Safe and Sound Program Analysis with Flix

Magnus Madsen
Aalborg University
Denmark
magnus@cs.aau.dk

Ondřej Lhoták
University of Waterloo
Canada
olhotak@uwaterloo.ca

ABSTRACT

Program development tools such as bug finders, build automation tools, compilers, debuggers, integrated development environments, and refactoring tools increasingly rely on static analysis techniques to reason about program behavior. Implementing such static analysis tools is a complex and difficult task with concerns about *safety* and *soundness*. Safety guarantees that the fixed point computation – inherent in most static analyses – converges and ultimately terminates with a deterministic result. Soundness guarantees that the computed result over-approximates the concrete behavior of the program under analysis. But how do we know if we can trust the result of the static analysis itself? *Who will guard the guards?*

In this paper, we propose the use of automatic program verification techniques based on symbolic execution and SMT solvers to verify the correctness of the abstract domains used in static analysis tools. We implement a verification toolchain for Flix, a functional and logic programming language tailored for the implementation of static analyses. We apply this toolchain to several abstract domains. The experimental results show that we are able to prove 99.5% and 96.3% of the required safety and soundness properties, respectively.

CCS CONCEPTS

• **Theory of computation → Program analysis**;

KEYWORDS

static analysis, lattices, safety, soundness, monotonicity

ACM Reference Format:
Magnus Madsen and Ondřej Lhoták. 2018. Safe and Sound Program Analysis with Flix. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3213846.3213847

1 INTRODUCTION

Designing, implementing, and testing static analysis tools is a challenging task. The analysis designer is faced with difficult trade-offs between ensuring soundness, precision, and scalability of the analysis. The fixed point nature of many static analyses combined with these trade-offs often lead to implementations that are hard to understand, debug, and extend. To avoid these issues, some designers have turned to Datalog [Ceri et al. 1989]. Datalog is a declarative programming language for constraints on relations. A Datalog program is a set of rules that together with a set of input facts imply a minimal model. Datalog has been successfully used for specification of large-scale points-to analyses for object-oriented programs, in particular for Java [Bravenboer and Smaragdakis 2009; Smaragdakis and Bravenboer 2011]. The use of Datalog is not limited to points-to analyses; other analyses can be implemented in Datalog, such as definite assignment, reaching definitions, and available expressions. These analyses involve *constraints on relations*. However, many analyses are not defined as constraints on *relations*, but as constraints on *lattices*. This includes classic textbook analyses, such as sign analysis, constant propagation, and interval analysis. Regrettably, such analyses cannot be expressed in Datalog.

Flix, a functional and logic language, has been proposed to overcome these limitations [Madsen et al. 2016a,b, 2018]. Flix is inspired by Datalog and extends it with user-defined lattices as well as monotone filter and transfer functions. In Flix, it is straightforward to express dataflow analyses on lattices, and other types of analyses.

Datalog has an important property that is also desirable for Flix: Every Datalog program has a unique minimal model that can be efficiently computed by semi-naïve evaluation [Ceri et al. 1989]. Intuitively, the solution to a Datalog program is a finite set of facts, there is a finite number of such solutions, and exactly one of them is the minimal model. In Flix, the situation is more complicated. The semantics of a Flix program is dependent on user-defined functions, expressed in a functional language, which define lattices and operations on lattices. Unfortunately, if these user-provided functions, which are program code, are erroneous, then the meaning of a Flix program is undefined and the fixed point computation may diverge. We want to avoid that and ensure the same *safety* guarantees for Flix as those of Datalog.

But safety alone is not enough, we also want *soundness*. Soundness means that the model computed by a safe Flix program is a sound over-approximation of the behaviour of the program under analysis. In other words, soundness means that we can trust the results of the analysis when reasoning about the program.

In the current state-of-affairs, when reasoning about the safety and soundness of an analysis, the analysis designer has essentially three choices: i) hand-wave an argument or proof (requires the least amount of work), ii) work out a pen and paper proof (requires a significant amount of work), or iii) construct a mechanized proof (requires a substantial effort). In this paper, we propose a fourth approach: to use program verification techniques to automatically *prove* that the static analyzer is safe and sound. Our hope is to provide robust guarantees without imposing any significant burden on the analysis designer.

Automatically proving *soundness* of an entire static analysis is very challenging, so as a first step, we settle for an easier problem:

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213847

Proving that the abstract domains used by the analysis are sound. If this is the case, what remains to be proven is that the constraints of the logic language accurately model the concrete semantics of the program. Fortunately, there is often a one-to-one correspondence between the concrete and abstract semantics. For example, and as we shall see later, the constraint for an addition statement in the concrete and abstract semantics is often the same.

The concerns of safety and soundness are not unique to Flix, but apply equally well to other static analysis frameworks and tools.

In this paper, we extend the Flix language with annotations to describe the necessary mathematical properties of functions, and with a verifier based on symbolic execution and satisfiability modulo theories to automatically check whether these properties hold, or if not, to produce counter-examples. The reliance on fully-automatic techniques is in line with our philosophy that Datalog (and in extension Flix) should be easy to use and that benefits should come for free whenever possible.

In summary, our paper makes the following contributions:

- We demonstrate, with several examples, that ensuring the correctness, i.e. safety and soundness, of even a simple static analysis is a non-trivial task with many pitfalls.
- We extend the Flix programming language with annotations and laws to capture the mathematical properties required of abstract domains and their operations.
- We implement a verification toolchain, based on symbolic execution and satisfiability modulo theories, to verify that Flix programs are correct, i.e. that user-defined lattices and their operations satisfy the mathematical properties necessary to ensure safety and soundness.
- We experimentally evaluate the usefulness of our verification toolchain by applying it to several abstract domains. The results are encouraging and show that we can verify 99.5% and 96.3% of the required safety and soundness properties.

2 MOTIVATION

In this section, we introduce the Flix programming language by showing how it can be used to express a simple dataflow analysis. We then discuss several unsafe and unsound Flix programs that motivate the need for verification of abstract domains.

2.1 A Simple Dataflow Analysis

Figure 1 shows a simple intra-procedural, flow-sensitive dataflow analysis implemented in Flix. This analysis computes, for every local variable, and at every program point, whether the variable is an odd or even number.

