mathbb{P}}$ are computed using modular values with the result being obtained using modular arithmetic with respect to the corresponding primes in $\mathbb{P}$. Note that the only comparison operator allowed over modular expressions is == and that the division operator *cannot* be applied to modular expressions. While the syntax does not directly allow for holes to be represented modularly (i.e., we do not have expressions of the form $??^{\mathbb{P}}$) an expression of the form $\textsc{toPrime}(??)$ effectively achieves the objective of representing a hole ?? modularly.

4.3 Equivalence between the Two Semantics

Next, we provide an alternative integer semantics, which applies the IMP integer semantics to modular expressions and show that, under some assumptions on the magnitude of the values

[3]We consider the simple subset for a clear presentation of the semantics, but our framework works for the full IMP language (and for more complex language constructs) as we will see in the later sections.

$$
\begin{array}{rcl}
[\![\textsc{toPrime}(a)]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} & := & [\ [\![a]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} \bmod p_1, \cdots\ ] \\
[\![v^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} & := & \sigma^{\mathbb{P}}(v) \\
[\![c^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} & := & [\ c \bmod p_1, \cdots, c \bmod p_k\ ] \\
[\![a_1^{\mathbb{P}}\ op_a^{\mathbb{P}}\ a_2^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} & := & [\ (x_1^1\ op_a^{\mathbb{P}}\ x_1^2) \bmod p_1, \cdots\ ] \text{ where } [\![a_i^{\mathbb{P}}]\!]^{\mathbb{P}} = [\ x_1^i, \cdots, x_k^i\ ] \\
[\![\texttt{not}\ b]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} & := & \neg[\![b]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} \\
[\![a_1\ op_c\ a_2]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} & := & [\![a_1]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}}\ op_c\ [\![a_2]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} \\
[\![b_1\ \texttt{and}\ a_2]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} & := & [\![a_1]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} \wedge [\![a_2]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} \\
[\![a_1\ \texttt{or}\ a_2]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} & := & [\![a_1]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} \vee [\![a_2]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} \\
[\![a_1^{\mathbb{P}} == a_2^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} & := & x_1^1 == x_1^2 \wedge \cdots \wedge x_k^1 == x_k^2 \text{ where } [\![a_i^{\mathbb{P}}]\!]^{\mathbb{P}} = [\ x_1^i, \cdots, x_k^i\ ] \\
[\![c]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} & := & c \\
[\![v]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} & := & \sigma(v) \\
[\![a_1\ op_a\ a_2]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} & := & [\![a_1]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}}\ op_a\ [\![a_2]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} \\
[\![v = a]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} & := & (\sigma[v \hookleftarrow [\![a]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}}], \sigma^{\mathbb{P}}) \\
[\![v^{\mathbb{P}} = a^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}} & := & (\sigma, \sigma^{\mathbb{P}}[v^{\mathbb{P}} \hookleftarrow [\![a^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}}])
\end{array}
$$

Fig. 5. Modular semantics.

$$
\begin{array}{rcl}
[\![\textsc{toPrime}(a)]\!]_{\sigma_1,\sigma_2} & := & [\![a]\!]_{\sigma_1,\sigma_2} \\
[\![v^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2} & := & \sigma_2(v^{\mathbb{P}}) \\
[\![c^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2} & := & c \\
[\![a_1^{\mathbb{P}}\ op_a^{\mathbb{P}}\ a_2^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2} & := & [\![a_1^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}\ op_a\ [\![a_2^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2} \\
[\![\texttt{not}\ b]\!]_{\sigma_1,\sigma_2} & := & \neg[\![b]\!]_{\sigma_1,\sigma_2} \\
[\![a_1\ op_c\ a_2]\!]_{\sigma_1,\sigma_2} & := & [\![a_1]\!]_{\sigma_1,\sigma_2}\ op_c\ [\![a_2]\!]_{\sigma_1,\sigma_2} \\
[\![b_1\ \texttt{and}\ a_2]\!]_{\sigma_1,\sigma_2} & := & [\![a_1]\!]_{\sigma_1,\sigma_2} \wedge [\![a_2]\!]_{\sigma_1,\sigma_2}
\end{array}
\qquad
\begin{array}{rcl}
[\![a_1\ \texttt{or}\ a_2]\!]_{\sigma_1,\sigma_2} & := & [\![a_1]\!]_{\sigma_1,\sigma_2} \vee [\![a_2]\!]_{\sigma_1,\sigma_2} \\
[\![a_1^{\mathbb{P}} == a_2^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2} & := & [\![a_1^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2} == [\![a_2^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2} \\
[\![c]\!]_{\sigma_1,\sigma_2} & := & c \\
[\![v]\!]_{\sigma_1,\sigma_2} & := & \sigma_1(v) \\
[\![a_1\ op_a\ a_2]\!]_{\sigma_1,\sigma_2} & := & [\![a_1]\!]_{\sigma_1,\sigma_2}\ op_a\ [\![a_2]\!]_{\sigma_1,\sigma_2} \\
[\![v = a]\!]_{\sigma_1,\sigma_2} & := & (\sigma_1[v \hookleftarrow [\![a]\!]_{\sigma_1,\sigma_2}], \sigma_2) \\
[\![v^{\mathbb{P}} = a^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2} & := & (\sigma_1, \sigma_2[v^{\mathbb{P}} \hookleftarrow [\![a^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}])
\end{array}
$$

Fig. 6. Integer semantics.

manipulated by the program, the modular and integer semantics are equivalent. We will use this result to build our modified synthesis algorithm.

Integer Semantics. The *integer semantics* of IMP-MOD is shown in Figure 6 (denoted $[\![\cdot]\!]_{\sigma_1,\sigma_2}$). In this semantics, modular expressions are evaluated as integer expressions using the same semantics as for IMP, i.e., the values of modular variables and modular arithmetic expressions are denoted by integer values. Therefore, in the integer semantics, we use two valuation functions $\sigma_1 : V^{\mathbb{Z}} \cup H \mapsto \mathbb{Z}$ mapping variables and holes to integers and $\sigma_2 : V^{\mathbb{P}} \mapsto \mathbb{Z}$ mapping modular variables to integers.

Relation between the Two Semantics. We now show that the modular semantics is, in some sense, equivalent to the integer semantics. For the rest of this section, we fix a set of distinct primes $\mathbb{P} = \{p_1, \ldots, p_k\}$.

To prove the equivalence of the two program semantics, we will require the values of modular expressions to lie in some range that is covered by the prime numbers in $\mathbb{P}$. The following definition captures this restriction.

Definition 4.6. Given a modular arithmetic expression $a^{\mathbb{P}}$ (respectively, Boolean expression b) and some integers $L < U$, we say $a^{\mathbb{P}}$ with context (σ_1, σ_2) is *uniformly in the range* $R := [L, U)$ $-a^{\mathbb{P}} \in_{\sigma_1,\sigma_2} R$ for short if, under the integer semantics, all evaluation of modular subexpressions of $a^{\mathbb{P}}$ (respectively, b) are in the range R,

- $a^{\mathbb{P}} \in_{\sigma_1,\sigma_2} R$, iff $[\![a^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2} \in R$;
- $a_1^{\mathbb{P}} == a_2^{\mathbb{P}} \in_{\sigma_1,\sigma_2} R$, iff $a_1^{\mathbb{P}} \in_{\sigma_1,\sigma_2} R$, $a_2^{\mathbb{P}} \in_{\sigma_1,\sigma_2} R$;
- b_1 and $b_2 \in_{\sigma_1,\sigma_2} R$, iff $b_1 \in_{\sigma_1,\sigma_2} R$, $b_2 \in_{\sigma_1,\sigma_2} R$;

- b_1 or $b_2 \in_{\sigma_1,\sigma_2} R$, iff $b_1 \in_{\sigma_1,\sigma_2} R$, $b_2 \in_{\sigma_1,\sigma_2} R$;
- not $b \in_{\sigma_1,\sigma_2} R$, iff $b \in_{\sigma_1,\sigma_2} R$;
- $a_1\ op_c\ a_2 \in_{\sigma_1,\sigma_2} R$ for any arithmetic expressions a_1, a_2 and operator op_c.

Given a valuation function $\sigma : V^{\mathbb{P}} \mapsto \mathbb{Z}$, we write $m_{\mathbb{P}} \circ \sigma$ to denote the modular valuation obtained by applying the $m_{\mathbb{P}}$ function to σ, i.e., for every $v^{\mathbb{P}} \in V^{\mathbb{P}}$, $(m_{\mathbb{P}} \circ \sigma)(v^{\mathbb{P}}) = m_{\mathbb{P}}(\sigma(v^{\mathbb{P}}))$. Similarly, for a modular valuation function $\sigma^{\mathbb{P}} : V^{\mathbb{P}} \to [0, p_1) \times \cdots [0, p_k)$, we denote $m_{\mathbb{P}}^{-1,R} \circ \sigma^{\mathbb{P}}$ the integer valuation from $V^{\mathbb{P}}$ to R such that, for every $v^{\mathbb{P}} \in V^{\mathbb{P}}$, $(m_{\mathbb{P}}^{-1,R} \circ \sigma^{\mathbb{P}})(v^{\mathbb{P}}) = m_{\mathbb{P}}^{-1,R}(\sigma^{\mathbb{P}}(v^{\mathbb{P}}))$. The following lemma shows that, when the values of modular arithmetic expressions lay in an interval of size $N = p_1 \cdot \ldots \cdot p_k$ the modular and integer semantics of modular arithmetic expressions are equivalent. In practice, the assumption that numbers need to lie in a given range is not a limitation, since one can choose enough prime numbers to cover any practical range. We will show in Section 8 how the choice of prime numbers affects the efficiency of our technique.

