Comprehensive Reachability Refutation and Witnesses Generation via Language and Tooling Co-Design

MARK MARRON, Microsoft Research, USA
DEEPAK KAPUR, University of New Mexico, USA

This paper presents a core programming language calculus, BosqueIR, that is uniquely suited for automated reasoning. The co-design of the language and associated strategy for encoding the program semantics into first order logic enables the translation of BosqueIR programs, including the use of collections and dynamic data structures, into decidable fragments of logic that are efficiently dischargeable using modern SMT solvers. This encoding is semantically *precise* and logically *complete* for the majority of the language and, even in the cases where completeness is not possible, we use heuristics to precisely encode common idiomatic uses. Using this encoding we construct a program checker BSQChk that is focused on the pragmatic task of providing *actionable* results to a developer for possible program errors. Depending on the program and the error at hand this could be a full proof of infeasibility, generating a witness input that triggers the error, or a report that, in a simplified partial model of the program, the error could not be triggered.

1 INTRODUCTION

This paper introduces a novel core programming language calculus, BosqueIR, a strategy for encoding the language semantics into first order logic, and the BSQChk program checker. These three components are designed as a group with the goal of creating a programming system that was highly amenable to automated reasoning and mechanization using SAT Modulo Theory (SMT) theorem provers [3, 11].

A core design principle of the BosqueIR language is the elimination of foundational sources of undecidability and complexity commonly seen in programming language semantics – iteration/recursion, mutability, and referential observability. The BosqueIR semantics also employ carefully chosen definitions of language features, including sources of non-determinism, numeric type definitions, formalizing various out-of-* limits, etc., to ensure the encodings to first order logic are simple, compact (or at least finite), and easily dischargeable. This regularized programming model enables us to encode most of the language semantics in fully *precise* strongest postcondition form using decidable fragments of first order logic.

By construction the base theories needed by the BSQChk checker for the core BosqueIR operations are limited to *uninterpreted functions*, *integer arithmetic*, and *bitvectors* UF+IA+BV [3]. The extension to include most container operations introduces several quantified formula forms that are contained in the decidable theory of *quantified bitvector formula* QBVF [56]. As all of these theories are decidable, the BSQChk system can refute or generate witnesses for many errors in BosqueIR programs. Even in the presence of general recursion the BSQChk checker has a pair of refutation and model generation algorithms that can precisely handle common idiomatic forms while safely falling back to best-effort modes when unavoidable.

Our approach to program checking, and thus the design of the BSQChk checker, is pragmatic. Ideally a developer would like to have a fully automatic proof that, under all possible executions, a given state is unreachable or to get debuggable witness input in the cases when the state is reachable (*e.g.* a bug exists). In practice this proof or input may not be possible to generate as it may involve recursive code our heuristics do not cover or the theorem prover may be unable to discharge the query in a reasonable time frame. Thus, we consider the following hierarchy of

Authors' addresses: Mark Marron, Microsoft Research, USA, marron@microsoft.com; Deepak Kapur, Computer Science, University of New Mexico, USA.

50 *confidence boosting* results that the BSQCHK checker can produce such that in all cases it is able to
51 provide useful feedback to a developer:
52
53 1a. Proof that the error is infeasible in all possible executions
54 1b. Feasibility witness input that reaches target error state
55 2a. Proof that the error is infeasible on a simplified set of executions
56 2b. No witness input found before search time exhausted
57
58 The 1a and 1b cases are our ideal outcomes where the checker either proves that the error is
59 infeasible for all possible executions or provides a concrete witness that can be used to debug the
60 issue. The 2a and 2b cases represent useful *best effort* results. While they do not entirely rule out
61 (or witness) a given error, they do provide a substantial boost in a developers confidence that the
62 error is infeasible on a subset of inputs [44].
63 To maximize the ability of the system to produce useful outcomes in this hierarchy we leverage
64 the unique features of the BosqueIR language and novel encodings into efficiently checkable
65 fragments of first order logic to power the following features:
66
67 **Large & Inductive Model Capable** – The BSQCHK checker is able to do witness input gen-
68 eration and refutation proofs even when they require reasoning over a large model (Sec-
69 tion 2.1). In the case of general reduction/recursion (Section 2.4) the checker has heuristics
70 to generate a witness input or proof without unrolling inductive operations.
71 **Small Model Optimized** – If there is a small witness input model then the BSQCHK checker
72 can find it using mostly (or entirely) quantifier free solving (Section 2.2). For refutation we
73 also use small width bit-vector sizes to determine if a state is unreachable in an limited
74 approximation of the program (Section 2.3).
75 **Focused Search Structure** – In the case where the BSQCHK checker heuristics are unable
76 to generate a witness or establish a refutation proof, it falls back to a classic unroll+explore
77 strategy. Even in this scenario, the restrictions of the BosqueIR language and the encoding
78 maximize the amount of content in efficiently decidable logic UF+IA+QBVF fragment and
79 minimizes the state space that must be explored with iterative unrolling (Section 2.5).
80
81 The code in Figure 1 shows a BosqueIR implementation of a business application modeling
82 example from Morgan Stanley's Morphir framework [39]. The code snippet is focused on the
83 `availability` function. This function computes the number of items still available to sell based
84 on the number at start of the day (`startOfDayPosition`) and the list of buy transactions (`buys`) so
85 far. As a precondition it asserts that the `startOfDayPosition` is non-negative and that the return
86 value `$return` is bounded by the start of day value.
87 The code to compute the number of buy transactions that have been completed successfully
88 and the sum of the quantities from these purchases is concisely expressed using the functor chain
89 `buys.filter`$_{BuyAccepted}$`().map`$_{x.quantity}$`().sum()`. While this code is conceptually simple from a developer
90 viewpoint, the actual strongest postcondition logic semantics for it are quite complex. They include
91 a subset relation and predicate satisfaction relation on the `filter`, a quantified user defined binary
92 relation with the `map`, and an inductively defined relation as a result of the `sum`. Thus, trying to
93 prove that the postcondition is satisfied (or finding an input that demonstrates the error is possible)
94 is a challenging task involving inductive reasoning, relationships between container sizes and
95 contents, and quantified formula.
96 Despite these complexities the BSQCHK checker can model this code, in strongest postcondition
97 form, as a logical formula in a decidable fragment of first-order logic! Further, the BSQCHK prover
98 can instantaneously solve this formula.