Partial Order and Lattice Definition. Lines 1–6 define an algebraic data type named `Parity`. This data type has four values: `Top`, `Even`, `Odd`, and `Bot`. Lines 8–17 define a partial order on lattice elements of the `Parity` type. The partial order is defined as a boolean valued function of type `leq` : `Parity` × `Parity` → `Bool` which determines when one element is less than or equal to another element. Line 9–10 associates five *property annotations* with the function. We will discuss these in greater detail later. Lines 19–21 and 23–25 define the least upper bound operator and greatest lower bound operator of the lattice. Lines 27–28 define a function `beta` to lift 32-bit integers into elements of the parity lattice.

```
1   /* elements of the parity lattice */
2   enum Parity {
3           case Top,
4     case Even, case Odd,
5           case Bot
6   }
7
8   /* the partial order */
9   #reflexive #antiSymmetric #transitive
10  #leastElement(Bot) #greatestElement(Top)
11  def leq(e1: Parity, e2: Parity): Bool = match (e1, e2) with {
12    case (Bot, _)     => true
13    case (Even, Even) => true
14    case (Odd, Odd)   => true
15    case (_, Top)     => true
16    case _            => false
17  }
18
19  /* the least upper bound */
20  #upperBound #leastUpperBound
21  def lub(x: Parity, y: Parity): Parity = /* ... */
22
23  /* the greatest lower bound */
24  #lowerBound #greatestLowerBound
25  def glb(x: Parity, y: Parity): Parity = /* ... */
26
27  /* the abstraction function */
28  def beta(x: Int): Parity = if (x % 2 != 0) Odd else Even
29
30  /* filter and transfer functions */
31  #strict1 #monotone1 #sound1(x -> x == 0)
32  def isMaybeZero(x: Parity): Bool = /* ... */
33
34  #strict2 #monotone2 #sound2((x, y) -> x + y)
35  def sum(x: Parity, y: Parity): Parity =  /* ... */
36
37  #strict2 #monotone2 #sound2((x, y) -> x / y)
38  def div(x: Parity, y: Parity): Parity =  /* ... */
39
40  /* declaration of input relations */
41  rel CFG(s1: Stm, s2: Stm)
42  rel CstStm(s: Stm, x: Var, i: Int)
43  rel AddStm(s: Stm, r: Var, x: Var, y: Var)
44  rel DivStm(s: Stm, r: Var, x: Var, y: Var)
45  rel NotKill(s: Stm, x: Var)
46
47  /* declaration of lattices */
48  lat In(s: Stm, x: Var, v: Parity)
49  lat Out(s: Stm, x: Var, v: Parity)
50  rel DivByZero(s: Stm, x: Var)
51
52  /* incoming dataflow */
53  In(s2, x, v) :- CFG(s1, s2), Out(s1, x, v).
54
55  /* outgoing dataflow */
56  Out(s, x, beta(i)) :- CstStm(s, x, i).
57  Out(s, r, sum(v1, v2)) :-
58    AddStm(s, r, x, y), In(s, x, v1), In(s, y, v2).
59  Out(s, r, div(v1, v2)) :-
60    DivStm(s, r, x, y), In(s, x, v1), In(s, y, v2).
61  Out(s, x, v) :- In(s, v), NotKill(s, x).
62
63  /* check for division by zero */
64  DivByZero(s, y) :-
65    DivStm(s, _, _, y), In(s, y, v), isMaybeZero(v).
```

Figure 1: An intra-procedural dataflow analysis.

Filter and Transfer Functions. Lines 30–38 define *filter* and *transfer* functions for the parity lattice, e.g. abstract addition and division. Specifically, lines 30–32 define a function to determine whether a parity lattice element may represent zero, lines 34–35 and lines 37–38 define abstract addition and division, respectively.

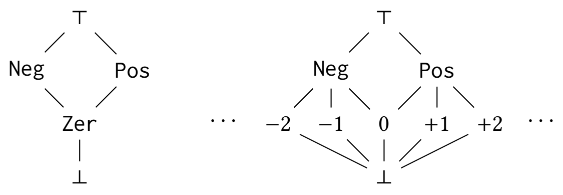
Relations & Lattices. Lines 40–45 declare four input relations where: a fact CFG(s_1, s_2) represents an edge from s_1 to s_2 in the control-flow graph, a fact CstStm(s, x, i) represents a constant assignment $x = i$ at statement s where the variable x is assigned the value i, a fact AddStm(s, r, x, y) represents an addition expression $r = x + y$ at statement s, a fact DivStm(s, r, x, y) represents a division expression $r = x/y$ at statement s, and finally a fact NotKill(s, x) represents that the variable x is not killed by the statement s. Lines 48–49 declare two lattices that represent incoming and outgoing dataflow of each control-flow graph node, i.e. the dataflow immediately before and immediately after a statement. Line 50 declares a relation to collect the statements (and local variables) where a division-by-zero error may occur. This is ultimately the result of the analysis.

Semantic Rules. Lines 52–65 define the semantic rules of the analysis. Line 53 defines a rule which propagates dataflow from every control-flow graph node to its successors. Lines 56–61 define the semantics of the three kinds of statements: constant assignment, addition and division. For example, the rule for addition says that if there is an addition statement $s : r = x + y$, then the value of r, immediately after statement s, is at least the result of applying the transfer function sum to the values of the variables x and y immediately before the statement s. Line 61 propagates dataflow values, from the program point immediately before a statement to the program point immediately after that statement, for every variable which is not killed by that statement. Finally, Line 65 uses the filter function isMaybeZero to select the statements and variables that may cause a division by zero error.

Safety and Soundness. An important point here is that the analysis is separated into two parts: one part defines the constraints of the analysis in a logic language, and the other part defines the lattices and their operations in a functional language. In the program, the functions leq, lub, glb, isMaybeZero, sum, and div are decorated with a set of *annotations*. For example, the lub function has the annotations #upperBound and #leastUpperBound specifying that the function must be both an upper bound and *the* least upper bound (according to the partial order defined by leq). As another example, the sum function has the annotations #strict2, #monotone2, and #sound2((x, y) -> x + y) specifying that the function must be strict, monotone, and an over-approximation of the function $\lambda x.\lambda y.x + y$. For practical reasons, each annotation includes its arity. For example, the #monotone1 and #monotone2 annotations capture the monotonicity property of unary and binary functions, respectively. In this example, we have explicitly annotated every function with its required properties, but in practice all annotations except for the soundness annotations can be inferred.

2.2 How Flix Programs May "Go Wrong": Unsafe and Unsound Programs

As discussed, a Flix program has a logic and a functional part. The programmer uses the functional language to express abstract domains, whereas the logic language is used to express constraints on elements of the abstract domains. Unfortunately, user-defined functions may contains bugs and fail to satisfy the required safety and soundness properties. In this subsection, we consider three such erroneous programs. We stress that these problems are *not* specific to Flix, but could occur in any program analysis implementation.



(a) Sign **(b) ConstSign**

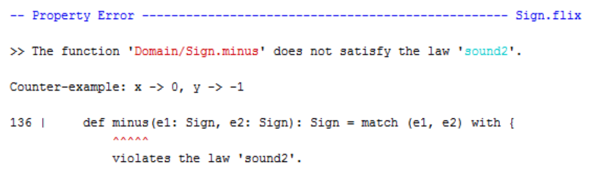
Figure 2: The Sign and ConstSign lattices.

```
def minus(e1: Sign, e2: Sign): Sign =
  match (e1, e2) with {
    case (Bot, _)   => Bot
    case (_, Bot)   => Bot
    case (Neg, Neg) => Top
    case (Neg, Zer) => Neg
    case (Neg, Pos) => Neg
    case (Zer, Neg) => Neg
    case (Zer, Zer) => Zer
    case (Zer, Pos) => Neg
    case (Pos, Neg) => Pos
    case (Pos, Zer) => Pos
    case (Pos, Pos) => Top
    case  _         => Top
  }
```

Figure 3: Erroneous transfer function.

Example I. Figure 2a shows the sign lattice. This lattice has five elements: Neg, which represents all negative integers including zero; Zer, which represents zero; Pos, which represents all positive integers including zero; ⊤, which represents any integer; and ⊥, which represents "not an integer". Figure 3 shows an implementation of the minus operation on sign lattice elements. The intention is that the function is an over-approximation of subtraction on integers. For example, if you have a positive number Pos and you subtract zero Zer, then you get a positive number Pos.