Lemma 4.7. *Given a set of primes $\mathbb{P} = \{p_1, \ldots, p_k\}$, an arithmetic expression $a^{\mathbb{P}}$, and two valuation functions $\sigma_1 : V^{\mathbb{Z}} \cup H \mapsto \mathbb{Z}$ and $\sigma_2 : V^{\mathbb{P}} \mapsto \mathbb{Z}$, we have*

$$m_{\mathbb{P}}([\![a^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}) = [\![a^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}.$$

Moreover, if there exists an interval R of size $N = p_1 \cdot \ldots \cdot p_k$ such that $a^{\mathbb{P}} \in_{\sigma_1,\sigma_2} R$, then

$$m_{\mathbb{P}}^{-1,R}([\![a^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}) = [\![a^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}.$$

Proof. We prove this lemma by induction on $a^{\mathbb{P}}$.

- If $a^{\mathbb{P}} = c^{\mathbb{P}}$, then we have $m_{\mathbb{P}}([\![c^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}) = m_{\mathbb{P}}(c)$
  $= [c \bmod p_1, \ldots, c \bmod p_k] = [\![c^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}$.
- If $a^{\mathbb{P}} = v^{\mathbb{P}}$, then we have $m_{\mathbb{P}}([\![v^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}) = m_{\mathbb{P}}(\sigma_2(v^{\mathbb{P}})) = (m_{\mathbb{P}} \circ \sigma_2)(v^{\mathbb{P}}) = [\![v^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}$.
- If $a^{\mathbb{P}} = a_1^{\mathbb{P}}\ op_a^{\mathbb{P}}\ a_2^{\mathbb{P}}$, then we have from the induction hypothesis that $m_{\mathbb{P}}([\![a_1^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}) = [\![a_1^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}$ and $m_{\mathbb{P}}([\![a_2^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}) = [\![a_2^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}$. Then

$$\begin{aligned}
m_{\mathbb{P}}([\![a_1^{\mathbb{P}}\ op_a^{\mathbb{P}}\ a_2^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}) &= m_{\mathbb{P}}([\![a_1^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}\ op_a^{\mathbb{P}}\ [\![a_2^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}) \\
\text{(Lemma.4.5)} &= m_{\mathbb{P}}([\![a_1^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2})\ op_a^{\mathbb{P}}\ m_{\mathbb{P}}([\![a_2^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}) \\
\text{(Induction hyp.)} &= [\![a_1^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}\ op_a^{\mathbb{P}}\ [\![a_2^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2} \\
\text{(Def. of } [\![\cdot]\!]^{\mathbb{P}}_{\sigma,\sigma^{\mathbb{P}}}) &= [\![a_1^{\mathbb{P}}\ op_a^{\mathbb{P}}\ a_2^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2} \\
&= [\![e^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}.
\end{aligned}$$

- If $a^{\mathbb{P}} = \textsc{toPrime}(a_1)$, then we have

$$\begin{aligned}
m_{\mathbb{P}}([\![\textsc{toPrime}(a_1)]\!]_{\sigma_1,\sigma_2}) &= m_{\mathbb{P}}([\![a_1]\!]_{\sigma_1,\sigma_2}) \\
&= [[\![a_1]\!]_{\sigma_1,\sigma_2} \bmod p_1, \ldots] \\
&= [[\![a_1]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2} \bmod p_1, \ldots] (*) \\
&= [\![\textsc{toPrime}(a_1)]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2},
\end{aligned}$$

  where the deduction (*) follows from the fact that, on an integer expression a_1, the semantics $[\![a_1]\!]$ and $[\![a_1]\!]^{\mathbb{P}}$ are identical and are only affected by σ_1.

For the second part of the lemma, according to Corollary 4.3 and the assumption that $a^{\mathbb{P}} \in_{\sigma_1,\sigma_2} R \Rightarrow [\![a^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2} \in R$, the function $m_{\mathbb{P}}$ is a bijection from R to $[0, p_1) \times \cdots \times [0, p_k)$. Hence, $m_{\mathbb{P}}^{-1,R}([\![a^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}) = [\![a^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}$ follows directly from $m_{\mathbb{P}}([\![a^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}) = [\![a^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}$. □

Similarly, we show that the two semantics are also equivalent for Boolean expressions.

Lemma 4.8. *Given a set of primes $\mathbb{P} = \{p_1, \ldots, p_k\}$, an interval R of size $N = p_1 \cdot \ldots \cdot p_k$, a Boolean expression b, and two valuation functions $\sigma_1 : V^{\mathbb{Z}} \cup H \mapsto \mathbb{Z}$ and $\sigma_2 : V^{\mathbb{P}} \mapsto \mathbb{Z}$, if $b \in_{\sigma_1,\sigma_2} R$, then $[\![b]\!]_{\sigma_1,\sigma_2} = [\![b]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}$.*

Proof. We prove this lemma by induction on b.

- If $b = (a_1^{\mathbb{P}} == a_2^{\mathbb{P}})$, then we have
$$[\![b]\!]_{\sigma_1,\sigma_2} = ([\![a_1^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2} == [\![a_2^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2})$$
and
$$[\![b]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2} = ([\![a_1^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2} == [\![a_2^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}).$$
According to Lemma. 4.7 and the assumption that $b \in_{\sigma_1,\sigma_2} R \Rightarrow a_1^{\mathbb{P}}, a_2^{\mathbb{P}} \in_{\sigma_1,\sigma_2} R$, the function $m_{\mathbb{P}}$ is bijective over R and the following implication holds:
$$\begin{aligned} [\![a_1^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2} == [\![a_2^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2} \quad &\Leftrightarrow \quad m_{\mathbb{P}}([\![a_1^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}) == m_{\mathbb{P}}([\![a_2^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}) \\ &\Leftrightarrow \quad [\![a_1^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2} == [\![a_2^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}. \end{aligned}$$
- If $b = b_1$ or b_2, then we assume by induction that $[\![b_1]\!]_{\sigma_1,\sigma_2} = [\![b_1]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}$ and $[\![b_2]\!]_{\sigma_1,\sigma_2} = [\![b_2]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}$. Then
$$\begin{aligned} [\![b_1 \text{ or } b_2]\!]_{\sigma_1,\sigma_2} \quad &\Leftrightarrow \quad ([\![b_1]\!]_{\sigma_1,\sigma_2} \text{ or } [\![b_2]\!]_{\sigma_1,\sigma_2}) \\ &\Leftrightarrow \quad \left([\![b_1]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2} \text{ or } [\![b_2]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}\right) \\ &\Leftrightarrow \quad [\![b_1 \text{ or } b_2]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}. \end{aligned}$$
- The case of and and not is similar as the previous case.
- If $b = a_0\ op_c\ a_1$, then $[\![a_0\ op_c\ a_1]\!]_{\sigma_1,\sigma_2} = [\![a_0\ op_c\ a_1]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}$, since a_0 and a_1 are not modular expressions and the two semantics are therefore identical. □

We are now ready to show the equivalence between the modular semantics and the integer semantics for programs $P \in$ IMP-MOD. The semantics of a program $P = f(V^{\mathbb{Z}}, V^{\mathbb{P}}, H)\ \{s\}$ is a map from valuations to valuations, i.e., given a valuation $\sigma_1 : V^{\mathbb{Z}} \to \mathbb{Z}$ for integer variables, a valuation $\sigma_2 : V^{\mathbb{P}} \to \mathbb{Z}$ for modular variables and a valuation $\sigma^H : H \to \mathbb{Z}$ for holes, we have $[\![P]\!](\sigma_1, \sigma_2, \sigma^H) = [\![s]\!]_{\sigma_1 \cup \sigma^H, \sigma_2}$ and $[\![P]\!]^{\mathbb{P}}(\sigma_1, \sigma_2, \sigma^H) = [\![s]\!]^{\mathbb{P}}_{\sigma_1 \cup \sigma^H, m_{\mathbb{P}}\circ\sigma_2}$. Therefore, it is sufficient to show that the two semantics are equivalent for any statement s.

The two semantics are equivalent for a statement s if, under the same input valuations, the resulting valuations of the semantics can be translated to each other. Formally, given valuations σ_1, σ_2 and an interval R of size N, we say $[\![s]\!]_{\sigma_1,\sigma_2} \equiv_{\mathbb{P}} [\![s]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2}$ iff $\sigma_1' = \sigma_1''$, $m_{\mathbb{P}} \circ \sigma_2' = \sigma_2^{\mathbb{P}}$ and $\sigma_2' = m_{\mathbb{P}}^{-1,R} \circ \sigma_2^{\mathbb{P}}$ where $[\![s]\!]_{\sigma_1,\sigma_2} = (\sigma_1', \sigma_2')$ and $[\![s]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}}\circ\sigma_2} = (\sigma_1'', \sigma_2^{\mathbb{P}})$.

We define uniform inclusion for statements.

Definition 4.9. Given a set of primes $\mathbb{P}$, two integers $L < U$ and a statement s, we say s with context (σ_1, σ_2) is *uniformly in the range* $R := [L, U) — s \in_{\sigma_1,\sigma_2} R$ for short if under the integer semantics, all evaluation of modular subexpressions of s are in the range R:

- $(v^{\mathbb{P}} = a^{\mathbb{P}}) \in_{\sigma_1,\sigma_2} R$ iff $a^{\mathbb{P}} \in_{\sigma_1,\sigma_2} R$.
- while$(b)\{s\} \in_{\sigma_1,\sigma_2} R$ iff $s \in_{\sigma_1,\sigma_2} R$ and $b \in_{\sigma_1,\sigma_2} R$.
- $s_1; s_2 \in_{\sigma_1,\sigma_2} R$ iff $s_1 \in_{\sigma_1,\sigma_2} R$ and $s_2 \in_{\sigma_1,\sigma_2} R$.
- if(b) s_1 else $s_2 \in_{\sigma_1,\sigma_2} R$ iff $s_1 \in_{\sigma_1,\sigma_2} R$, $s_2 \in_{\sigma_1,\sigma_2} R$ and $b \in_{\sigma_1,\sigma_2} R$.
- assert $b \in_{\sigma_1,\sigma_2} R$ iff $b \in_{\sigma_1,\sigma_2} R$.