99
100
101
102
103
104
105
106
107
108
109
110

```
function availability(startOfDayPosition: BigInt, buys: List_BuyResponse): BigInt
    requires startOfDayPosition >= 0
    ensures $return <= startOfDayPosition
{
    let sumOfBuys = buys.filter_BuyAccepted().map_x.quantity().sum() in
    startOfDayPosition - sumOfBuys;
}

...

entity BuyAccepted provides BuyResponse {
    field productId: String;
    field price: Decimal;
    field quantity: BigInt; //<--- should be BigNat
}
```

111 Fig. 1. BosqueIR implementation of order processing code.

112
113 Using the following satisfying assignment as a witness input a developer can run the application,
114 investigate the problem, and identify the appropriate course of action to resolve the issue.

115
$$startOfDayPosition = 0 \land buys = List_{BuyResponse}@\{BuyAccepted\{``a", 0.0, -1\}\}$$

116 In this case the fix is, using the fact that the BosqueIR language supports `BigNat` in addition to
117 `BigInt` numbers to ensure that the buy quantity is always non-negative.
118 With this simple change the BSQChk tool can be run again, and even with the complexity of the
119 logical structure, will instantaneously return that the program state where the `ensures` clause is
120 $false$ is unreachable. All of this analysis and proving is fully automated and does not require any
121 assistance, knowledge of the underlying theorem prover, or use of specialized logical assertions by
122 the developer.
123 This example shows how, carefully constructing the programming language with the specific
124 intent of being translatable to efficiently solvable logic, enables the construction of tooling with
125 sophisticated automated reasoning capabilities that provides compelling developer experiences.
126 As such, this result represents an important step in the transformation of programming from a
127 human labor intensive task into one where human ingenuity is augmented with a powerful set of
128 automated tools. The contributions[1] of this paper are:

129
130
- An core language calculus, BosqueIR, that is designed specifically in conjunction with and
131 to support automated program analysis using modern SMT solvers (Section 3).
132
- Encoding the core of the BosqueIR language into a efficiently decidable fragment of first-
133 order-logic, UF+IA+BV and optionally *real arithmatic* RA (Section 3).
134
- Encoding containers and operations on them (excluding *reduce* operations) into the decidable,
135 QBVF, fragment of first-order logic (Section 4).
136
- Heuristics for processing container *reduce* operations and arbitrary recursion, without
137 unrolling that remains in the UF+IA+QBVF fragment (Section 5).
138
- Introduces the BSQChk checker (Section 2) which uses these techniques + novel method-
139 ologies enabled by the language and encoding to system to produce a tool that is effective
140 at both refutation and witness generation in a range of scenarios.

141
2 BSQCHK OVERVIEW

142
143 The BSQChk checker first builds the code under analysis and enumerates all possible error condi-
144 tions. For each identified error the BSQChk checker follows the algorithm shown in Figure 2.

145
[1]This system forms the foundation of the Bosque Programming Language project under development at Microsoft Research
146 and availble as open source software – https://github.com/microsoft/BosqueLanguage.
147

148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166

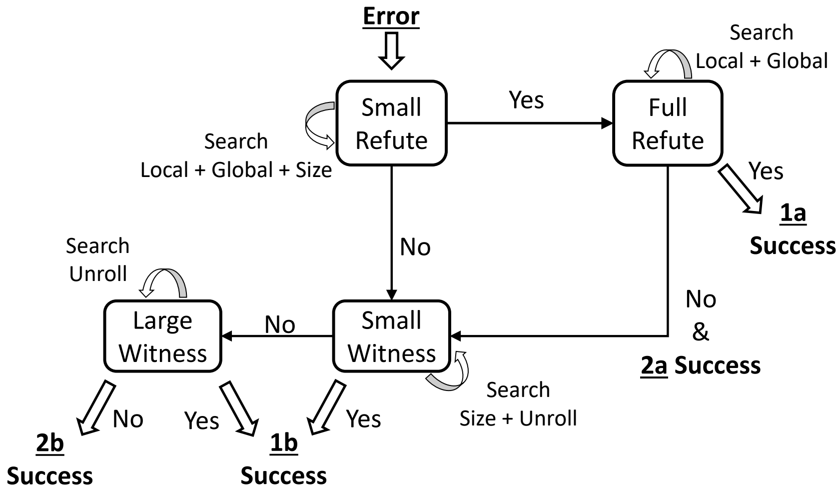

Fig. 2. BSQCHK checker workflow.

167 The first step in this algorithm is to see if the error can be refuted under various definitions of
168 simplified models of the program. As a low cost first check the refutation proofs are all attempted
169 without considering any call context. If this fails then the proofs are attempted using the, more
170 expensive, whole program encoding. In both the local and whole program proofs, small bit-widths
171 are used and increased in size up to 16 bits. If the small bitwidth proofs are successful refutations
172 (either locally or globally) then then system attempts a refutation proof with the full bit-widths
173 (first locally and then whole program). If either of these is successful then the checker has shown
174 that the error is infeasible on all executions and we achieved the highest quality, 1a, confidence
175 level.