We may now ask ourselves: is this function strict, monotone, and a sound over-approximation of subtraction for integers? Strictness and monotonicity are required for the minimal model of the Flix program to exist (i.e. determinancy and termination), whereas soundness is required to trust the result of the analysis. When run on this program, Flix reports that the function is *unsound*:



```
-- Property Error ------------------------------------------------ Sign.flix

>> The function 'Domain/Sign.minus' does not satisfy the law 'sound2'.

Counter-example: x -> 0, y -> -1

136 |     def minus(e1: Sign, e2: Sign): Sign = match (e1, e2) with {
              ^^^^^
              violates the law 'sound2'.
```

Flix tells us that $x = 0$ and $y = -1$ is a counter-example to the soundess property. We can calculate that:

$$0 - (-1) = +1 \quad \text{whereas} \quad \texttt{minus(Zer,Neg)} = \texttt{Neg}$$

but the abstraction of +1 yields Pos and clearly Pos $\not\sqsubseteq$ Neg, hence the function is unsound! In this case, the mistake is in case (Zer, Neg) => Neg which should have been case (Zer, Neg) => Pos since subtracting a negative number from zero yields a positive number. After fixing this issue, Flix does not issue any other warnings and we can rest assured that the function is strict, monotone, and sound.

```
def increment(e: ConstSign): ConstSign =
  match e with {
    case Bot    => Bot
    case Cst(n) => Cst(n + 1)
    case Neg    => Top
    case Pos    => Pos
    case Top    => Top
  }
```

Figure 4: Erroneous transfer function.

Example II. Figure 2b shows the constant sign lattice, which has the elements $\bot$, $\top$, Neg, Pos, and `Cst(n)` for any $n \in \mathbb{Z}$. As before, Neg and Pos represent the negative and positive integers (including zero), and `Cst(n)` represents the integer n. Intuitively, this lattice is the reduced product of the constant propagation and sign lattices. Figure 4 shows the implementation of the "increment by one" operation on elements of this lattice. Again, we may ask ourselves: is this function strict, monotone, and sound?

Perhaps surprisingly, this seemingly simple function contains a subtle bug. When run on this program, FLIX reports that the function is *not* monotone and provides a counter-example $x = \texttt{Cst}(2147483647)$ and $y = \texttt{Pos}$. We can calculate:

$$\texttt{increment}(\texttt{Cst}(2147483647)) = \texttt{Cst}(-2147483648)$$
$$\texttt{increment}(\texttt{Pos}) = \texttt{Pos}$$

Let us denote `increment` by f. Monotonicity requires that when $x \sqsubseteq y$ it must be that $f(x) \sqsubseteq f(y)$, but due to integer overflow $f(\texttt{Cst}(2147483647)) = \texttt{Cst}(-2147483648)$, and $\texttt{Cst}(-2147483648) \not\sqsubseteq \texttt{Pos}$. Consequently, the function is not monotone and the fixed point computation may not terminate. The problem is that in the definition of `Cst`, not shown, we used a regular 32-bit integer. The analysis must take overflow into account to ensure monotonicity!

Example III. Assume we have fixed the previous issue. Figure 5 shows an implementation of division on the `ConstSign` lattice. Again, we ask ourselves: is this function, strict, monotone, and sound? FLIX reports that the function is not monotone when:

$$x = \texttt{Cst}(0) \quad \text{and} \quad y = \texttt{Pos}$$

Specifically, we have that $\texttt{divide}(\texttt{Cst}(0), \texttt{Neg}) = \texttt{Pos}$, and at the same time $\texttt{divide}(\texttt{Pos}, \texttt{Neg}) = \texttt{Neg}$. However, $\texttt{Pos} \not\sqsubseteq \texttt{Neg}$. Consequently, the function is not monotone and the fixed point computation may not terminate. The issue is due to the missing case:

```
  case (Cst(0), _) => Cst(0)
```

Adding this fixes the bug and FLIX reports no other warnings and we can rest assured that the function is strict, monotone, and sound.

Discussion. As these examples demonstrate, the implementation of abstract domains is a difficult task fraught with error. The analysis designer must work carefully to ensure both safety and soundness. The consequences of a small mistake can be grave: The fixed point computation may produce non-deterministic results, fail to terminate, or perhaps worst of all, produce unsound results. Adding insult to injury, debugging fixed point computations tends to be a difficult and time consuming process.

We propose to overcome these issues with automatic program verification techniques. We want an automatic technique so that we can help the programmer without burdening him or her with extra work. A verification tool can provide the programmer with confidence that his implementation is correct, or if not, provide

```
def divide(e1: ConstSign, e2: ConstSign): ConstSign =
  match (e1, e2) with {
    case (Bot, _)            => Bot
    case (_, Bot)            => Bot
    case (_, Cst(0))         => Bot
    case (Cst(n1), Cst(n2)) => Cst(n1 / n2)
    case (Cst(n1), Neg)      => if (n1 > 0) Neg else Pos
    case (Neg, Cst(n1))      => if (n1 > 0) Neg else Pos
    case (Cst(n1), Pos)      => if (n1 < 0) Neg else Pos
    case (Pos, Cst(n1))      => if (n1 < 0) Neg else Pos
    case (Neg, Neg)          => Pos
    case (Pos, Neg)          => Neg
    case (Neg, Pos)          => Neg
    case (Pos, Pos)          => Pos
    case _                   => Top
}
```

Figure 5: Example of buggy transfer function.

concrete counter-examples showing (a) what properties are violated and (b) give specific inputs that violate the properties.

3 BACKGROUND

In this section, we briefly recap the mathematical background behind sound static analyzers. We pay close attention to the role of lattices and their operations. Our goal is three-fold: (a) to specify the necessary conditions for a FLIX program to be safe, i.e. to capture when a FLIX program has a unique minimal model which is efficiently computable by semi-naïve evaluation, (b) to specify the necessary conditions for a FLIX program to soundly over-approximate the concrete behaviour of some program under analysis, and (c) to demonstrate the complexity that faces the analysis designer and motivate the use of automatic verification techniques.

3.1 Soundness of Abstract Semantics

At a high level, we seek to ensure that the abstract semantics embodied in the FLIX program is a sound over-approximation of a specified concrete semantics of the language being analyzed.

At a general level, the concrete semantics is defined by a set S of machine states, and a transition function $f : S \to S$. For the abstract semantics, we choose an abstract domain $\hat{S}$ and an abstract transition function $\hat{f} : \hat{S} \to \hat{S}$. To show a correspondence between the concrete and abstract semantics, we must first define the relationship between an abstract state and a concrete state, in the form of a function $\beta : S \to \hat{S}$ that axiomatizes the most precise abstract representation of each concrete state. We then require that every state transition in the concrete semantics is soundly over-approximated by the abstract semantics. If the concrete semantics transitions from a concrete state s to $f(s)$, then the abstract semantics must transition from $\beta(s)$ to $\beta(f(s))$. However, this requirement is too strict: it does not admit abstraction in the abstract semantics. We add abstraction to the abstract domain by defining a partial order $\sqsubseteq$ that means that if $\hat{s} \sqsubseteq \hat{s}'$, then we consider $\hat{s}'$ to be an abstraction of $\hat{s}$. Our soundness obligation is then:

$$\forall s \in S : \beta(f(s)) \sqsubseteq \hat{f}(\beta(s))$$

Using this inequation, we can prove that if s is the concrete state after a finite sequence of concrete execution steps, it is abstracted by the abstract state after the same number of abstract steps:

$$\forall n \geq 0 : \beta(f^{(n)}(i)) \sqsubseteq \hat{f}^{(n)}(\beta(i)) \qquad (*)$$

The proof is by induction on the number of execution steps. Here, i is the initial program state.