At last, the two semantics are equivalent for statements.

Theorem 4.10. *Given a set of primes $\mathbb{P} = [p_1, \ldots, p_k]$, a statement s and two valuation functions $\sigma_1 : V^{\mathbb{Z}} \cup H \to \mathbb{Z}$ and $\sigma_2 : V^{\mathbb{P}} \to \mathbb{Z}$, if there exists an interval R of size N such that $s \in_{\sigma_1,\sigma_2} R$, then $[\![s]\!]_{\sigma_1,\sigma_2} \equiv_{\mathbb{P}} [\![s]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}} \circ \sigma_2}$.*

Proof. We prove this theorem by induction on statements s. In the following, we let $[\![s]\!]_{\sigma_1,\sigma_2} = (\sigma_1', \sigma_2')$ and $[\![s]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}} \circ \sigma_2} = (\sigma_1'', \sigma_2^{\mathbb{P}})$.

- If $s = (v^{\mathbb{P}} = a^{\mathbb{P}})$, then we have that $[\![v^{\mathbb{P}} = a^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2} = (\sigma_1, \sigma_2[v^{\mathbb{P}} \hookleftarrow [\![a^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}])$ and $[\![v^{\mathbb{P}} = a^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}} \circ \sigma_2} = (\sigma_1, (m_{\mathbb{P}} \circ \sigma_2)[v^{\mathbb{P}} \hookleftarrow [\![a^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}} \circ \sigma_2}])$. To show $m_{\mathbb{P}} \circ \sigma_2' = \sigma_2^{\mathbb{P}}$, we need to show that
$$m_{\mathbb{P}} \circ (\sigma_2[v^{\mathbb{P}} \hookleftarrow [\![a^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}]) = (m_{\mathbb{P}} \circ \sigma_2)[v^{\mathbb{P}} \hookleftarrow [\![a^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}} \circ \sigma_2}].$$
In fact, for any primed variable $u^{\mathbb{P}} \neq v^{\mathbb{P}}$, both sides in the above equation will return $(m_{\mathbb{P}} \circ \sigma_2)(u^{\mathbb{P}})$. For the primed variable $v^{\mathbb{P}}$, we have
$$left\ side = m_{\mathbb{P}}([\![a^{\mathbb{P}}]\!]_{\sigma_1,\sigma_2}) = [\![a^{\mathbb{P}}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}} \circ \sigma_2} = right\ side,$$
where the middle equation is induced from Lemma. 4.7 with the assumption $s \in_{\sigma_1,\sigma_2} R \Rightarrow a^{\mathbb{P}} \in_{\sigma_1,\sigma_2} R$.
- If $s = \texttt{while}(b)\{s_1\}$, then we have by induction that $[\![s_1]\!]_{\sigma_3,\sigma_4} \equiv_{\mathbb{P}} [\![s_1]\!]^{\mathbb{P}}_{\sigma_3, m_{\mathbb{P}} \circ \sigma_4}$ for any valuation σ_3 and σ_4. Deduced from Lemma. 4.8 and the assumption that $s \in_{\sigma_1,\sigma_2} R \Rightarrow b \in_{\sigma_1,\sigma_2} R$, we know that the Boolean values $[\![b]\!]_{\sigma_1,\sigma_2} = [\![b]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}} \circ \sigma_2}$. If both $[\![b]\!]_{\sigma_1,\sigma_2}$ and $[\![b]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}} \circ \sigma_2}$ are false, then
$$\begin{aligned}[\![\texttt{while(false)}\{s_1\}]\!]_{\sigma_1,\sigma_2} &= (\sigma_1, \sigma_2) \equiv_{\mathbb{P}} (\sigma_1, m_{\mathbb{P}} \circ \sigma_2) \\ &= [\![\texttt{while(false)}\{s_1\}]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}} \circ \sigma_2}.\end{aligned}$$
So it is left to show the true-branch $[\![s_1]\!]_{\sigma_1,\sigma_2} \equiv_{\mathbb{P}} [\![s_1]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}} \circ \sigma_2}$, which is exactly the IH.
- If $s = s_1; s_2$, then we assume by induction that $[\![s_1]\!]_{\sigma_1,\sigma_2} \equiv_{\mathbb{P}} [\![s_1]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}} \circ \sigma_2}$ and $[\![s_2]\!]_{\sigma_3,\sigma_4} \equiv_{\mathbb{P}} [\![s_2]\!]^{\mathbb{P}}_{\sigma_3, m_{\mathbb{P}} \circ \sigma_4}$ for any valuations σ_3 and σ_4. Then
$$\begin{aligned}[\![s_1; s_2]\!]_{\sigma_1,\sigma_2} = [\![s_2]\!]_{\sigma_3',\sigma_4'} &\equiv_{\mathbb{P}} [\![s_2]\!]^{\mathbb{P}}_{\sigma_3', m_{\mathbb{P}} \circ \sigma_4'} \\ &= [\![s_1; s_2]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}} \circ \sigma_2},\end{aligned}$$
where $[\![s_1]\!]_{\sigma_1,\sigma_2} = (\sigma_3', \sigma_4') \equiv_{\mathbb{P}} (\sigma_3', m_{\mathbb{P}} \circ \sigma_4') = [\![s_1]\!]^{\mathbb{P}}_{\sigma_1, m_{\mathbb{P}} \circ \sigma_2}$ is implied by IH.
- If $s = \texttt{assert}\ b$, then it is similar to the case of while-statements. Since the two semantics for Boolean expressions are equivalent, the two semantics for s will choose the same branch that is either $\perp$ or unchanged.
- If $s = \texttt{if}\ s_1\ \texttt{else}\ s_2$, then it is also similar to the case of while-statements.
- If $s = (v = a)$, then the two semantics are equivalent, since there are no modular expressions in s. □

5 FROM IMP TO IMP-MOD PROGRAMS

In this section, we develop a dataflow analysis for detecting variables in IMP programs for which it is sound to track values modularly. We then use this dataflow analysis to rewrite an IMP program to an equivalent IMP-MOD program.

5.1 Dataflow Analysis

The formalization of IMP-MOD in Section 4.2 made it clear that the modular semantics is only appropriate when integer values are manipulated using addition, multiplication, subtraction, and equality. Other operations like division and less-than comparison cannot be computed soundly in modular arithmetic.

ALGORITHM 1: Returns variables that should be tracked using modular/integer semantics.

```
/* f: sketched function, V^P variables to be tracked modularly, V^Z variables to be
   tracked with integer values                                                    */
function DataFlowAnalysis(f)
    S ← {/, <, >, ≤, ≥}; V^Z ← ∅
    for op ∈ S do
        /* Compute all variables v that may flow into op                          */
        V^Z ← V^Z ∪ Dataflow(op, f)
    V^P ← V \ V^Z
    return (V^Z, V^P)
```

Example 5.1. Consider an integer variable x with modular value x_2 under modulus 2 and x_3 under modulus 3 and an integer variable y with modular value y_2, y_3 under corresponding moduli. Then the assignment of $x = y + y$; implies $x_2 = (y_2 + y_2) \mod 2$; and $x_3 = (y_3 + y_3) \mod 3$. However, $x = x/y$; does not imply $x_2 = (x_2/y_2) \mod 2$; and $x_3 = (x_3/y_3) \mod 3$.

Dataflow Analysis for Partitioning Variables. We now define a dataflow analysis for computing which variables in a program must be tracked with the integer semantics (i.e., the set $V^{\mathbb{Z}}$) and which variables can be soundly tracked using the modular semantics (i.e., the set $V^{\mathbb{P}}$). For each operator *op* in $\{/, <, >, \leq, \geq\}$, the analysis computes the set of variables that *may* flow into the operands of an expression of the form e_1 *op* e_2. In practice, this is done via *backward may analysis*, noted as the `Dataflow` procedure in Algorithm 1. The obtained set of variables must be tracked using the integer semantics. The remaining variables will never flow into a problematic operator and can therefore be tracked using the modular semantics.

Implementation Remark. Since our implementation also supports arrays and recursion, the dataflow analysis in Algorithm 1 is inter-procedural and the set S also contains the array indexing operator [ ], i.e., given an expression $arr[a]$, if a variable v may flow into a, then a must be tracked using the integer semantics. Furthermore, while in our formalization we allow variables to be tracked using only one of the two semantics, in our implementation, we allow variables to be tracked differently (using actual values or modular values) at different program points by tracking, for each variable v, the program points for which the actual value of v is needed, which is done by using the same dataflow analysis. In this case, a variable might initially need to be tracked using actual values but can later be tracked using modular values.

Example 5.2. Consider the sketch program `polyArray` in Figure 1(b). For this program, Algorithm 1 will return that the variables `x` and `y` can be tracked modularly. However, the variables `i` and `n` must be tracked using the integer semantics, since they are used in a `<` operation and as array indices.

5.2 From IMP to IMP-MOD

Now that we have computed what sets of variables can be tracked modularly, we can transform the IMP program into an IMP-MOD program. The transformation R_f that rewrites f into an IMP-MOD program is shown in Figure 7. The key idea of the program transformation is to use the sets $V^{\mathbb{Z}}$ and $V^{\mathbb{P}}$ to only rewrite variables and sub-expressions of f for which the modular arithmetic can be performed soundly.