176 If the refutation proofs fail then we move on to searching for a witness input for the error. If
177 we succeeded in proving the error infeasible for the small bitwidths, but then failed to prove the
178 infeasibility for the full case, we also achieved the partial, 2a, confidence level as well. In the small
179 witness search we incrementally expand the bitwidths up to size 16 and gradually increase the
180 limit of unrolling allowed for recursive calls – based on timeout limits. If we find an input that
181 reaches the target error then we have succeeded in producing a high value actionable result for the
182 developer, 1b, in our quality confidence level. With this result we know there is a real failure and
183 have a small input that can be used to trigger and debug it.

184 In the case we cannot generate a small witness we make a final witness generation attempt with
185 the full bitwidth and, again, perform iterative unrolling of the remaining recursive calls. As in
186 the small input case, if we find an input that reaches the target error then we have succeeded in
187 producing a high value actionable result for the developer. Otherwise we produce our minimal
188 success result, 2b, where we aggressively explored the input space and, while we cannot fully rule
189 our the feasibility of the error, we believe it is very unlikely to be triggered in practice.
190

191 ## 2.1 Large Model Witness Generation

192 Symbolic fuzzers or model checking [16, 23] techniques explore the input/execution space using
193 heuristics to iteratively unblock control flow paths (either conditionals or loop iterations). Other
194 techniques use input structure mutation and generation [57] to iteratively cover larger sets of the
195 relevant input space for a program. These techniques have proven highly effective at finding bugs
196

197 with a *small scope hypothesis* [24] – *i.e.* there exists a small input and/or small execution trace that
198 will exercise the error. In practice many errors have this property but there are important scenarios,
199 like cache invalidation logic or large lookup tables, where this hypothesis does not hold and these
200 techniques will be of limited effectiveness. Consider the code:

```
const cl = List_Int@{0, 1, 2, ..., 254, 333, 256 };

function largeunroll(x: Int): Bool {
  if (!cl.contains(x)) then
    false
  else
    let i = cl.findIndex(x) in
    i == 255
}

function largeinput(l: List_Int): Bool {
  l.size() == 256 && l.get(255) == 333 && !l.slice(0, 255).contains(333)
}
```

211
212 For techniques that use iterative state space expansion the `largeunroll` function requires
213 unrolling the logic in `contains` and `findIndex` functions 255 steps before they observe the function
214 returning `true`. This can be even more challenging for input state generation based exploration
215 techniques which need to enumerate from the space of valid integers to find a single value. In the
216 case of the `largeinput` function the problem becomes more complex as both the iteration space
217 *and* the input need to be large, a List of 256 elements, in order to reach the path where the function
218 returns `true`. While heuristics exist to increase the likelihood of finding these errors, *e.g.* seeding
219 the input generator with constants from the program, in general iterative exploration techniques
220 will exhaust their resource bounds without finding the error state.
221 The use of QBVF encodings enable the SMT solver to reason from both constraints related to the
222 input and the target error. This allows it to easily generate satisfying witness values for the input
223 that reach the target error state. Consider the formula that needs to be solved for the `largeunroll`
224 function – which is trivially solvable and satisfiable with the assignment of $x = 333$:

225 $$\exists n \in [0, len(lc)) \; s.t. \; get(n, cl) = x \wedge i == 255 \wedge get(i, cl) = x \wedge get(0, cl) = 0 \ldots get(255, cl) = 333$$

226
227 The formula for the `largeinput` example when it is expected to return `true` is more interesting.
228 It may seem we need to enumerate and solve for all 256 values in the formula.

229 $$len(l) = 256 \wedge get(l, 255) = 333 \wedge \forall n \in [0, 255) \; get(n, l) \neq 333$$

230 However, the method for solving satisfiability assignments in Z3 [56] uses conditional model
231 logic and is able to produce the compact model for the contents of l as `(ite (= n 255) 333 0)`.
232 This example has neither a small iteration space nor a small input the logical witness input model
233 is still compact and efficiently computable!

234

235 2.2 Small Model Witness Generation

236 In practice many of the error states that we are interested in checking for satisfy the small scope
237 hypothesis. While the QBVF encoding approach can handle these cases, just like for the large
238 models, we can further optimize the performance for smaller models. Although QBVF is decidable
239 it is asymptotically and practically more expensive than solving over quantifier-free UF+IA+BV
240 formula. Consider the example below where the function checks to see if there is an element in the
241 list that satisfies the predicate `x * x == 0`.

```
function smallwitness(l: List_Int): Bool {
  l.someOf_{x*x==0}()
}
```

245

246 Using the QBVF encoding to generate a witness input for when this function returns true requires
247 us to solve the quantified formula:

$$\exists n \in [0, len(l))\ s.t.\ get(n, l) * get(n, l) = 0 \land \forall n \in [0, len(l))\ get(n, l) = havoc_{Int}(n)$$

250 To limit the introduction of quantifiers we use path splitting and implement a special *small-model*
251 *path* in the library implementation that is optimized for generating easily solved quantifier free
252 formula. An implementation of the `someOf` and `havoc` functions in the BosqueIR standard library
253 shows how this works:

```
function someOf_p(l: List_Int): Bool {
  let ct = len(l) in
  if(ct == 0) then false
  elif(ct == 1) then p(l.get(0))
  elif(ct == 2) then p(l.get(0)) || p(l.get(1))
  elif(ct == 3) then p(l.get(0)) || p(l.get(1)) || p(l.get(2))
  else quantified_someOf_p(l) //full quantifier path
}

function havoc_List_Int(c: Ctx): List_Int {
  let ct = havoc_{Int}(c) in
  if(ct == 0) then List_Int@{}
  elif(ct == 1) then List_Int@{havoc_Int(c ⊕ 0)}
  elif(ct == 2) then List_Int@{havoc_Int(c ⊕ 0), havoc_Int(c ⊕ 1)}
  elif(ct == 3) then List_Int@{havoc_Int(c ⊕ 0), havoc_Int(c ⊕ 1), havoc_Int(c ⊕ 2)}
  else quantified_havoc_List_Int(c, ct) //full quantifier path
}
```