In practice, we usually allow the concrete semantics to be non-deterministic. This is necessary, for example, so that the static analysis soundly approximates the concrete execution on all possible, unknown inputs to the program. To implement the abstract transition function, we then need an upper bound operator to compute an over-approximation of multiple possible concrete states. A *least* upper bound operator is desirable for two reasons. The first reason is that a least upper bound yields the most precise abstraction of multiple states. For the second reason, we must consider how we will obtain a solution to the soundness property ($*$).

We seek a value $\hat{s}$ in the abstract domain such that $\forall n \geq 0 : \beta(f^{(n)}(i)) \sqsubseteq \hat{s}$. By ($*$), it is sufficient if $\forall n \geq 0 : \hat{f}^{(n)}(\beta(i)) \sqsubseteq \hat{s}$. Kleene's fixed point theorem provides an algorithm to compute such an $\hat{s}$, but it requires $\hat{S}$ to be a complete semi-lattice (to have least upper bounds), and $\hat{f}$ to be continuous and therefore monotone on that lattice ($\forall \hat{s} \sqsubseteq \hat{s}' : \hat{f}(\hat{s}) \sqsubseteq \hat{f}(\hat{s}')$). We must therefore ensure that $\hat{S}$ and $\hat{f}$ have those properties in order to use the algorithm.

We have laid out the general properties that we must to prove, but the domains $S, \hat{S}$ and functions $f, \hat{f}$ are usually intricate in practice, composed of many other smaller sets and functions. Proving the general properties requires proving similar lemmas on the constituent sets and functions.

For the example from Section 2, we may define the concrete state as the set of pairs of a program counter c and a map m from variables to integers. The abstract state could then be a map from program counters c to maps $\hat{m}$ from variables to elements of, say, the sign lattice. The function β would then be:

$$\beta(c, m) = \lambda c'. \begin{cases} \lambda v.\beta_{\mathbb{Z}}(m(v)) & \text{if } c = c' \\ \lambda v.\bot & \text{otherwise} \end{cases}$$

Here, $\beta_{\mathbb{Z}}(n)$ gives the sign of integer n. The concrete transition function f could be defined as follows:

$$f(c, m) = \begin{cases} (c', m[x \mapsto m(y) + m(z)]) & \text{if instr}(c) = x := y + z\ \& \\ & \quad c' = succ(c) \\ \ldots & \text{if instr}(c) = \ldots \end{cases}$$

The abstract transition function $\hat{f}$ could be defined as follows:

$$\hat{f}(\hat{s}) = \lambda c'. \begin{cases} \hat{s}(c)[x \mapsto \hat{s}(c)(y) \mathbin{\hat{+}} \hat{s}(c)(z)]) & \text{if instr}(c) = x := y + z\ \& \\ & \quad c' = succ(c) \\ \ldots & \text{if instr}(c) = \ldots \end{cases}$$

Note that the concrete and abstract transition functions have the same structure. In order to prove that $\forall s \in S : \beta(f(s)) \sqsubseteq \hat{f}(\beta(s))$, we need to prove similar statements on the constituent functions such as $+$ and $\hat{+}$, in particular that $\forall i, j \in \mathbb{Z} : \beta_{\mathbb{Z}}(i + j) \sqsubseteq \beta_{\mathbb{Z}}(i) \hat{+} \beta_{\mathbb{Z}}(j)$.

The take away is that a function $\widehat{f}$ is a sound over-approximation of a function f when:

$$\forall x.\, \beta(f(x)) \sqsubseteq \widehat{f}(\beta(x)) \qquad (\textsc{Soundness})$$

where the function β lifts a single concrete element into an element of the abstract domain and $\sqsubseteq$ is the partial order of the abstract domain. This is the property we must verify for every operation of an abstract domain.

3.2 Partial Orders and Lattices

In the previous section, we discussed the criteria necessary for soundness. We now turn our attention to the criteria for safety, specifically that partial orders, lattices, and operations on them satisfy a range of properties. Readers who are familiar with this material may want to skip to Section 4.

Partial Order. A *partial order* $\sqsubseteq$ is a binary relation on a set S that satisfies three properties:

$$\forall x.\, x \sqsubseteq x \qquad (\textsc{Reflexivity})$$
$$\forall x, y.\, x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y \qquad (\textsc{Anti-Symmetry})$$
$$\forall x, y, z.\, x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z \qquad (\textsc{Transitivity})$$

Intuitively, reflexivity states that an element must be equal to itself, anti-symmetry states that two different elements cannot be mutually less than each other, and transitivity states that if x is less than y, and y is less than z, then x must be less than z.

Least Element. A partial order may have a "smallest" or "least" element. This element is usually called bottom and denoted by the $\bot$ symbol. It must satisfy the property:

$$\forall x.\, \bot \sqsubseteq x \qquad (\textsc{Least-Element})$$

Greatest Element. Symmetrically, a partial order may have a "largest" or "greatest" element. This element is usually called top and denoted by the $\top$ symbol. It must satisfy the property:

$$\forall x.\, x \sqsubseteq \top \qquad (\textsc{Greatest-Element})$$

Least Upper Bound. A partial order may be equipped with a least upper bound operator $\sqcup$. The least upper bound must return an element that is greater than or equal to its two arguments:

$$\forall x, y.\, x \sqsubseteq (x \sqcup y) \land y \sqsubseteq (x \sqcup y) \qquad (\textsc{Upper-Bound})$$

And the element must be the *least* upper bound:

$$\forall x, y, z.\, x \sqsubseteq z \land y \sqsubseteq z \Rightarrow (x \sqcup y) \sqsubseteq z \qquad (\textsc{Least-Upper})$$

That is, the least upper bound must be the *smallest* of all the upper bounds of x and y. A partial order equipped with a least upper bound is called a *join semi lattice*.

Greatest Lower Bound. Analogously, a partial order may be equipped with a greatest lower bound operator $\sqcap$ which must return an element smaller than or equal to its two arguments:

$$\forall x, y.\, (x \sqcap y) \sqsubseteq x \land (x \sqcap y) \sqsubseteq y \qquad (\textsc{Lower Bound})$$

And the element must be the *greatest* lower bound:

$$\forall x, y, z.\, z \sqsubseteq x \land z \sqsubseteq y \Rightarrow z \sqsubseteq (x \sqcap y) \qquad (\textsc{Greatest Bound})$$

A partial order equipped with a greatest lower bound is called a *meet semi lattice*. A *lattice* is a partial order that is both a join and meet semi lattice.

3.3 Functions between Orders and Lattices

We recap two important properties of functions between lattices: strictness and monotonicity.

Strict Functions. A function $f : A \rightarrow B$ is *strict* if it maps the bottom $\bot_A$ in its domain A to bottom $\bot_B$ in its co-domain B:

$$f(\bot_A) = \bot_B \qquad (\textsc{Strict})$$

Monotone Functions. A function $f : A \longrightarrow B$ is *monotone* (also called *order-preserving*) if it satisfies the property:

$$\forall x, y.\, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y) \qquad \text{(MONOTONE)}$$

A function defined over multiple parameters is monotone if it is monotone in each parameter. Intuitively, when the input to a monotone function becomes "bigger", its output also becomes "bigger".