Once we get a solution for the IMP-MOD program as hole values, we can get a solution for the IMP program by mapping the hole to integer values given by the integer semantics.

$$
\mathrm{R}_a(a) = \begin{cases} v^{\mathbb{P}} & \text{if } a \equiv v \text{ and } v \in V^{\mathbb{P}} \\ c^{\mathbb{P}} & \text{if } a \equiv c \\ \mathrm{R}_a(a_1)\ op_a^{\mathbb{P}}\ \mathrm{R}_a(a_2) & \text{if } a \equiv a_1\ op_a^{\mathbb{P}}\ a_2 \\ \textsc{toPrime}(a) & \text{otherwise} \end{cases}
$$

$$
\mathrm{R}_b(b) = \begin{cases} \mathrm{R}_a(a_1) == \mathrm{R}_a(a_2) & \text{if } b \equiv a_1 == a_2 \\ \mathrm{R}_b(b_1) \text{ and } \mathrm{R}_b(b_2) & \text{if } b \equiv b_1 \text{ and } b_2 \\ \text{not } \mathrm{R}_b(b_1) & \text{if } b \equiv \text{not } b_2 \\ b & \text{otherwise} \end{cases}
$$

$$
\mathrm{R}_s(s) = \begin{cases} \mathrm{R}_s(s_1); \mathrm{R}_s(s_2) & \text{if } s \equiv s_1; s_2 \\ v\ =\ a & \text{if } s \equiv v = a \text{ and } v \in V^{\mathbb{Z}} \\ v^{\mathbb{P}}\ =\ \mathrm{R}_a(a) & \text{if } s \equiv v = a \text{ and } v \in V^{\mathbb{P}} \\ \texttt{if}(\mathrm{R}_b(b))\ \mathrm{R}_s(s_0)\ \texttt{else}\ \mathrm{R}_s(s_1) & \text{if } s \equiv \texttt{if}(b)\ s_0\ \texttt{else}\ s_1 \\ \texttt{while}(\mathrm{R}_b(b))\ \{\mathrm{R}_s(s)\} & \text{if } s \equiv \texttt{while}\ b\ \{s\} \\ \texttt{assert}\ \mathrm{R}_b(b) & \text{if } s \equiv \texttt{assert}\ b \end{cases}
$$

Fig. 7. Rules for the translation from IMP to IMP-MOD programs. Rules are parametric in $V^{\mathbb{Z}}$, $V^{\mathbb{P}}$ with $\mathbb{P}$: $\mathrm{R}_f(f(V, ??)\{s\}) = f(V^{\mathbb{Z}}, V^{\mathbb{P}}, ??)\{\mathrm{R}_s(s)\}$.

```
LoopInc(int n){
 int i = 0;
 int x = 0;
 while (i<n){
  x = x + i + 1;
  i++; }
 assert x==??*n*n;
}
```

(a)

```
LoopIncMod(int n){
 int i = 0;
 int x^P = 0^P;
 while (i<n){
  x^P = x^P + toPrime(i) + 1^P;
  i++; }
 assert x^P == toPrime(??)
         *toPrime(n) * toPrime(n);
}
```

(b)

Fig. 8. Transformation from the IMP program LoopInc in (a) to an IMP-MOD program LoopIncMod in (b).

Example 5.3. Consider the example program transformation shown in Figure 8. For this program, the dataflow analysis computes $V^{\mathbb{Z}} = \{i, n\}$ and $V^{\mathbb{P}} = \{x\}$. Therefore, the transformation rewrites the statement $x = x + i + 1$ to $x^{\mathbb{P}} = x^{\mathbb{P}} + \textsc{toPrime}(i) + 1^{\mathbb{P}}$.

The transformation R_f is sound.

Theorem 5.4. *Given an* IMP *program f, and sets $V^{\mathbb{Z}}$ and $V^{\mathbb{P}}$ resulting from the dataflow analysis on f, the program $R_f(f)$ is in the* IMP-MOD *language. Moreover, $[\![f]\!]^{\texttt{IMP}} = [\![R_f(f)]\!]$.*

Proof. First a few lemmas (note that in the formalization we only consider the language described in Figure 4, which for example does not include arrays).

Lemma 1. *For every a, $R_a(a)$ is a modular arithmetic expression in $a^{\mathbb{P}}$. We proceed by induction on a. The interesting case is the "otherwise" case. In this case, $R_a(a) = \textsc{toPrime}(a)$. Hence, we need to show that a is an* IMP-MOD *arithmetic expression, i.e., a does not contain any variable in $V^{\mathbb{P}}$. Let us analyze all the cases that can follow in the "otherwise" case:*

- *If $a \equiv v$, and $v \in V^{\mathbb{Z}}$, then a does not contain variables in $V^{\mathbb{P}}$.*
- *If $a \equiv ??$, then a does not contain variables in $V^{\mathbb{P}}$.*
- *If $a \equiv a_1/a_2$, then, from the dataflow analysis, all variables in a_1 and a_2 belong to $V^{\mathbb{Z}}$.*

Lemma 2. *For every b, $R_b(b)$ is a* IMP-MOD *Boolean expression in b. We proceed by induction on b. The first interesting case is $b \equiv a_1 == a_2$. Here, we have $R_b(b) = R_a(a_1) == R_a(a_2)$. From Lemma 1, $R_a(a_1)$ and $R_a(a_2)$ are in $a^{\mathbb{P}}$, and hence we are done.*
The other interesting case is the "otherwise" case. In this case, $R_b(b) = b$ and b is of the form $a_1\ op_c\ a_2$ where $op_c \in \{<, >, \leq, \geq\}$. Hence, we need to show that b is an IMP-MOD *Boolean expression, i.e., a does not contain any variable in $V^{\mathbb{P}}$. Due the operators in op_c, from the dataflow analysis, all variables in a_1 and a_2 belong to $V^{\mathbb{Z}}$.*

Now, we are ready to show that $\mathrm{R}_f(f) = f(V^{\mathbb{Z}}, V^{\mathbb{P}}, ??)\{\mathrm{R}_s(s)\}$ is in the IMP-MOD language. To do so, we show that $\mathrm{R}_s(s)$ is an IMP-MOD statement by induction on s. The interesting cases are the following:

- If $s \equiv v^{\mathbb{P}} = a$, then $\mathrm{R}_s(s) = v^{\mathbb{P}} = \mathrm{R}_a(a)$. From Lemma 1, $\mathrm{R}_a(a)$ is a modular arithmetic expression in $a^{\mathbb{P}}$. Hence we are done.
- If $s \equiv$ `assert` b, then $\mathrm{R}_s(s) =$ `assert` $\mathrm{R}_b(b)$. From Lemma 2, $\mathrm{R}_b(b)$ is a IMP-MOD Boolean expression. Hence we are done.

The fact that $[\![f]\!]^{\mathtt{IMP}} = [\![\mathrm{R}_f(f)]\!]$ follows from a straightforward induction on s, b, and a. □

6 SOLVING IMP-MOD SKETCHES

In this section, we discuss how synthesis in the modular semantics relates to synthesis in the integer semantics and provide an incremental algorithm for solving IMP-MOD sketches.

6.1 Synthesis in IMP-MOD

Given a set of integers R, we say that a variable valuation σ is in R (denoted $\sigma \in R$) if for every v we have $\sigma(v) \in R$. Similarly to what we saw in Section 3, we assume that the sketch has to be solved for finite ranges of possible values for the hole (R_H) and input values (R_{in}). Solving an IMP-MOD problem $P = f(V, V^{\mathbb{P}}, H)\{s\}$ for the integer semantics amounts to solving the following constraint:

$$\exists \sigma^H \in R_H. \forall \sigma_1, \sigma_2 \in R_{in}. [\![s]\!]_{\sigma_1 \cup \sigma^H, \sigma_2} \neq \bot.$$

According to Theorem 4.10, given a set of distinct primes $\mathbb{P} = \{p_1, \ldots, p_k\}$ and variable valuations σ^H, σ_1, and σ_2, if there exists a range R of size $N = p_1 \cdot \ldots \cdot \cdots p_k$ such that $s \in_{\sigma_1 \cup \sigma^H, \sigma_2} R$, then the modular semantics and the integer semantics are equivalent to each other. Using this observation, we can define the set of variable valuations for which the two semantics are guaranteed to be equivalent:

$$\mathcal{I}_R^{\mathbb{P}} := \left\{ (\sigma_1, \sigma_2) \mid \forall \sigma^H \in R_H. \exists R.\ |R| = N \wedge s \in_{\sigma_1 \cup \sigma^H, \sigma_2} R \right\}.$$

Since for every $\sigma^H \in R_H$ and $\sigma_1, \sigma_2 \in \mathcal{I}_R^{\mathbb{P}}$ we have that $[\![s]\!]^{\mathbb{P}}_{\sigma_1 \cup \sigma^H, m_{\mathbb{P}} \circ \sigma_2} = [\![s]\!]_{\sigma_1 \cup \sigma^H, \sigma_2}$, any solution to an IMP-MOD program in the modular semantics is also a solution to the following formula in the integer semantics:

$$\exists \sigma^H \in R_H. \forall \sigma_1, \sigma_2 \in \mathcal{I}_R^{\mathbb{P}}. [\![s]\!]_{\sigma_1 \cup \sigma^H, \sigma_2} \neq \bot.$$

When all valuations in $\sigma_1, \sigma_2 \in R_{in}$ are also elements of $\mathcal{I}_R^{\mathbb{P}}$, any solution to an IMP-MOD program in the modular semantics is guaranteed to be a correct solution under the integer semantics.

To summarize, if the synthesizer returns UNSAT for the IMP-MOD program, then the problem is unrealizable and does not admit a solution. When it returns a solution, the solution is correct if it only produces valuations in the range allowed by the choice of prime numbers. In practice, one can use a verifier to check the correctness of the synthesized solution and add more prime numbers to

ALGORITHM 2: Incremental synthesis for IMP-MOD.

```
/* f: function, ℙ: set of primes                                    */
function IncrementalSynthesis(f, ℙ)
    ℙ′ ← [p_1]
    f_syn ← Synthesis(f, ℙ′)
    while ∃p_cex ∈ ℙ : ¬Verify(f_syn, p_cex) do
        ℙ′ ← ℙ′ ∪ p_cex
        f_syn ← Synthesis(f, ℙ′)
        if f_syn == UNSAT then return ∅ ;
    return f_syn
```

the modular synthesizer if needed. In fact, this is the main idea behind the counterexample-guided inductive synthesis algorithm used by Sketch (Section 3).