267 These implementations include *small-model* paths which produce quantifier free formula for
268 lists of lengths less than 4. To construct a satisfying assignment to the input variables the SMT
269 solver can, when possible, find one in the propositional fragment of the formula. This design also
270 cleanly handles cases where there is a large constant or computed list, handled by the *quantifier*
271 path, while still allowing most reasoning to take place on the quantifier-free formula.

2.3 Small Refutation Construction

274 In the BSQChk checker our approach for 2b, from the confidence hierarchy, is to turn one of the
275 challenges of using fixed width bit-vectors – reasoning with large widths can be computationally
276 expensive – into a strength since reasoning with small widths is very efficient. Consider the example
277 below when we want to check if this function returns false.

```
function smallproof(x: Int): Bool {
  let ll = l.append(List_Int@{x}, List_Int@{1, 3}) in
  ll.contains(3)
}
```

282 From the code we can see that, regardless of the input value, the second list contains the constant
283 3 and so the return value is trivially true. In many cases, including our example, we can interpret
284 all the operations and show that that the false return value is infeasible (the formula is *unsat*)
285 entirely with a small width bitvector (anything 2 or larger). In practice *unsat* stability is quite
286 common and, unless an error is blocked by an overflow error, then like in our example the proof of
287 infeasibility almost always generalizes to the full bit width [25]. This technique combines with the
288 optimizations in Section 2.2 to, heuristically, further optimize witness generation when they exist
289 for small container sizes.

2.4 Heuristically Decidable Induction

292 By design the majority of the standard library container operations can be encoded in decidable
293 fragments of first order logic and in, BosqueIR programs, the use of explicit recursion is discouraged.
294

295 However, the BosqueIR language does support the use of arbitrary recursive computation and
296 there are times when it is needed. In these cases we still want to do best effort refutation proofs
297 and/or witness input generation. The *cover set method* for mechanizing induction [58] allows us to
298 implement a heuristic tactic that transforms inductive definitions into a closed form that can be
299 encoded in our decidable fragment.

300 Suppose a programmer implements Peano numbers using the standard inductive successor axioms
301 and, the recursive add and add3, functions defined below. They might also include an assertion that
302 the implementation is associative, *i.e.* $add(x, add(y, z)) = add((x, y), z)$. In the BosqueIR language
303 this is done by computing the two values and asserting the equality explicitly.

```
304 concept Peano {}
305 entity Zero provides Peano {}
306 entity Succ provides Peano { field val: Peano; }
307
    function add(x: Peano, y: Peano): Peano {
308   switch(y) {
309     case Zero => x
310     case Succ@{z} => Succ@{add(x, z)}
      }
311 }
312
313 function add3(a: Peano, b: Peano, c: Peano): Peano {
314   let rl = add(a, add(b, c))
      let rr = add(add(a, b), c)
315   if(rl != rr) then error
316   else rr
    }
317
```

318 Proving the infeasibility of the error statement requires discharging an inductive proof over the
319 recursive structure of add. In general this problem is undecidable but, in many cases, we can use the
320 construction of a cover set and a list of clauses that capture the inductive proof obligations. These
321 obligations can then be encoded as guards for the desired property and, if successfully discharged,
322 complete the inductive proof.

323 For this example the cover set generation heuristic produces a set of constraints that match the
324 principle of mathematical induction: the basis step is to instantiate the second argument of *add* to
325 be Zero, and the induction step is to instantiate the second argument to be $Succ@\{w\}$ with the
326 induction hypothesis generated by instantiating the second argument to be w.

327 Thus in our example the basis step is generated by instantiating c to be Zero producing a subgoal
328 (without any induction hypothesis since there are no recursive calls to add in the first equation):
329

$$\dagger \equiv add(a, add(b, Zero)) = add(add(a, b), Zero)$$

330

331 The induction step is generated by instantiating c to be $Succ@\{c\}$ with the induction hypothesis
332 obtained by instantiating c to be c:
333

$$\dagger\dagger \equiv add(a, add(b, c)) = add(add(a, b), c) \implies add(a, add(b, Succ@\{c\})) = add(add(a, b), Succ@\{c\})$$

334

335 Using the equality rules implied by the definition of add, the validity of both of these subgoal
336 can also be discharged.

337 This technique is powerful enough to support our needs wrt. container functors not convertable
338 to QBVF and also generalizes to handle a wide range of inductive proofs arising from common
339 idiomatic uses of recursion. In addition to the generality of the cover set method it also has the
340 desirable property that it does not require a costly iterative unrolling-and-check algorithm and
341 instead only performs a single guarded clause expansion – *i.e.* in our encoding we simply introduce
342 the guarded formula $\dagger \wedge \dagger\dagger \implies rl = rr$ as the closed form guarded constraint.
343

344
2.5 Focused Semi-Decision Search

Despite the wide range of techniques we use to encode a program in decidable fragments of first order logic, there are inherently times when the heuristics will fail. In these cases we accept that no proof of infeasibility can be constructed and instead fall back to the 2b case in out hierarchy of confidence boosting results.