3.4 Pre-Orders and Normalization

A *pre-order* $\preceq$ (or *quasi-order*) is "almost" a partial order and must satisfy reflexivity and transitivity, but not necessarily anti-symmetry:

$$\forall x.\, x \preceq x \qquad \text{(REFLEXIVITY)}$$

$$\forall x, y, z.\, x \preceq y \land y \preceq z \Rightarrow x \preceq z \qquad \text{(TRANSITIVITY)}$$

Pre-orders are common in program analysis. For example, a typical definition of the interval "lattice" is $L = \{[b, e] \mid b, e \in \mathbb{Z} \cup \{-\infty, \infty\}\}$ where we identify $\top$ with $[-\infty, \infty]$ and $\bot$ with any element $[b, e]$ where $b > e$. So, $[1, -1]$ and $[5, -7]$ are smaller than each other, but they are not equal.

We can turn a pre-order into a partial order with a normalization function $\boxdot$. The function selects a representative among elements that are individually smaller than each other according to the pre-order. The normalization function thus re-establishes anti-symmetry and turns the pre-order into a partial order:

$$\forall x, y.\ \boxdot(x) \preceq \boxdot(y) \land \boxdot(y) \preceq \boxdot(x) \Rightarrow \boxdot(x) = \boxdot(y) \qquad \text{(ANTI-SYMMETRY)}$$

In the case of the interval "lattice" we can use a normalization function that maps any element $[b, e]$ where $b > e$ to the element $[1, -1]$, thus turning the pre-order on intervals into a partial order.

Applying the normalization function everywhere in a FLIX program is tedious. For this reason, we have been experimenting with a compiler extension that automatically applies the normalization function every time a value of a pre-order is constructed.

3.5 Finite Height Lattices

We want FLIX programs to eventually terminate, i.e. to reach the least fixed point in a finite number of steps starting from the bottom element(s) of the lattice(s). If a partial order satisfies the ascending chain condition we can prove that the analysis terminates.

Ascending Chain Condition. A partial order satisfies the *ascending chain condition (ACC)* if, for every chain, the sequence:

$$x_1 \sqsubseteq x_2 \sqsubseteq \cdots \sqsubseteq x_i \sqsubseteq x_{i+1} \cdots$$

is eventually stationary, i.e. there is a k such that $x_k = x_{k+1}$.

Proving the ascending chain condition is tricky for automatic program verification techniques. Instead, we aim to prove a stronger property with a little help from the analysis designer.

Let $\diamond : E \longrightarrow \texttt{Int}$ be a "height" or "termination" function mapping each element of a lattice to a decreasing integer. We require that:

$$\forall x.\ \diamond(x) \geq 0 \qquad \text{(NON-NEGATIVE)}$$

$$\forall x, y.\, x \sqsubseteq y \land x \neq y \Rightarrow \diamond(x) > \diamond(y) \qquad \text{(DECREASING)}$$

If the programmer supplies $\diamond$ and we are able to prove the two properties above, then the partial order satisfies the ascending chain condition [Nielson et al. 2005]. Consequently, the fixed point computation will eventually terminate.

4 VERIFICATION TOOLCHAIN

We now discuss how to express and use properties in FLIX and the implementation of the verification toolchain.

4.1 Annotations and Laws

In the previous section, we discussed the mathematical properties that user-defined lattices and functions must satisfy to ensure safety and soundness. We now turn to the real world and go into the details of how to express and use these properties in FLIX.

In FLIX, a property is expressed as a boolean valued function using the `law` keyword. For example,

```
law commutative[a, b](f: (a, a) -> b): Bool =
    ∀(x: a, y: a). f(x, y) == f(y, x)
```

declares a law named `commutative` which describes a property of a binary function `f` from values of type `a` to a value of type `b`. The body of the law contains a universal quantifier over two variables `x` and `y` both of type `a`. The quantified expression then asserts that `f(x, y)` must be equal to `f(y, x)`. As the example shows, laws are essentially polymorphic higher-order functions.

We associate a law with a function using the law name as an annotation. For example,

```
#commutative
def plus(x: Int, y: Int): Int = x + y
```

and

```
#commutative
def eq(x: Int, y: Int): Bool = x == y
```

assert that the two functions `plus` and `eq` must be commutative.

The FLIX compiler iterates through all annotations in the program and instantiates each law annotation. Specifically, the first argument of a law is the function value on which the annotation is placed. For example, the above code fragments give rise to the two property expressions `commutative(plus)` and `commutative(eq)`. The instantiated properties are collected and passed on to the verification toolchain. It is important to note that laws exist only at compile-time and do not affect the run-time semantics of FLIX.

The commutative law is clear and simple. For safety and soundness properties, the laws are more involved. Here is the general definition of monotonicity for binary functions:

```
 law monotone2[a, b, c](f: (a, b) -> c,
                        leqa: (a, a) -> Bool,
                        leqb: (b, b) -> Bool,
                        leqc: (c, c) -> Bool): Bool =
     ∀(x1: a, x2: b, y1: a, y2: b).
         ((x1 `leqa` y1) ∧ (x2 `leqb` y2))
             → (f(x1, x2) `leqc` f(y1, y2))
```

We can read the law as follows: Given a function `f` of two arguments of type `a` and `b` with a result type of `c`, and given partial orders on `a`, `b`, and `c` named `leqa`, `leqb`, and `leqc`, then the body of the law must be true. Specifically, for any values x_1 and y_1 of type `a` where $x_1 \sqsubseteq_a y_1$, and analogously for x_2 and y_2 of type `b` where $x_2 \sqsubseteq_b y_2$, it must be the case that $f(x_1, x_2) \sqsubseteq_c f(y_1, y_2)$. This is just a spelled out version of the monotonicity law.

Here is the general definition of soundness for binary functions:

```
law sound2[a1, a2, c1, c2](fa: (a1, a1) -> a2,
                           fc: (c1, c1) -> c2,
                           beta1: c1 -> a1,
                           beta2: c2 -> a2,
                           leq: (a2, a2) -> Bool): Bool =
    ∀(x: c1, y: c1).
        beta2(fc(x, y)) `leq` fa(beta1(x), beta1(y))
```

which captures the intuition discussed earlier, while taking the different partial orders into account.

We can use these laws on a function, for example,

```
#monotone2(leq, leq, leq)
def plus(e1: Sign, e2: Sign): Sign = ...
```

where `leq` is the partial order for elements of type `Sign`. Since this can become somewhat verbose and tedious to write, we can define a new law specialized to sign lattice elements:

```
law monotoneSign2(f: (Sign, Sign) -> Sign): Bool =
    monotone2(f, leq, leq, leq)
```

This defines a law named `monotoneSign2`, allowing us to write:

```
#monotoneSign2
def plus(e1: Sign, e2: Sign): Sign = ...
```

Annotation Inference. Most annotations can be inferred automatically from the declarations, constraints, and types of a Flix program. For example, if there is a declaration:

```
    lat In(s: Stm, x: Var, v: Parity).
```

then we know that `Parity` must be a lattice and satisfy all the properties of a lattice. Similarly, if there is a rule:

```
Out(s, r, sum(v1, v2)) :-
    AddStm(s, r, x, y), In(s, x, v1), In(s, y, v2).
```

then we know that the function `sum` must be strict and monotone, and we know the types of its arguments, so we can find the appropriate partial orders (a type has at most one partial order).

The only annotations that cannot be inferred for a Flix program are the soundness annotations, since Flix does not know the concrete semantics of the program under analysis.