6.2 Incremental Synthesis Algorithm

In this section, we propose an incremental synthesis algorithm that builds on the following observation. The set of variable valuations for which modular and integer semantics are equivalent increases monotonically in the size of $\mathbb{P}$:

$$\mathbb{P}_1 \subseteq \mathbb{P}_2 \implies \mathcal{I}_R^{\mathbb{P}_1} \subseteq \mathcal{I}_R^{\mathbb{P}_2}. \tag{1}$$

Algorithm 2 uses Equation (1) to add prime numbers lazily during the synthesis process. The algorithm first constructs a set $\mathbb{P}' = \{p_1\}$ with the first prime number $p_1 \in \mathbb{P}$ and synthesizes a solution that is correct for computations modulo the set $\mathbb{P}'$. It then checks if the synthesized solution f_{syn} satisfies the assertions with respect to all prime numbers in $\mathbb{P}$. If yes, then f_{syn} is returned as the solution. Otherwise, the algorithm finds a prime $p_{\text{cex}} \in \mathbb{P}$ where $\texttt{Verify}(f_{\text{syn}}, p_{\text{cex}})$ does not hold and it adds it to the set $\mathbb{P}'$ continuing the iterative algorithm. Due to Equation (1), Algorithm 2 is sound and complete with respect to the synthesis algorithm that considers the full prime set $\mathbb{P}$ all at once.

In practice, the user could use domain knowledge to estimate a suitable set of primes or alternatively use our incremental algorithm to discover appropriate prime sets. The set of prime numbers $\{2, 3, 5, 7, 11, 13, 17\}$ could usually instantiate a range R that is large enough for most synthesis tasks based on Sketch.

7 COMPLEXITY OF REWRITTEN PROGRAMS

In this section, we analyze how many bits are necessary to encode numbers for both semantics using unary and binary[4] bit-vector encodings of integers (Sections 7.1 and 7.2) and show how many prime numbers are necessary in the modular semantics to cover values up to a certain bound (Section 7.3). The following results build upon several number theory results that the reader can consult in References [9, 15].

7.1 Bit-complexity of Binary Encoding

In this section, we analyze how many bits are necessary when representing an interval of size N in binary in our modular semantics. In the rest of the section, we consider the set of primes $\mathbb{P}_n = \{p \mid p < n\} = \{p_1, \ldots, p_k\}$ containing the prime numbers that have value smaller than n. We

[4]Sketch does not currently implement a binary encoding, but we include this result for completeness and to show that applying our technique to a binary encoding is not as beneficial.

will show in Section 8 that this choice of prime number also yields good performance in practice. Concretely, we are interested in knowing what is the magnitude of the number $N = p_1 \cdot \ldots \cdot p_k$ and how many bits are used to represent the numbers in $\mathbb{P}_n$.

We start by introducing the notion of primorial.

Definition 7.1 (Primorial). Given a number n, the *primorial* $n\#$ is defined as the product of all primes smaller than n, i.e., $n\# = \prod_{p \in \mathbb{P}_n} p$.

The primorial captures the size N of the interval covered by the Chinese Remainder Theorem when using prime numbers up to n. The following number theory result gives us a close form for the primorial and shows that the number N has approximately n bits,

$$n\# = e^{(1+o(1))n} = 2^{(1+o(1))n}. \tag{2}$$

We use another number theory notion to quantify the number of bits in $\mathbb{P}_n$.

Definition 7.2 (Chebyshev Function). Given a number n, the *Chebyshev function* $\vartheta(n)$ is the sum of the logarithms of all the prime numbers smaller than n,i.e., $\vartheta(n) = \sum_{p \in \mathbb{P}_n} \log p$.

The following number theory result relates the primorial to the Chebyshev function:

$$\vartheta(n) = \log(n\#) = \log 2^{(1+o(1))n} = (1 + o(1))n. \tag{3}$$

Aside from rounding errors, the Chebyshev function captures the number of bits required to represent the numbers in $\mathbb{P}_n$. To obtain a more precise bound on this number, we need a bound for the formula $\sum_{p \in \mathbb{P}_n} \lceil \log p \rceil$.

We start by recalling the following fundamental number theory result.

Theorem 7.3 (Prime Number Theorem). *The set $\mathbb{P}_n$ has size approximately $n/\log n$.*

Using Theorem 7.3, we get the following result:

$$\sum_{p \in \mathbb{P}_n} \lceil \log p \rceil \leq n/\log n + \sum_{p \in \mathbb{P}_n} \log p \approx (1 + o(1))n. \tag{4}$$

Representing a number e^n in a classic binary encoding requires $\log_2(e^n) = (1 + o(1))n$ bits, and, combining Equations (2) and (4), we get the following result.

Theorem 7.4. *Representing a number 2^n in binary requires $(1 + o(1))n$ bits under both modular and integer semantics.*

Hence, representing a number in binary requires the same number of bits in the both semantics.

Example 7.5. Consider the set $\mathbb{P}_{18} = \{2, 3, 5, 7, 11, 13, 17\}$, which can model an interval of $N = 510,510$ integers (i.e., $n = 18$ in Theorem 7.4). Representing N in binary requires 19 bits while the binary representations of all the primes in $\mathbb{P}_{18}$ use 22 bits. Both numbers are close to 18 as predicted by the theorem.

7.2 Bit-complexity of Unary Encoding

As discussed in Section 3, the default Sketch solver encodes numbers using a unary encoding, i.e., Sketch requires 2^n bits to encode the number 2^n. Representing the same number in unary under the modular semantics requires only prime numbers smaller than n and therefore $\sum_{p \in \mathbb{P}_n} p$ bits. We can then use the following closed form to approximate this quantity:

$$\sum_{p \in \mathbb{P}_n} p \sim \frac{n^2}{2 \log n}. \tag{5}$$

Equation (5) yields the following theorem.

Theorem 7.6. *Representing a number 2^n in unary requires 2^n bits in the integer semantics and approximately $\frac{n^2}{2\log n}$ bits in the modular semantics.*

These results show that, under a unary encoding, the modular semantics is exponentially more succinct than the integer semantics.

Example 7.7. Consider again the prime set $\mathbb{P}_{18} = \{2, 3, 5, 7, 11, 13, 17\}$, which can model an interval of $N = 510,510$ integers. Representing N in unary requires 510,510 bits. However, the sum of the bits in the unary encoding of the primes in $\mathbb{P}_{18}$ is 58.

7.3 Number of Required Primes

We analyze how many primes are needed to represent a certain number in the modular semantics. We start by introducing the following alternative version of the primorial.

Definition 7.8 (Prime Primorial). For the nth prime number p_n, the *prime primorial* $p_n\#$ is defined as the product of the first n primes, i.e., $p_n\# = \prod_{k=1}^{n} p_i$.

The following known number theory result gives us an approximation for the prime primorial:

$$p_n\# = e^{(1+o(1))n\log n}. \tag{6}$$

Notice how the approximation of the primorial differs from that of the prime primorial. This is due to the fact that prime numbers are sparse, i.e., the nth prime number is approximately $n\log n$.

Using Equation (6) we obtain the following result.

Theorem 7.9. *Representing numbers in an interval of size $N = e^{n\log n}$ in the modular semantics requires the first n prime numbers.*

Since the relation $k = n\log n$ does not admit a closed form for n, we cannot derive exactly how many primes are needed to represent a number 2^k with k bits. It is, however, clear from the theorem that the number of required primes grows slower than k.

Example 7.10. Consider again the prime set $\mathbb{P}_{18} = \{2, 3, 5, 7, 11, 13, 17\}$, which can model an interval of $N = 510,510$ integers and consists of the first seven primes. According to Theorem 7.9, $\mathbb{P}_{18}$ should be able to model an integer interval of size approximately $e^{(1+o(1))(7\log 7)}$. In this case, N is approximately $e^{2.22(7\log 7)}$.

8 EVALUATION

We implemented a prototype of our technique as a simple compiler in Java. Our implementation provides a simplified Sketch frontend, which only allows the limited syntax we support. Given a Sketch file, our tool rewrites it into a different Sketch file that operates according to the modular semantics. We will use Unary to denote the result obtained by running the default version of Sketch with unary integer encoding on the original Sketch file; Native-Ints to denote the result obtained by running the version of Sketch using a native integer solver that extends the SAT solver with native integer variables, which it propagates them through the integer constraints, and learns from conflict clauses (flag `--slv-nativeints`)[5]; Unary-p to denote the result of running the default Sketch version on our modified Sketch file; and Unary-p-inc to denote the result of running the default version of Sketch on the file generated by the incremental version of our algorithm described in Section 6. All experiments were performed on a machine with 4.0-GHz Intel

[5]In the short version of this article published at ESOP20 [17], this version of Sketch was denoted with the name Binary. Since the encoding used by Native-Ints does not operate directly on a binary representation of numbers, we decided to use the name of the flag instead.

Core i7 CPU with 16 GB RAM with SKETCH-1.7.5, and we use a timeout value of 300 seconds (we also report out-of-memory errors as timeouts).

Our evaluation answers the following research questions:

Q1 How does the performance of UNARY-P compare to UNARY and NATIVE-INTS?
Q2 How does the incremental algorithm compare to the non-incremental one?
Q3 Is UNARY-P's performance sensitive to the set of selected prime numbers?
Q4 How many primes are needed by UNARY-P to produce correct solutions?
Q5 Does UNARY generate larger SAT queries than UNARY-P?

8.1 Benchmarks

We perform our evaluation on three families of programs.