In this mode we use brute force iterative unrolling of recursive functions and the library calls that cannot be encoded in QBVF. This is a simple semi-decision procedure for generating a witness input for a given error state but has limited effectiveness when the number of candidate calls to unroll is large. However, by the BosqueIR language design and encodings in the BSQChk checker we eliminate most of these candidates in practice. This results is a drastic reduction in the search space and increases the overall success rate of the procedure.

3 BOSQUEIR CORE AND ENCODING

This section provides the syntax and operators of the BosqueIR language with an emphasis on features that are not widely seen in existing programming IRs and/or that are particularly important for encoding to first order logic that is efficiently solvable by modern SMT solvers. Our encoding is a *deep embedding* that tries to push as much knowledge about the program structure as possible explicitly into the logical encoding.

3.1 Primitives

The BosqueIR language provides the standard assortment of primitive types and values including a special `none` value (`None` type), a `Bool`, `Nat` and `Int` numbers, safe `BigNat` and `BigInt` numbers, along with `Float`, `Decimal`, and `Rational` numbers. The BosqueIR `String` type represents immutable unicode string values. The language also includes support for commonly used types/values in modern cloud applications like, ISO times, SHA hashcodes, UUIDs, and other miscellany.

The `Nat` and `Int` types are mapped to fixed-width bitvectors. The choice of bitvectors allows us to support nonlinear operations while remaining in a decidable fragment of logic. The *BigInt* and *BigNat* types are more interesting as they are mapped to SMT *Int*. For non-linear operations we support both an over approximate encoding to uninterpreted functions + a simple axiomatization as well as a precise mode that uses the underlying NLA solver.

For solver performance we consider tradeoffs between precise encodings of semantics and versions that, in some cases, admits infeasible or excludes feasible execution traces. One of the places we allow this relaxation is in the encoding of `Float`, `Decimal`, and `Rational` numbers. We provide 3 flavors of encoding for these values in the BosqueIR system:

- Exact – where `Float` and `Decimal` are represented as a fixed width IEEE value and `Rational` is represented explicitly using the semantics of the underlying integer number representation. This can be computationally costly but is precise and will never generate an incorrect refutation or infeasible witness input.
- Conservative – where `Float`, `Decimal`, and `Rational` are defined as opaque sorts and all operations on them are uninterpreted functions with simple axiomatizations. This is efficiently decidable and a safe over-approximation so is sound for refutation but prone to generating infeasible witness inputs.
- Approximate – where `Float`, `Decimal`, and `Rational` are defined as *Real* sorts. This is the default encoding and, in theory, can admit both admit infeasible or exclude feasible execution traces. In practice the situations where these approximations negatively influence the outcome are limited and the overall increase in performance + success rate in refutation and witness input generation is a sweet spot for practical usage.

393
3.2 Self Describing Types

394 Structural *Tuple* and *Record* types provide standard forms of self describing types. One notable
395 aspect is the presence of optional indecies or properties. In most languages this would lead to
396 problems with analysis and require dynamic runtime support when accessing optional properties.
397 However, as we enforce a *closed world assumption* on BosqueIR programs we can compute the
398 actual set of concrete tuple/record types that may appear in the program.
399 Using the theory of constructors is a natural way to encode the tuple and record types. Since
400 the BosqueIR language enforces a closed world compilation model we can generate all possible
401 concrete `Tuple` and `Record` values that the program operates on. Consider a program that uses
402 the tuple `[Int, Bool]` and the record `{f: Int, g: String}`. The SMT constructor encoding for
403 these types would be:
404
405
```
type Tuple_[Int,Bool] = cons of BV * Bool
type Record_{f:Int,g:String} = cons of BV * String
```

406 This representation results in substantial simplifications when reasoning about operations on
407 these values as lookups are simply constructor argument resolutions instead of requiring the
408 analysis of various dynamic properties –*e.g.* looking up a property value based on a key in a
409 dictionary structure.
410 The BosqueIR language also supports self describing *union* types – *e.g.* `Int | None` or `Int |`
411 `String | [Bool, Bool]`. These unions are not encoded using direct constructors. Instead we use
412 an abstract value encoding, Section 3.5, for these unions and also for tuples/records with optional
413 entries (which are logically equivalent to union types).
414
415
3.3 Nominal Types

416 The BosqueIR language supports a nominal type system that allows the construction of object-
417 oriented style type inheritance structures. Abstract `Concepts` provide a means for inheritance and
418 multi-inheritance via *Conjunction*. The nominal type system differs from many oo-type systems in
419 that the *Concepts* are always abstract and can never be concretely instantiated while *Entity* (class)
420 types are always concrete and can never be subclassed.
421 This design simplifies the representation encoding, and as with the tuples and records, we can
422 enumerate all possible concrete object types and encode them using the theory of constructors.
423 Consider a program that uses the entity `entity Foo { field a: Float; field b: [Int,`
424 `Bool]}`. The SMT constructor encoding for this types would be:
425
426
```
type Foo = cons of Real * Tuple_[Int,Bool]
```