4.2 Symbolic Evaluation

The symbolic evaluator is the main component of the verifier. Its job is to take a property, which is a boolean valued expression that may contain universal quantifiers, and determine whether it is true, or if not, to produce a counter-example. The symbolic evaluator works similarly to a normal interpreter, but where a normal interpreter would have a signature like:

```
def eval(e: Expression, env: Map[Symbol, Value]): Value
```

the signature of the symbolic evaluator is:

```
def eval(e: Expression,
        env: Map[Symbol, SymVal]): List[(PathConstraint, SymVal)]
```

where `SymVal` is a symbolic value, i.e. a symbolic variable or a regular normal value, and `PathConstraint` is a first-order formula over the theory of bitvectors, floating-point numbers, and integers. Intuitively, whereas the normal interpreter returns a single value, the symbolic evaluator returns a set of (symbolic) values where each value is a possible outcome if and only if the path constraint has a model, i.e. a satisfying assignment.

The symbolic evaluator works as follows: The evaluator performs concrete execution whenever possible. When it encounters a quantified expression, e.g. $\forall x : \texttt{Parity}.\, e$ where e is an expression, it inspects the type of the quantified variable and attempts to instantiate it. For example, if the type is `Parity`, then the quantifier is instantiated to the (concrete) values $\bot$, $\top$, `Even` and `Odd`. If the type is not enumerable, e.g. the type contains an integer, then a symbolic value is instantiated. For example, if the type is `Constant` (the constant propagation lattice), then the three values $\bot$, $\top$, and $\texttt{Cst}(x)$ are instantiated, where x is a fresh symbolic variable. If every quantifier has an enumerable type, then symbolic evaluation corresponds to concrete execution, but that is rare.

Whenever the symbolic evaluator encounters a computation with a symbolic value where it cannot continue concrete evaluation, it grows the path constraint and, if necessary, returns a new symbolic value to represent the result of the computation. For example, if the symbolic evaluator encounters the expression `x == 5`, where x is a symbolic variable of type `Int`, it returns two values, `true` and `false`, where the first value is guarded by the path constraint $x = 5$ and the second is guarded by $x \neq 5$. If, on the other hand, the symbolic evaluator encounters an expression `x + 5`, then it returns a fresh symbolic variable y to represent the result with the path constraint $y = x + 5$. As a consequence, in branches where the condition cannot be determined, the evaluator proceeds along both paths under different path constraints.

Eventually, the symbolic evaluator returns a list of (path constraint, symbolic value)-pairs. Since every property is boolean valued, the value is either `true` or `false`. In the rare case that a user-defined function contains an infinite loop, e.g. due to infinite recursion, the symbolic evaluator loops forever and never returns. However, in practice, this often overflows the stack and crashes the program with a stack overflow error.

For each returned value and path constraint, we consider:

If the returned value is `true` under the path constraint `pc`, then the property *holds* regardless of whether the path is realizable or not. That is, if the path constraint is realizable, then the symbolic execution is a possible concrete execution and it must return `true`, i.e. the property holds. If, on the other hand, the path constraint is unrealizable, then the symbolic execution does not correspond to *any* concrete execution and can safely be ignored. In other words, we can ignore the path constraint entirely; we do not need to feed it to the SMT solver.

If the returned value is `false` under the path constraint `pc`, then the property may or may not hold. If the path constraint is realizable, then we have found a counter-example, whereas if the path constraint is unrealizable, then the symbolic execution does not correspond to any concrete execution, and it can safely be ignored. To determine which is the case, we feed the path constraint to the SMT solver. The SMT solver may do one of four things:

- The SMT solver may return "SAT" and a model (an assignment of the free variables in the path constraint). If so, we have found a concrete execution that returns `false`, i.e. a counter-example to the property. We extract the free variables from the model and recompose them with some information about how we instantiated the quantified variables earlier, and we report the counter-example to the user.
- The SMT solver may return "UNSAT", which means that the path constraint has no solution and so does not correspond to any concrete execution. In this case, we have dodged a bullet, and the property still holds.
- The SMT solver may return "UNKNOWN" if the path constraint belongs to an undecidable fragment of logic or if the

SMT solver took longer than a user-specified timeout. In both cases, we have to conservatively assume that the property does not hold. Unfortunately, since there is no model, we cannot report a counter-example to the programmer.
- The SMT solver may loop forever if the path constraint belongs to an undecidable fragment of logic. As stated above, we handle this with a user-specified timeout and report to the programmer that it is unknown if the property holds.

We perform all the above steps for every property in the program. If a property fails to hold, the verifier issues an error message and the counter-example, as shown in Section 2.

Language Support. The symbolic evaluator supports the entire Flix language, including algebraic data types, pattern matching, closures, higher-order functions, arbitrary and machine-sized arithmetic, and floating-point arithmetic. In fact, a specific design goal of the functional language in Flix was for it to be amenable to automatic program verification techniques.

4.3 Quick Checker

If the symbolic evaluator or the SMT solver fail to verify a property, all is not lost. We can fall back on testing as an empirical way to gain some confidence that our program is correct. Quick checking is an automatic test generation technique originally proposed by Classen and Hughes for the Haskell programming language [Claessen and Hughes 2011]. The idea is that given a universally quantified formula, a quick checker randomly instantiates each quantifier and evaluates the formula to determine whether or not it holds for the specific input. In essence, quick checking is a form of randomized testing, but it is smart about how it selects its inputs.

Inspired by Midtgaard and Møller [2015], we have implemented a quick checker on top of our verification infrastructure. Naturally, if the verifier has proven a property, there is no reason to test or quick check it, but in cases where the SMT solver reported "UNKNOWN", quick checking can serve as a useful fallback. For details of how to implement such a quick checker, we refer the reader to [Midtgaard and Møller 2015].

4.4 Limitations

Undecidability. A limitation of our approach is decidability of the path constraints generated by the symbolic evaluator. For example, linear integer constraints are decidable, whereas non-linear arithmetic is undecidable. Undecidability is an unfortunate, but inherent limitation of any fully automatic verification technique. In Section 5, we experimentally evaluate how often the SMT solver reports "UNKNOWN".

Recursion and Infinite Loops. Another limitation of the symbolic evaluator is that it cannot prove properties of recursive functions unless the arguments happen to be known statically, which is rarely the case. If the symbolic evaluator encounters a recursive call, it will enter that call, and if the arguments are symbolic, it will continue the recursion forever. What is needed is an induction hypothesis that can be applied at the place of the recursive call. One option would be to ask the programmer for the induction hypothesis. This is something we want to explore more in the future.

Correctness of Constraints. We have discussed how to verify the soundness of the abstract domains and the operations on them. The abstract domains are the building blocks from which the static analysis is constructed. Specifically, as shown in Section 2, the static analysis is formulated as a set of constraints on these domains. Each constraint is a Horn clause that may use the functions which implement the abstract operations of the abstract domains. The constraints are *safe* by construction, i.e. strict and monotone, provided that the functions on the abstract domain are. The *soundness* of the constraints, and hence of the overall analysis, remains the responsibility of the analysis designer.

In practice, the complete concrete semantics of a language is often not known, and much of the job of the analysis designer is to actually determine and specify the high level structure of that semantics. On the other hand, the semantics of the underlying operations, such as integer arithmetic, is well known. Thus, it is feasible and useful to automatically verify the soundness of these underlying operations. For the high level structure, we have to trust that the analysis designer has understood the semantics correctly and structured the constraints of the Flix program accordingly.

5 EVALUATION

In this section, we evaluate the Flix verification toolchain on several lattices from the book *Introduction to Lattices and Order* [Davey and Priestley 2002] and on several well-known abstract domains from the static analysis literature.