Polynomials. The first set of benchmarks contains 81 variants of the polynomial synthesis problem presented in Figure 1. The original version of this benchmark appears in the SKETCH benchmark suite under the name `polynomial.sk`. For each benchmark, we generate a random polynomial f, random inputs $\{\overrightarrow{x}\}$, and take the set $\{(\overrightarrow{x}, f(x))\}$ as specification. Each benchmark in this set has the following parameters: `#Ex`$\in \{2, 4, 6\}$ is the number of input-output examples as specification, `cbits`$\in \{5, 6, 7\}$ denote the number of bits hole values can use, `exIn`$\in \{[-10, 10], [-30, 30], [-50, 50]\}$ denotes the range of randomly generated input examples, and `coeff`$\in \{[-10, 10], [-30, 30], [-50, 50]\}$ denotes the range of randomly generated coefficients in the polynomial f.

Invariants. The second set of benchmarks contain 46 variants of two invariant generation problems obtained from a public set of programs that require polynomial invariants to be verified [8]. We selected the two programs in which at least one variable could be tracked modularly by our tool (the other programs involved complex array operations or inequality operators) and turned the verification problems into synthesis problems by asking SKETCH to find a polynomial equality (using the program variables) that is an invariant for the loop in the program. To control the size of the magnitudes of the inputs, we only require the invariants to hold for a fixed set of input examples.

The first problem, `mannadiv`, iteratively computes the remainder and the quotient of two numbers given as input. The invariant required to verify `mannadiv` is a polynomial equality of degree 2 involving five variables. The SKETCH template required to describe the space of all polynomial equalities has 32 holes and cannot be handled by any of the SKETCH solvers we consider. We therefore simplify the invariant synthesis problems in two ways. In the first variant, we reduce the ranges of the hole values in the templates by considering `cbits` $\in \{2, 3\}$. In the second variant, we set `cbits` $= \{5, 6, 7\}$ but reduce the number of missing hole values to 4 (i.e., we provide part of the invariant). Each benchmark takes two random inputs, and we consider the following input ranges $\{[1, 50], [1, 100]\}$. In total, we have 10 benchmarks for `mannadiv`.

The second problem, `petter`, iteratively computes the sum $\sum_{1 \le i \le n} i^5$ for a given input n. The invariant required to verify `petter` is a polynomial equality of degree 6 involving three variables. The SKETCH template required to describe all such polynomial equalities has 56 holes and cannot be handled by any of the SKETCH solvers we consider. We consider the following simplified variants of the problem: (i) `petter_0` computes $\sum_{1 \le i \le n} 1$ and requires a polynomial invariant of degree one, (ii) `petter_x` computes $\sum_{1 \le i \le n} x$ for a given input variable x and requires a polynomial invariant of degree two, (iii) `petter_1` computes $\sum_{1 \le i \le n} i$ and requires a polynomial invariant of degree two, and (iv) `petter_10` computes $\sum_{1 \le i \le n} i + 1$ and requires a polynomial invariant of degree two. Each benchmark takes two random inputs and we consider the following input

Table 1. Effectiveness of Different Solvers

Solver	Solved	Polynomials			Invariants			Program repair		
		SAT	UNSAT	TO	SAT	UNSAT	TO	SAT	UNSAT	TO
Unary	**69**/181	12	4	65	5	0	41	48	0	6
Native-Ints	**127**/181	70	6	5	17	0	29	34	0	20
Unary-p	**169**/181	73	5	3	41	2	3	48	0	6
Unary-p-inc	**172**/181	73	6	2	41	2	3	50	0	4

SAT (respectively, UNSAT) denotes the number of benchmarks for which solver could find a solution to the benchmarks (respectively, prove no solution existed) while TO denotes the number of timeouts.

ranges $\{[1, 10], [1, 100], [1, 1000]\}$. In total, we have 12 variants of `petter`, each run for values of `cbits` $\in \{5, 6, 7\}$, i.e., a total of 36 benchmarks.

Program Repair. The third set of benchmarks contains 54 variants of Sketch problems from the domain of automatic feedback generation for introductory programming assignments [7]. Each benchmark corresponds to an incorrect program submitted by a student and the goal of the synthesizer is to find a small variation of the program that behaves correctly on a set of test cases. We select the 6/11 benchmarks from the tool Qlose [7] for which (i) our implementation can support all the features in the program and (ii) our dataflow analysis identifies at least one variable that can be tracked modularly. Of the remaining benchmarks, 3/11 do not contain variables that can be tracked modularly and 2/11 call auxiliary functions that cannot be translated into Sketch. For each program, we consider the original problem and two variants where the integer inputs are multiplied by 10 and 100, respectively. Further, for each program variants, we impose an assertion specifying that the distance between the original program and the repaired program is within a certain bound. We select three different bounds for each program: the minimum cost c, $c + 100$, and $c + 200$.

8.2 Performance of Unary-p

Table 1 summarizes our comparison.

First, we compare the performance of Unary-p and Unary. We use $\mathbb{P} = \{2, 3, 5, 7, 11, 13, 17\}$, which is enough for Unary-p to always find correct solutions (we verify the correctness of a solution by instantiating the hole values in the original sketch programs and using Sketch (without the `--slv-nativeints` flag) to verify that the assertions hold). We note that the Chinese Remainder Theorem does not require one to use only prime numbers, and it only requires pairwise co-prime integers. In our evaluation, we only consider prime numbers for simplicity and to avoid arbitrary choices of non-prime numbers. Unary can only solve 69/181 benchmarks while Unary-p can solve 169/181. Figure 10(a) shows a scatter plot (log scale) of the solving times for the two techniques: Each point below the diagonal line denotes a benchmark on which Unary-p was faster than Unary. Points on the extreme right-hand side of the plot denote timeout for Unary. When both solvers terminate, Unary-p (average 1.7 seconds) is 6.1× (geometric mean) faster than Unary (average 25.0 seconds).

Next, we compare the performance of Unary-p and Native-Ints (Figure 10(b)). On the 64 easier benchmarks that Native-Ints can solve in less than 1 second, Native-Ints (average 0.55 seconds) outperforms Unary-p (average 2.32 seconds), but Unary-p still has reasonable performance. On the 49 benchmarks that Native-Ints can solve between 1 and 10 seconds, Unary-p (average 3.5 seconds) is on average 1.9× faster than Native-Ints (average 6.9 seconds). Most interestingly, for the 14 harder benchmarks for which Native-Ints takes more than 10 seconds, Unary-p (average 5.7 seconds) is on average 15.9× faster than Native-Ints (average 90.9 seconds). Remarkably,

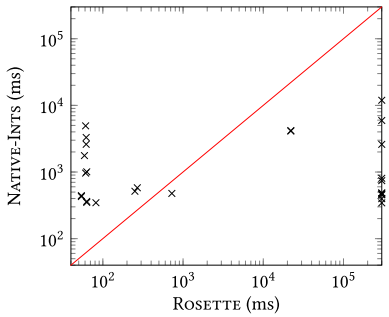


Fig. 9. Rosette vs. Native-Ints.

Unary-p solved 43 of the benchmarks (in less than 8 seconds each) for which Native-Ints timed out,[6] and Unary-p only timed out for two benchmarks that Native-Ints could solve in less than a second and one benchmark that Native-Ints could solve in 260 seconds. Finally, we highlight that for 41/208 benchmarks, even Unary outperforms Native-Ints. As expected from the discussion throughout the article, these are benchmarks typically involving complex operations but not involving overly large numbers.

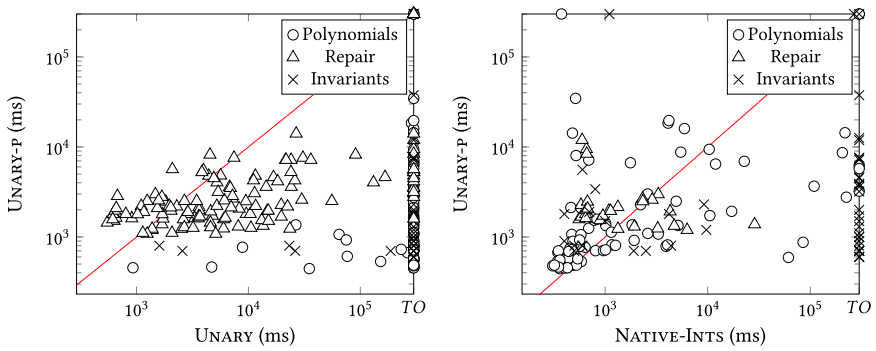
We can now answer **Q1**. First, *Unary-p consistently outperforms Unary* across all benchmarks. Second, *Unary-p outperforms Native-Ints on hard-to-solve problems and can solve problems that Native-Ints cannot solve*, e.g., Unary-p solved 28/46 invariant problems that Sketch could not solve with either encoding. Unary-p and Native-Ints have similar performance on easy problems.

Comparison to full SMT encoding. For completeness, we also compare our approach to a tool that uses SMT solvers to model the entire synthesis problem. We choose the state-of-the-art SMT-based synthesizer Rosette [26] (version 3.0 with Z3 v4.8.7) for our comparison. Rosette is a programming language that encodes verification and synthesis constraints written in a domain-specific language into SMT formulas that can be solved using SMT solvers.

We only run Rosette on the set of Polynomials, because Rosette does support the theories of integers but does not have native support for loops, so there is no direct way to encode Invariants and Program Repair benchmarks. To our knowledge, Rosette provides a way to specify the number k it uses to model integers and reals as k-bit words, but the user has no control over how many bits it uses for unknown holes specifically. So we evaluate 27 instead of 81 variants of the polynomial synthesis problem on Rosette, i.e., we consider different numbers of cbits. We run Rosette with the theory of integers (though Rosette supports many SMT theories including reals) in our experiments.

Figure 9 shows the running times (log scale) for Rosette and Native-Ints with `cbits=6`. Rosette successfully solved 16/27 benchmarks and it terminates quickly (average 2.9 seconds) when it can find a solution. However, Rosette times out on 11 benchmarks for which Native-Ints terminates. The timeouts are due to the fact that Rosette employs full SMT encodings that combine multiple theories while Native-Ints uses a SAT solver that is only modified to accommodate SMT-like integer constraints. Since we now know full SMT encodings are not as general and efficient as the encodings used in Sketch, we will only evaluate the effectiveness of our technique based on comparison with Native-Ints.