427
428
3.4 Key Types and Equality

429 Equality is a multifaceted concept in programming [43] and ensuring consistent behavior across the
430 many areas it surfaces in a modern programming language such as `==`, `.equals`, `Set.has`, and `List.sort`, is
431 source of subtle bugs [22]. This complexity further manifests when equality can involve referential
432 identity which introduces issue of needing to model aliasing relations on values, in addition to
433 their structural data, in order to understand the equality relation. The fact that *reference equality* is
434 chosen as a default, or is an option, is also a bit of an anachronism as reference equality heavily
435 ties the execution semantics of the language to a hardware model in which objects are associated
436 with a memory location.
437 To avoid these behavioral complications, and the need to model aliasing, the BosqueIR language is
438 *referentially transparent*. The only values which may be compared for equality are special primitive
439 values including *none*, booleans, primitive integral numbers, strings, and then tuples/records made
440 from other equality comparable values. In our encoding we want to ensure that this equality
441

relation is equivalent to *term equality* in the SMT solver. In their base representations these types all map to different SMT kinds so cannot be compared directly. Thus, we introduce a uniform representation for *Key Types* that boxes all of the values into a uniform SMT kind and tags them with the underlying type. As an example consider the SMT encoding for a program that uses the types `None`, `Bool`, `Int`, `String`, and the tuple/record definitions from above:

```
type KeyValueRepr =
  None@box
  | Bool@box of Bool
  | Int@box of BV
  | String@box of (Seq BV8)
  | Tuple_[Int,Bool]@box of Tuple_[Int,Bool]
  | Record_{f:Int,g:String}@box of Record_{f:Int,g:String}

type KeyValue =
  cons of TypeTag * KeyValueRepr
```

With this representation we can equality compare any `KeyType` values with the SMT term = semantics making the equality operations trivially decidable in UF! In conjunction with the immutability of the values (Section 3.6) this ensures that BosqueIR code is referentially transparent and functions do not need to use *frame rules* [34].

3.5 Abstract Types

The BosqueIR language allows for type generalization in a number of contexts – optional fields in records/tuples create types that may contain many different concrete records/tuples, abstract `Concept` types may be implemented by many concrete `Entity` types, and the language also allows unrestricted union types. One approach would be to use the closed world assumption and create a datatype for each possible abstract type in a program. In addition to the large number of types created, this can also result in large amounts of data reshaping when assigning between variables of different union types. Consider assigning an `Int | Bool` typed variable to an `Int | Bool | None` typed variable or to an `Any` typed variable. In these cases each assignment would involve extracting the concrete value from the source value and injecting it into the representation for the target value. This results in type checks and switches on each assignment and a quadratic (worst case) number of cases to handle assignments between all possible options.

Instead we use only two representations for all generalized types that appear in the program. One representation, *KeyValue*, as described above is for any key valued abstract type. Another representation, *Value*, is for all other abstract values and includes a representation for key types that are combined with non-key types. For each concrete type in the program we define a unary constructor that maps the concrete type into a `KeyValue` (above) or `Value` representation:

```
type ValueRepr =
  Foo@box of TypeTag * Foo

type Value =
  KeyValue@cons of TypeTag * KeyValue
  | Value@cons of TypeTag * ValueRepr
```

This encoding eliminates the need extensive for reshaping when assigning between most pairs of abstract types, even in the special case of converting between *KeyValue*/*Value* representations is a single op, and does not require any type specific logic on the concrete type. The extraction/injection operations from these representations to concrete values are also simple single ops that, in most cases, can be done without any additional type tag checking.

Primitive	:=	$none \mid true \mid false \mid i \mid f \mid s \mid v \mid \ldots$
Operator	:=	$(\neg\mid-)e \mid e_1(+\mid-\mid*\mid/)e_2 \mid e_1(\wedge\mid\vee)e_2 \mid e_1(<\mid=)e_2$
Cons	:=	$[Exp_1, \ldots, Exp_j] \mid \{f_1 = Exp_1, \ldots, f_j = Exp_j\} \mid EntityName@(Exp_1, \ldots, Exp_j)$
TupleOp	:=	$Exp.n \mid Exp.[n_1, \ldots, n_k] \mid Exp.[n_1 = Exp_1, \ldots, n_k = Exp_k] \mid Exp \oplus Exp$
RecordOp	:=	$Exp.p \mid Exp.\{p_1, \ldots, p_k\} \mid Exp.\{p_1 = Exp_1, \ldots, p_k = Exp_k\} \mid Exp \uplus Exp$
EntityOp	:=	$Exp.f \mid Exp.\{f_1, \ldots, f_k\} \mid Exp.\{f_1 = Exp_1, \ldots, f_k = Exp_k\} \mid Exp \leftarrow Exp$
Invoke	:=	$\texttt{fn}(Exp_1, \ldots, Exp_j)$
TypeTest	:=	Exp `istype` $Type$
Assert	:=	$error_{trgt} \mid error_{other}$
Cond	:=	`if` Exp_c `then` Exp_t (`elif` Exp_c `then` Exp_t) $*$ `else` Exp_f
Switch	:=	`switch`(Exp)(`case` $Primitive$ `=>` Exp | `case` $pattern$ `=>` Exp) $*$
Let	:=	`let` ($v = Exp$) + `in` Exp
Exp	:=	$Primitive \mid Operator \mid Cons \mid {}^{*}Op \mid Invoke \mid TypeTest \mid Assert \mid Cond \mid Switch \mid Let$

Fig. 3. BosqueIR Expressions

3.6 Operations

The expression language for BosqueIR is shown in Figure 3. By construction the language maps directly to the SMTLIB expression language in many cases.

Primitive expressions include special constants like `true`, `false`, `none`, literal numeric values like `i` or `f`, literal strings `s`, and variables (either local, global, or argument). The BosqueIR language has the standard assortment of numeric and logical *Operators* which, thanks to our type encodings, map mostly to the semantics of the operations in UF+IA+BV. The only exceptions are the integral operations which we provide bounds checking on [40] – which are errors when over/under flows occur for `Nat`/`Int` values.

The `constructor` operations in the language are all simple and explicit operations with a type name + full list of values. These all map naturally to the SMT theory of constructors.