5.1 Implementation Details

The Flix compiler and run-time is currently 40,000 lines of Scala code. Of these, the symbolic evaluator is 1,500 lines of code, the infrastructure for laws and properties is 1,000 lines of code, and the interface for the SMT solver is around 500 lines of code. The symbolic evaluator handles all aspects of the Flix language, including closures, algebraic data types, and pattern matching. Flix uses the Z3 SMT solver from Microsoft [Moura and Bjørner 2008]. The Flix implementation is open-source and freely available online[1].

5.2 Case Study: Basic Lattices

We selected a variety of lattices from *Introduction to Lattices and Order* [Davey and Priestley 2002]. For each lattice, we defined the bottom element, the top element, the partial order, the least upper bound and the greatest lower bound. For details on these lattices, we refer the reader to the book.

Results. Table 1 shows the results of running the verifier on these nine lattices. For each lattice, we collected the following information: the number of lines of code used to implement the lattice, the number of properties to check, the number of paths executed by the symbolic evaluator, the number of queries issued to the SMT solver, and the total time spent by the verifier. As an example, the Cube lattice is implemented in 127 lines of Flix code, there were 13 properties to check, the symbolic evaluator encountered 2,904 paths during evaluation, 0 SMT queries were issued, and the verifier took 2.0 seconds. No SMT queries were issued by the symbolic evaluator because the lattice is finite and

[1]Official website: http://flix.github.io. GitHub repository: http://github.com/flix/flix

Table 1: Verification of lattices from Davey and Priestley [2002].

Lattice	Lines	Properties	Paths	Queries	Time
Cube	127	13	2,904	-	2.0s
InfNoAccNoDcc	59	13	189	340	2.9s
M_2	48	13	412	-	0.4s
M_3	51	13	765	-	0.5s
$M_2 + M_3$	194	13	4,077	-	2.6s
N_5	61	13	765	-	1.2s
Sub D_4	147	13	5,530	-	2.8s
Sub Z_2 Z_4	121	13	2,904	-	1.7s
Sub Zero★	282	13	11,869	-	9.1s
Totals	1,090	117	29,415	340	23.2s

thus the symbolic evaluation amounted to exhaustive enumeration and concrete evaluation of all the quantifiers in each property. Even for these basic lattices, the number of paths can be quite large and thus the burden for the programmer to verify these properties manually would be quite substantial. The running times range from less than half a second to around nine seconds. This is slower than a typical run of the Flix compiler, but still fast enough that the verifier can be run repeatedly during program development.

We were able to automatically verify all the required properties of the lattices. We did find some bugs during development, and in most cases, these were simple transcription errors or copy-paste errors. This suggests that even for "trivial" lattices that were merely transcribed from a book, subtle errors can and do creep in. In summary, we verified 117 properties for nine lattices implemented in around 1,000 lines of Flix code.

5.3 Case Study: Abstract Domains

We implemented eleven abstract domains from the static analysis literature. Many of these abstract domains are classic textbook examples, e.g. the 4-valued boolean Belnap domain, the constant propagation domain, interval domains, and sign domains. To have some richer or significantly more complicated domains, we constructed the reduced product of several of these domains, e.g. the reduced product of the constant propagation and parity domain.

Table 2 shows the results of verifying each of these domains.

For lack of space, we only briefly describe each abstract domain. `Belnap` is the classic four-valued boolean domain. `Constant` is the flat constant propagation domain. `Interval` is the interval domain with the use of a normalization function as described earlier. `IntervalAlt` is an alternative interval domain where every interval larger than ten is mapped to the top element. `Parity` is the classic parity domain. `PrefixSuffix` is a domain that tracks the first and last character of a string. `Mod3` is a modulo-3 domain. `Sign` and `StrictSign` are the domains described earlier. Finally, `ConstParity` and `ConstSign` are *reduced products*[2] of the constant domain and the parity and sign domains, respectively. The abstract domains are described in greater detail in the technical report.

Results. Table 2 shows the results of running the verifier on these eleven abstract domains. The structure of the table is similar to Table 1, but includes some extra columns. The column `Abstracts` refers to the concrete domain approximated by the abstract domain. The `Ops` column refers to the number of operations that were implemented on this lattice, excluding the partial order, least upper

Table 2: Verification of abstract domains.

Domain	Abstracts	Ops	Lines	Safety	Soundness	Paths	Queries	Time
Belnap	Boolean	7	177	38 / 38	7 / 7	2,353	0	0.8s
Constant	Int32	17	268	61 / 61	17 / 17	2,344	132	2.3s
Interval	BigInt	9	184	22 / 23	9 / 9	23,520	3,484	29.5s
IntervalAlt	BigInt	9	199	25 / 26	9 / 9	170,650	23,091	187.0s
Parity	Int32	17	296	61 / 61	17 / 17	4,637	39	2.3s
PrefixSuffix	String	3	147	24 / 24	0 / 3	29,416	10,209	75.3s
Mod3	BigInt	9	209	39 / 39	8 / 9	6,755	596	7.2s
Sign	BigInt	9	233	37 / 37	9 / 9	5,600	38	2.8s
StrictSign	BigInt	9	226	39 / 39	9 / 9	5,750	80	2.3s
ConstParity	BigInt	9	239	35 / 36	9 / 9	11,448	1,540	14.1s
ConstSign	BigInt	9	233	37 / 37	9 / 9	11,672	924	9.7s
Totals		107	2,411	419 / 421	103 / 107			

bound, etc. The columns `Safety` and `Soundness` refer to the numbers of safety and soundness properties that were present and that were verified. For example, we implemented 7 operations (excluding $\sqsubseteq$, $\sqcup$, and $\sqcap$) for the `Belnap` domain in 177 lines of Flix code. This resulted in 38 + 7 properties to be checked. The breakdown of these were 1x Reflexivity, 1x Anti-Symmetry, 1x Transitivity, 1x Least-Element, 1x Upper-Bound, 1x Least-Upper-Bound, 1x Greatest-Element, 1x Lower-Bound, 1x Greatest-Lower-Bound, 1x Non-Negative, 1x Decreasing, 7x Strictness, 7x Monotonicity, 7x Commutativity, and 6x Associativity, together with 1x Sound-1, and 6x Sound-2. The symbolic evaluator encountered 2,353 paths and issued 0 SMT queries. The total verification time, for this domain, was 0.8 second.

In total, we proved 419 out of 421 (99.5%) safety properties and 103 out of 107 soundness properties (96.3%). For the interval lattices, we failed to prove 2 safety properties due to uses of multiplication. For the `PrefixSuffix` lattice, we failed to prove 3 soundness properties due to the lack of string support in the SMT solver that we used. Finally, for the `Mod3` lattice, we failed to prove 1 soundness property due to multiplication.

We investigated the long running time of the verifier on the `Interval` and `PrefixSuffix` domains. We see that the number of explored paths and SMT queries is significantly higher than for the other domains. We looked into the details of each property and discovered that in most cases just a few properties are responsible for the majority of paths and SMT queries. For example, for the `PrefixSuffix` lattice, we discovered that `associativity` of the least upper bound explored 14,000 paths, issued 6,000 queries, and took 60s. If verification time is a concern, an option might be to set a small timeout for each property, which would cause the verification to complete quickly at the cost of failing to prove a few properties.

In summary, we implemented 11 abstract domains with 107 operations in 2,411 lines of Flix code. This gave rise to 421 safety and 107 soundness properties. Of these, we were able to prove 99.5% and 96.3% of the properties automatically.