[6]During our experiment, we observed that Native-Ints *incorrectly* reported UNSAT for 10 satisfiable benchmarks. We reported these benchmarks as timeouts and have contacted the authors of Sketch to address the issue.



(a) UNARY vs UNARY-P (b) NATIVE-INTS vs UNARY-P

Fig. 10. Performance of UNARY, NATIVE-INTS, and UNARY-P.

We also consider the unsound encoding in which the holes and numbers are encoded as bit-vectors (16 bits) and directly encode the polynomial problems in SMT using quantifiers (to encode that the polynomial should be correct on all inputs) and the theory of bit-vectors. Z3 (v4.8.7) can efficiently solve all these instances (<1 seconds per instance). However, it often returns solutions that are correct in bit-vector arithmetic but not in integer arithmetic (due to overflow).

Finally, we tried applying our prime-based technique to ROSETTE (with integer variables), and the technique is not beneficial and causes all benchmarks to timeout.

To summarize, (i) SMT solvers cannot efficiently handle the synthesis problems considered in this article when considering the theory of integers but can efficiently solve (though at times unsoundly) some of the problems using the theory of bit-vectors, and (ii) our technique is better suited for the Sketch unary encoding of integers than for the encodings used for integers in SMT solvers.

The results leave open whether it is possible to build an SMT-based approach for solving programs sketches that can benefit from the techniques presented in this article. This direction is beyond the scope of this article, as our focus is on addressing the existing limitations of the SKETCH solver.

8.3 Performance of Incremental Solving

Our implementation of the incremental solver UNARY-P-INC first attempts to find a solution with the prime set $\mathbb{P} = \{2, 3, 5, 7\}$. If the solver returns a correct solution, then UNARY-P-INC terminates. Otherwise, UNARY-P-INC incrementally adds the next prime to $\mathbb{P}$ until it finds a correct solution, it proves there is no solution, or it times out. UNARY-P-INC is 25.2% (geometric mean) slower than UNARY-P (Figure 11 (log scale)). UNARY-P-INC can solve three benchmarks for which both UNARY-P and NATIVE-INTS timed out. To answer **Q3**, *UNARY-P-INC performs slightly better than UNARY-P.*

8.4 Varying the Prime Number Set $\mathbb{P}$

In this experiment, we evaluate how different prime number sets affect the performance and correctness of UNARY-P.

We consider the five increasing sets of primes: $\mathbb{P}_5 = \{2, 3, 5\}$, $\mathbb{P}_7 = \{2, 3, 5, 7\}$, $\mathbb{P}_{11} = \{2, 3, 5, 7, 11\}$, $\mathbb{P}_{13} = \{2, 3, 5, 7, 11, 13\}$, and $\mathbb{P}_{17} = \{2, 3, 5, 7, 11, 13, 17\}$. Figure 12(a) (log scale) shows the running times for all the polynomial benchmarks with `cbits` = 7 (showing all benchmarks would clutter the plot). The points where the lines change from dashed to solid denote the number of primes for which the algorithm starts yielding correct solutions. As expected, a smaller set of primes leads to

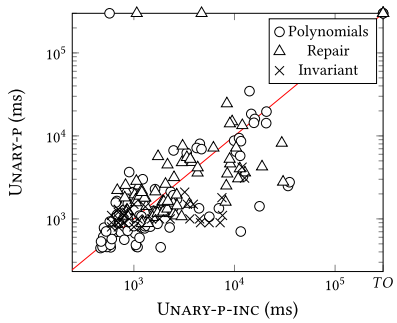
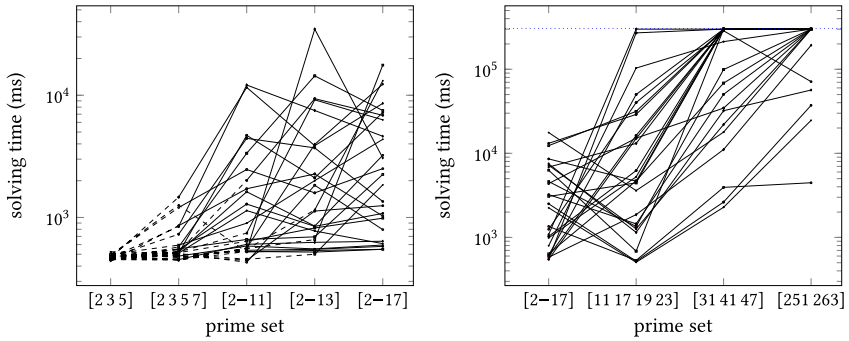


Fig. 11. Unary-p-inc vs. Unary-p.



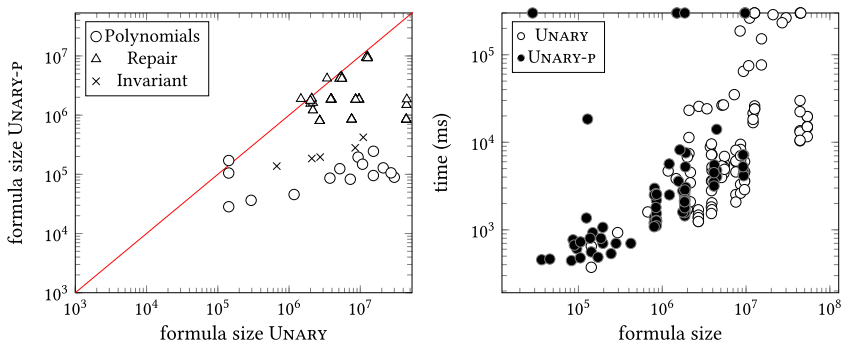
(a) Larger sets of primes (b) Larger primes

Fig. 12. Performance for different sets of prime numbers.

faster solving times as the resulting constraints are smaller and fewer bits are needed for encoding intermediate values. The runtime on average grows with the increasing size of the primes. For example, across all benchmarks, using $\mathbb{P}_{17}$ takes 23% longer on average than using $\mathbb{P}_{11}$. To answer **Q3**, *Unary-p is slower when using increasingly large sets of prime.*

In terms of correctness, we find that smaller prime sets often yield incorrect solutions ($\mathbb{P}_5$ (37% correct), $\mathbb{P}_7$ (70%), $\mathbb{P}_{11}$ (86%), $\mathbb{P}_{13}$ (97 %), and $\mathbb{P}_{17}$ (100%)), because there is not enough discriminative power with fewer primes and the solutions may overfit to the smaller set of intermediate values. It is interesting to note that even prime sets of intermediate size often lead to correct solutions in practice, which explains some of the speedups observed in the incremental synthesis algorithm. To answer **Q4**, *Unary-p is able to synthesize correct solutions even with intermediate sized sets of primes.*

Changing Magnitude of Primes. We also evaluate the performance of Unary-p when using primes of different magnitudes. We consider the sets of primes $\{11, 17, 19, 23\}$, $\{31, 41, 47\}$, and $\{251, 263\}$, which define similar integer ranges, but pose different tradeoffs between the number of used primes and their sizes, e.g., the set $\{251, 263\}$ only uses two very large primes. Since the different sets cover similar integer ranges, they all produce correct solutions. Figure 12(b) (log scale) shows the running time of Unary-p for the same benchmarks as Figure 12(a). Larger prime sets of smaller prime values require less time to solve than smaller prime sets of larger prime values. This result is expected, since, in the unary encoding of numbers, representing larger numbers requires more bits.



(a) Size: Unary-p vs Unary (b) Performance vs size

Fig. 13. SAT formulas sizes and performance.

8.5 Size of SAT Formulas

In this experiment, we compare the sizes of the intermediate SAT formulas generated by Unary-p and Unary. Figure 13(a) shows a scatter plot (log scale) of the number of clauses of the largest intermediate SAT query generated by the CEGIS algorithm for the two techniques. We only plot the instances in which Unary was able to produce at least a SAT formula. Unary produces SAT formulas that are on average 19.3× larger than those produced by Unary-p. To answer **Q5**, as predicted by our theory, *Unary-p produces significantly smaller SAT queries than Unary.*

Performance vs. Size of SAT Queries. We also evaluate the correlation between synthesis time and size of SAT queries. Figure 13(b) plots the synthesis times of both solvers against the sizes of the SAT queries. It is clear that the synthesis time increases with larger SAT queries. The plot illustrates how the solving time strongly depends on the size of the generated formulas.

9 RELATED WORK

Program Sketching. Program sketching was designed to automatically synthesize efficient bit-vector manipulations from inefficient iterative implementations [24]. A program sketch is a program template where a finite number of constants (holes) are missing and the goal of the synthesizer is to find values for such holes that can make a set of assertions hold. While sketching is limited to synthesizing constants, it is easy to encode problems that involve synthesizing a finite set of expressions by using the holes as guards of if-then-else operators. The Sketch tool can in fact support complex language features and operations [22]. Thanks to its simplicity, sketching has found wide adoption in applications such as optimizing database queries [3], automated feedback generation [21], program repair [7], and many others. Our work further extends the capabilities of Sketch in a new direction by leveraging number theory results. In particular, our technique allows Sketch to handle sketches manipulating large integer numbers. To the best of our knowledge, our technique is the first one that can solve many of the benchmarks presented in this article.

Uses of Chinese Remainder Theorem. The Chinese Remainder Theorem and its derivative corollaries have found wide application in several branches of Computer Science and, in particular, in Cryptography [11, 29].