There are similar sets of operations for tuples, records, and objects. There is the standard index ($e.n$ where n is a constant), or named property/field accessor ($e.f$). The language also includes bulk data operations for projecting out (or updating the values in) a set of indecies and creating a new tuple or a set of properties/fields creating a new record. The bulk update version handles the dynamic dispatch (and invariant checking) if the type of e is not unique as well. Finally, there is a tuple append operator ($\oplus$) which appends two tuples which do not have any optional fields, a record disjoint join operator ($\uplus$) which creates a new record from the (disjoint) set of properties in the two argument records, and a nominal type field merge using the properties/values from the right-hand side expression.

Due to the construction of the BosqueIR semantics, with referential transparency and no mutation, the encoding of BosqueIR functions closely matches the call to the corresponding SMT implemented function. The only modification is to encode the possibility of an error result in addition to the declared result type, `T`, of the BosqueIR function. We do this in the expected way by making the return type of every SMT function a $Result_T$ and, at the call site, either propagating the error or continuing on the computation.

For type testing we again leverage the closed world design of the BosqueIR language and enumerate the non-trivial subtype relations and encode them as binary functions. We note that we don't need to encode many *uninteresting* subtypes that are statically known to be infeasible, *e.g.* tuples are never subtypes of record types, or trivially true, *e.g.* only `none` is not a subtype of `Some`.

540
541
542
543
544
545
546
547
548

ListStructure	:=	$\texttt{empty} \mid \texttt{List}_T\texttt{@}\{e_1, \ldots, e_j\} \mid \texttt{slice}(l, a, b) \mid \texttt{concat2}(l_1, l_2) \ldots$
ListProperty	:=	$\texttt{fill}(n, v) \mid \texttt{range}(l, h) \mid \texttt{havoc} \mid \ldots$
ListCompute	:=	$\texttt{map}_{\texttt{fn}}(l) \mid \texttt{filter}_{\texttt{p}}(l) \mid \ldots$
ListAccess	:=	$\texttt{size}(l) \mid \texttt{get}(l, n) \mid \ldots$
ListPred	:=	$\texttt{has}_{\texttt{p}}(l) \mid \texttt{find}_{\texttt{p}}(l) \mid \texttt{findLast}_{\texttt{p}}(l) \mid \texttt{count}_{\texttt{p}}(l) \mid \ldots$
ListIterate	:=	$\texttt{reduce}_{\texttt{fn}}(l) \mid \texttt{reduceOrderd}_{\texttt{fn}}(l) \mid \ldots$

Fig. 4. BosqueIR List<T> Operations

549
550 There are two asserts in the language. Both of them produce an $\texttt{error}_T$ return value for the
551 function. The distinction in these *Assert* operations allows us to separate all possible errors into two
552 groups – the $\texttt{error}_{trgt}$ value for the specific error we are interested in and the $\texttt{error}_{other}$ value for
553 any other error that occurs.
554 The remaining *Cond*/*Switch* control flow and *Let* binding operations are self explanatory and
555 map simply to the SMTLIB *ite* and *let* constructs.
556 Finally, we note that all of these expressions are deterministic and that none of them mutate any
557 state. Interestingly none of these design choices individually seem surprising on their own and
558 individually every one of them can be seen in existing programming languages. However, when
559 taken in their totality they result in a remarkably simple encoding into first order logic in a way
560 that is highly amenable to solution with modern SMT solving techniques!
561
562

4 BOSQUEIR CONTAINERS AND ENCODING

563 Containers and operations on them play a major role in most programs but, as a design principle,
564 BosqueIR programs do not allow loops and the language is designed to discouraged the extensive
565 use of explicit recursion. Instead the BosqueIR language includes a rich set of container datatypes
566 and functor based operations on them. In practice these operations, and parameterizeable functors,
567 are sufficient to cover the majority of iterative operations [1].
568 The semantics of these containers and operations inherently involve reasoning over the (symbolic)
569 range of the containers contents. As a result many features of these libraries cannot be handled
570 using the quantifier free encoding techniques described in Section 3. Instead we introduce a novel
571 encoding for the container library code that is entirely in the decidable fragments of logic we
572 previously used along with the QBVF [56] fragment of first-order logic.
573
574

4.1 List<T> Type and Operations

575 Figure 4 shows an example of the operations provided for the List type[2]. The List type provides
576 a random access model for an immutable Nat indexed set of values. In practice this is often
577 implemented as a contiguous memory array or using RRB-vectors [47]. However, as we see in this
578 section, there are alternative representations that are more useful for formal reasoning.
579 The semantics of BosqueIR collections use distinct types/implementations for each instantiation
580 of a List type and each functor use is statically *defunctionalized*. This ensures that two lists of
581 different content types have distinct representations and every function call is first order.
582 The List type has a number of basic constructors including the empty constant and a literal
583 constructor $\texttt{List}_T\texttt{@}\{e_1, \ldots, e_j\}$. In addition there are the expected set of concat, slice, fill, size,
584 get, *etc.*. operations which have the usual semantics.
585

586 [2]The BosqueIR language provides a rich set of collections including Set and Map types but, as the principles we use for
587 List types extend in the expected manner, we focus on List<T> in this section.
588