Discussion. We used the verifier during the implementation of these lattices, and in our experience it was quite helpful. For example, we would frequently implement a function and then run the verifier to check that the function was correct. Furthermore, we found that when a mistake was made, frequently multiple properties would be broken (e.g. both Upper-Bound and Commutativity), and looking at the multiple counter-examples was more helpful than looking at a single counter-example.

[2]Note: The reduced product should *not* be confused with the Cartesian product.

6 RELATED WORK

This paper draws together several fields: the design and implementation of static analysis frameworks, program verification techniques, and dynamic symbolic execution.

Static Analysis Frameworks. Over the years, several static analysis frameworks have been proposed. PAG is a dataflow analysis framework that generates C code from the specification of lattices and transfer functions [Martin 1998]. WALA is a static analysis library written in Java [Fink and Dolby 2012]. WALA implements many popular static analyses, including points-to analysis, class hierarchy analysis, and the IFDS algorithm [Reps et al. 1995]. Soot is a Java bytecode analysis framework that has been widely used as a frontend for other static analyses [Vallée-Rai et al. 1999]. Doop is a points-to analysis framework for Java [Bravenboer and Smaragdakis 2009; Smaragdakis and Bravenboer 2011]. Hoopl is a dataflow analysis and transformation framework written in Haskell [Ramsey et al. 2010]. Frama-C is a static analysis framework for C programs [Cuoq et al. 2012].

A common limitation of these frameworks is that they assume the user-defined implementations of abstract domains and their operations are correct (safe and sound). As far as we know, Flix is the first static analysis "framework" that attempts to verify these implicit assumptions. If a Flix program passes the verifier, then the programmer can be confident that his or her program satisfies the necessary properties for the fixed point computation to terminate with a sound deterministic result.

Verified Static Analyzers. CompCert is a formally verified realistic compiler for the C programming language [Leroy 2006, 2009]. The compiler is implemented and proven correct using the Coq proof assistant [Barras et al. 1997; Bertot and Castéran 2013; Chlipala 2013]. The correctness proof guarantees that the compiler preserves the semantics of the original program through the phases of the compiler all the way down to machine code. Proving correctness of CompCert was a significant task and required a multiple person-year effort. The compiler is itself roughly 25% program code and 75% proof code. Multiple static analyses have been implemented on top of CompCert: (a) an intra-procedural, flow-sensitive, and field-sensitive alias analysis [Robert and Leroy 2012], (b) a value analysis with a non-relational interval domain [Blazy et al. 2013], and (c) a larger static analyzer, Verasco, which includes non-relational integer interval and congruence domains, and relational polyhedra and symbolic equalities domains [Jourdan et al. 2015].

Flix and Coq offer complementary approaches to the same problem, but starting from opposite ends. The Flix verifier is fully automatic and imposes no extra burden on the programmer. On the other hand, it is limited to fairly simple abstract domains defined without the use of recursion. Coq, in contrast, requires a lot of extra work from the programmer, but can prove properties involving inductive definitions. We do not think either approach is better; ultimately it depends on the type of analysis being implemented, and the amount of effort the analysis designer is willing to spend.

Randomized Testing. QuickCheck is a technique, originally developed for Haskell, which performs random light-weight testing [Claessen and Hughes 2011]. In QuickCheck, the developer or library specifies the properties that some mathematical object must satisfy and provides methods to randomly generate instances of those objects. QuickCheck then attempts to falsify the property by generating input aimed at interesting corner cases. Quick Checking has been adapted to test the correctness of lattices.

In relation to Flix, verification offers a stronger guarantee than testing: If we have proven some property, there is no reason to test that property. However, in cases where the verifier is unable to prove a property, testing can be used as a fallback. Inspired by [Midtgaard and Møller 2015], we have implemented a quick checker on top of the Flix verification infrastructure. One nice extension of this work is that our quick checker also checks soundness properties.

Symbolic Evaluation. The idea of symbolic execution goes back more than three decades [Boyer et al. 1975; King 1976]. Since then, a lot of research effort has been devoted to *concolic testing*, which executes the program both symbolically and concretely [Godefroid et al. 2005; Sen and Agha 2006]. The symbolic evaluator in Flix is a fairly straightforward implementation of these classic ideas with a special focus on (a) handling all aspects of the Flix language and (b) giving high-quality error messages.

7 CONCLUSION

In this paper we have proposed the use of automatic program verification techniques to ensure the safety and soundness of static analysis tools. Safety guarantees that the fixed point computation converges and terminates with a deterministic result. Soundness ensures that operations on abstract domains over-approximate their corresponding concrete operations.

We have implemented a verification toolchain in Flix, a functional and logic programming language for the implementation of static analysis tools. If the specification of the static analysis, i.e. the constraints of the Flix program, is sound, the verifier can be used to guarantee that the overall analysis is sound.

We have applied the verification toolchain to several abstract domains. The experiments showed that we were able to prove 99.5% of the safety and 96.3% of the soundness properties. The experimental results suggest that the use of automatic verification techniques is feasible and can help increase confidence in the overall correctness of static analyses.

ACKNOWLEDGMENT

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, and Others. 1997. *The Coq Proof Assistant Reference Manual.* Ph.D. Dissertation.

Yves Bertot and Pierre Castéran. 2013. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.* Springer.

Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. 2013. Formal Verification of a C Value Analysis Based on Abstract Interpretation. In *International Static Analysis Symposium (SAS).*

Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT — a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proc. of the International Conference on Reliable Software.*

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-To Analyses. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*

Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What You Always Wanted to Know About Datalog (and Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering (TKDE)* (1989).

Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press New York.

Koen Claessen and John Hughes. 2011. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *International Conference on Functional Programming (ICFP)* (2011).

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C – A Software Analysis Perspective. In *Software Engineering and Formal Methods (SEFM).*

Brian Davey and Hilary Priestley. 2002. *Introduction to Lattices and Order.* Cambridge University Press.

Stephen Fink and Julian Dolby. 2012. WALA – The TJ Watson Libraries for Analysis.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proc. Programming Language Design and Implementation (PLDI).*

Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *Proc. Principles of Programming Languages (POPL).*

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* (1976).

Xavier Leroy. 2006. Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant. In *Proc. Principles of Programming Languages (POPL).*

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* (2009).

Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016a. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proc. Programming Language Design and Implementation (PLDI).*

Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016b. Programming a Dataflow Analysis in Flix. In *Tools for Automatic Program Analysis (TAPAS).*

Magnus Madsen, Ramin Zarifi, and Ondřej Lhoták. 2018. Tail Call Elimination and Data Representation for Functional Languages on the Java Virtual Machine. In *Proc. International Conference on Compiler Construction (CC).*

Florian Martin. 1998. PAG – An Efficient Program Analyzer Generator. *The International Journal on Software Tools for Technology Transfer* (1998).

Jan Midtgaard and Anders Møller. 2015. QuickChecking Static Analysis Properties. In *Software Testing, Verification and Validation (ICST).*

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS).*

Flemming Nielson, Hanne Nielson, and Chris Hankin. 2005. *Principles of Program Analysis.* Springer.

Norman Ramsey, Joao Dias, and Simon Peyton Jones. 2010. Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation. In *Proc. Haskell Symposium.*

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proc. Principles of Programming Languages (POPL).*

Valentin Robert and Xavier Leroy. 2012. A Formally-Verified Alias Analysis. In *International Conference on Certified Programs and Proofs (CPP).*

Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *Computer Aided Verification (CAV).*

Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Datalog Reloaded.*

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot – A Java Bytecode Optimization Framework. In *Proc. Centre for Advanced Studies on Collaborative Research (CASCON).*