The idea of using modular arithmetic to abstract integer values has been used in program analysis. Since modular fields are finite, they can be used as an abstract domain for verifying programs manipulating integers [5, 16, 19], e.g., the abstract domain can track whether a number is even or odd. Our work extends this idea to the domain of program synthesis and requires us to solve

several challenges. First, when used for verifying programs, the modular abstraction is used to overapproximate the set of possible values of the program and does not need to be precise. In particular, Clark et al. [5] allow program operations that are in the IMP language but not in the IMP-MOD language and lose precision when modeling such operations, e.g., when performing the assignment $x = x/2$ the value of x mod 2 can be either 0 or 1. Such imprecision is fine in program analysis, since the abstraction is used to show that a program does not contain a bug, i.e., even in the abstract domain, the problem behaves fine. In our setting, the problem is opposite, as we use the abstraction to simplify the synthesis problem and provide a theory for when the modular and integer semantics are equivalent.

Pruning Spaces in Program Synthesis. One of the key challenges in program synthesis is to tackle the large search space of possible programs [14] and many techniques have been proposed to efficiently prune large search spaces. Enumerative techniques [27] and version-space algebra synthesis techniques [12, 13, 20] enumerate programs in a search space and avoid enumerating syntactically and semantically equivalent terms.

Some synthesizers such as Synquid [18] and Morpheus [10] use refinement types and first-order formulas over specifications of DSL constructs to refute inconsistent programs. Recently, Wang et al. [28] proposed a technique based on abstraction refinement for iteratively refining abstractions to construct synthesis problems of increasing complexity for incremental search over a large space of programs.

We note that our work differs from these approaches because it targets program sketching, where the goal is identify constant values, instead arbitrary program synthesis. While sketching can be used to encode a number of synthesis problems, it limits the types of search spaces one can express and the ability to syntactically prune search spaces. We believe our technique could be used in tandem with existing program synthesis approaches to synthesize programs containing large constants, but this extension is beyond the scope of this article. In terms of technical difference, instead of pruning programs in the syntactic space as the aforementioned techniques do, our technique uses modular arithmetic to prune the semantic space (i.e., the complexity of verifying the correctness of the synthesized solution), while maintaining the syntactic space of programs. Our approach is related to that of Tiwari et al. [25], who present a technique for component-based synthesis using dual semantics where syntactic symbols in a language are provided two different semantics to capture different requirements. Our technique is similar in the sense that we also provide an additional semantics based on modular arithmetic. However, we formalize our analysis based on number theory results and develop it in the context of general-purpose SKETCH programs that manipulate integer values, unlike Tiwari et al.'s work that is developed for straight-line programs composed of components.

Synthesis for Large Integer Values. Abate et al. propose CEGIS(T), a modification of the CEGIS algorithm for solving SyGuS problems with large constants [1]. CEGIS(T) uses two nested loops. In an outer loop, it enumerates "template programs" that contain holes for the constants, while in an inner loop it relies on an SMT solver to find the valuations for these constants. The main novelty is that the inner loop can use techniques such as quantifier elimination to discard large sets of constants that cannot fill the holes and potentially prove that no constants exist.

Our approach is orthogonal to that of Abate et al. First, our work is focused on program sketching and not SyGuS. While the two ways of specifying synthesis problems have ways to be related, they are fundamentally different. Sketching limits the user to providing templates where only a finite number of constants are missing and by bounding many quantities (e.g., loop iterations, input sizes), but it allows one to use many complex programming constructs and retain practical synthesis. SyGuS allows the user to specify grammars of programs (which need to consist of terms

in a given theory) from which the final program can be synthesized. Furthermore, existing SyGuS solvers are typically designed for specific theories (e.g., LIA). Many of the examples described in this article cannot directly be encoded in SyGuS, as they use complex control constructs (e.g., to encode loops one would require to explicitly unroll programs and operations such as non-linear integer arithmetic cannot be handled by existing SyGuS solvers). Dually, one cannot encode the infinite search spaces encoded by SyGuS grammar as a program sketch, making the two formalisms complementary. Both our approach and the one by Abate et al. were designed to address the inability of existing solvers to synthesize large constants, and they have different uses. Second, for CEGIS(T) to work, the constants that need to be synthesized have to appear in specific places in the template, e.g., they cannot be the coefficients of linear terms. For our technique to work, the constants can appear as coefficients but cannot be manipulated using certain operators (e.g., division). In principle, the two approaches could be combined, and the CEGIS(T) algorithm presented by Abate et al. could potentially make use of our technique to fill the holes in the templates they synthesize. This possible extension is beyond the scope of this article.

10 CONCLUSION

We presented a new technique for solving program sketches with large integer values. Our technique rewrites the sketches to operate in a different semantics that only manipulates small values. In particular, instead of tracking concrete integer values, the rewritten sketches only track the remainders of such values for an appropriate set of prime numbers. Using the Chinese Remainder Theorem, we showed that our technique is sound and yields correct solutions as long as the values manipulated by the program never exceed certain well-defined boundaries. We provide a dataflow analysis for detecting when program variables and values can be modeled using this modified semantics and implement our technique in UNARY-P. The evaluation of our technique on 181 benchmarks that manipulate large integer values shows that our solver can solve 100 new benchmarks that the SKETCH unary solver could not solve and is 15.9× faster than the SKETCH SMT-like integer solver on the hard benchmarks that take more than 10 seconds to solve.

This article opens new opportunity for encoding constraint problems involving large integer values. The proposed techniques are general and could be applied to other domains beyond program sketching. We leave investigating the effectiveness of the techniques on other domains (e.g., SMT solvers) as future work.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for commenting on earlier drafts and Rong Pan for his contributions to this article.

REFERENCES

[1] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. 2018. Counterexample guided inductive synthesis modulo theories. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'18),* Lecture Notes in Computer Science. Springer.

[2] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD'13).* 1–8.

[3] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13).* 3–14.

[4] Lindsay N. Childs (Ed.). 2009. *The Chinese Remainder Theorem.* Springer, New York, NY, 253–281. https://doi.org/10.1007/978-0-387-74725-5_12

[5] Edmund M. Clarke, Orna Grumberg, and David E. Long. 1994. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.* 16, 5 (Sept. 1994), 1512–1542. https://doi.org/10.1145/186025.186051

[6] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'77)*. ACM, New York, NY, 238–252. https://doi.org/10.1145/512950.512973

[7] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'16)*, Lecture Notes in Computer Science, Vol. 9780. Springer, 383–401.

[8] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. 2016. Polynomial invariants by linear algebra. In *Automated Technology for Verification and Analysis*, Cyrille Artho, Axel Legay, and Doron Peled (Eds.). Springer International Publishing, Cham, 479–494.

[9] Pierre Dusart. 2016. Estimates of ψ,ϑ for large values of x without the Riemann hypothesis. *Math. Comput.* 85, 298 (2016), 875–888. https://doi.org/10.1090/mcom/3005

[10] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, 422–436. https://doi.org/10.1145/3062341.3062351

[11] J. Grobchadl. 2000. The chinese remainder theorem and its application in a high-speed RSA crypto chip. In *Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC'00)*. IEEE Computer Society, Washington, DC, USA, 384–.

[12] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. 317–330. https://doi.org/10.1145/1926385.1926423

[13] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105. https://doi.org/10.1145/2240236.2240260

[14] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program synthesis. *Found. Trends Program. Lang.* 4, 1–2 (2017), 1–119. https://doi.org/10.1561/2500000010

[15] G. J. O. Jameson. 2003. *The Prime Number Theorem*. Cambridge University Press. https://doi.org/10.1017/CBO9781139164986

[16] Shinji Kimura. 1995. Residue BDD and its application to the verification of arithmetic circuits. In *Proceedings of the 32nd Design Automation Conference*. 542–545. https://doi.org/10.1109/DAC.1995.250006

[17] Rong Pan, Qinheping Hu, Rishabh Singh, and Loris D'Antoni. 2020. Solving program sketches with large integer values. In *Proceedings of the 29th European Symposium on Programming (ESOP'20)*, Lecture Notes in Computer Science, Peter Müller (Ed.), Vol. 12075. Springer, 572–598. https://doi.org/10.1007/978-3-030-44914-8_21

[18] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI'16)*. 522–538. https://doi.org/10.1145/2908080.2908093

[19] Kavita Ravi, Abelardo Pardo, Gary D. Hachtel, and Fabio Somenzi. 1996. Modular verification of multipliers. In *Proceedings of the 1st International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*. Springer-Verlag, Berlin, 49–63.

[20] Rishabh Singh and Sumit Gulwani. 2016. Transforming spreadsheet data types using examples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*. 343–356. https://doi.org/10.1145/2837614.2837668

[21] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. 15–26. https://doi.org/10.1145/2462156.2462195

[22] Armando Solar-Lezama. 2013. Program sketching. *Int. J. Softw. Tools Technol. Transf.* 15, 5–6 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

[23] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. 2008. Sketching concurrent data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*. 136–148. https://doi.org/10.1145/1375581.1375599

[24] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. *SIGOPS Operat. Syst. Rev.* 40, 5 (Oct. 2006), 404–415. https://doi.org/10.1145/1168917.1168907

[25] Ashish Tiwari, Adrià Gascón, and Bruno Dutertre. 2015. Program synthesis using dual interpretation. In *Proceedings of the 25th International Conference on Automated Deduction (CADE'15)*. 482–497. https://doi.org/10.1007/978-3-319-21401-6_33

[26] Emina Torlak and Rastislav Bodik. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, New York, NY, 530–541. https://doi.org/10.1145/2594291.2594340

[27] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying protocols with concolic snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. 287–296.

[28] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.* 2 (2018), 63:1–63:30. https://doi.org/10.1145/3158151

[29] Sung-Ming Yen, Seungjoo Kim, Seongan Lim, and Sang-Jae Moon. 2003. RSA speedup with chinese remainder theorem immune against hardware fault cryptanalysis. *IEEE Trans. Comput.* 52, 4 (April 2003), 461–472. https://doi.org/10.1109/TC.2003.1190587

Received April 2020; revised September 2021; accepted November 2021