589 We also provide a special *havoc* constructor. This constructor takes an opaque `context token` and
590 returns a list with a symbolic length and contents that when accessed are havoc values themselves.
591 This constructor (and similar constructors for other types) are the only form of non-determinism
592 allowed in BosqueIR programs. The following code illustrates the use of a havoc:
593
```
594 function main(): Bool {
595   let l = List_Int::havoc(0) in
596   if (empty(l)) then false
597   else get(l, 0) == 0
    }
```
598
599 In this sample the `havoc` constructor can create a length `0` list and take the first branch (to return
600 false) or a list with at least one element. The `get` operation causes a call to the havoc constructor
601 for `Int` which can return `0`, and the function will be `true`, or an arbitrary nonzero number to make
602 the result `false`.
603 To motivate this section we use the following running example that illustrates the challenges
604 and subtleties of efficiently reasoning over the BosqueIR collection libraries:
605
```
606 function main(c: Int, args: List_Int): Bool {
607   let la = List_Int@{1, 2 , 3} in
608   let lb = List_Int::fill(c, 0) in
      let lc = List_Int::concat(la, args, lb) in

609   let ld = lc.map_|x|() in
610   let le =  ld.filter_<10() in

611   le.someOf_=0()
612 }
```
613
614 In this code sample several lists are constructed using both constant and symbolic values, via the
615 function arguments `c` and `args`, processed with the functors `map` and `filter`, finally we perform a
616 logical computation on the resulting list. The first line constructs a constant list with 3 elements
617 while the second line constructs a list of `c` elements all of which are set to `0`. The next line builds
618 a list, `lc`, that is the concatenation of these two lists with the symbolic input list `args`. Next we
619 construct the list `ld` by applying the function `abs(x)` to every element in the list `lc`. The call to
620 `filter` then creates a sublist of elements from `ld` that satisfy the predicate `x < 10`. In the test
621 operations we do a test to see if there exists any element in the `le` list, via the `someOf` functor, that
622 satisfies the predicate `x = 0`.
623 The emphasis on including a rich set of functor operations as part of the code BosqueIR langugae
624 makes it possible to write large portions of an application without resorting to explicit iterative or
625 recursive code and then dealing with the challenges of loop or inductive invariant generation. Thus,
626 we can optimize the BSQChk reasoning for handling operations on this known set of functors and
627 use a more general technique as a fallback for explicitly recursive code (Section 5). However, we
628 still face the challenge that operations inherently involve reasoning over the (symbolic) ranges of
629 the container contents.
630 To address the challenge of quantified semantics in the container and operator specifications
631 we note that the operations of interest can actually be divided into three categories. The first is
632 operations that can be handled via encoding with quantifier free algebraic data-types as shown
633 in Section 4.2. The next is a set of functors that, fundamentally, require quantified logic in their
634 ground terms as described in Section 4.3. Finally, we introduce a special lemma for handling sublist
635 selection based operations in Section 4.4. The combination of these techniques gives us an optimized
636 encoding for, the majority of, our container library code in decidable fragments of logic.
637

638
4.2 Theory of Constructors

639 The direct way to model the core container types in the BosqueIR language would be to define
640 a SMT constructor using a size component and a contents array. While conceptually simple, this
641 approach has the problems of bringing another theory, arrays, into the logic and also results in
642 many operations, such as append, requiring quantification over the result array contents. Instead
643 we use the fact that BosqueIR equality is limited to primitive values – *i.e.* containers cannot be
644 compared for equality and thus we do not need to preserve structural equality in the encoding.
645 With this observation we can take the view that, instead of one list constructor kind $List =$
646 $BV * Array$, there are actually many constructors and we can encode the `List` type[3] using the
647 following algebraic structure:
648

```
649 type List =
650   empty
651   | const3     of BV * BV * BV
652   | fill       of BV * BV
653   | concat2    of List * List
654   | concat3    of List * List * List
655   | map_|x|    of List
      | filter_x<10 of List
      | havoc      of σ
```

656 This algebraic structure is generated per-program based on the uses of each container type. With
657 this encoding many collection operations reduce to simple quantifier free formula. Additional
658 simplification logic in the operator definitions minimizes the nesting depth of the constructor trees
659 as illustrated in the SMT implementations for the `concat2` and $\texttt{map}_{|x|}$ functors:
660

```
661 fun op_concat2 ((l1 List) (l2 List)) List =
662   if (l1 = empty ∧ l2 = empty)
663     empty
664   elif (l1 = empty)
665     l2
666   elif (l2 = empty)
667     l1
    else
      concat2(l1, l2)

668 recfun op_map_|x| ((l List)) List =
669   if (l = empty)
670     empty
671   elif (l = const3(a, b, c))
672     const3(|a|, |b|, |c|)
673   elif (l = fill(c, v))
674     fill(c, |v|)
675   elif (l = concat2(ll, lr))
676     op_concat2(op_map_|x|(ll), op_map_|x|(lr))
677   ...
    else
      map_|x|(l)
```

678 These examples show how the solver can perform substantial algebraic simplification to, in
679 many cases, entirely eliminate the need to expand and reason about the more complex constructor
680 operations. In the case of $\texttt{op_map}_{|x|}$ when the argument list size is zero we trivially drop the function
681 application and just return the `empty` list, More interestingly are the cases of `fill` where we can
682 expand and then reconstruct the fill with the map function applied to the single fill argument and
683 `concat2` where, similarly, we can expand the argument and apply the operation down to the two
684

685 [3]For brevity we assume lists are of type `Int` when not otherwise specified.
686

sublists. In our motivating example these simplifications, and similar simplifications in `filter`, allow us to finitize a large part of the `List` processing and show that `ld` is equivalent to:

```
concat3(
  const(1, 2, 3),
  map_|x|(havoc(σ_1)),
  fill(c, 0)
)
```

Similarly, accessor operations can be reduced to use algebraic traversals of the constructor trees in many cases. In our example the `someOf` operation would be implemented as follows:

```
recfun someof_=0 ((l List)) Bool =
  if (l = empty)
    false
  elif (l = const3(a, b, c))
    a = 0 ∨ b = 0 ∨ c = 0
  elif (l = fill(c, v))
    v = 0 ∧ c ≠ 0
  elif (l = concat2(ll, lr))
    someof_=0(ll) ∨ someof_=0(lr)
  elif (l = map_|x|(ll))
    ∃ n, n < ll.size() ∧ |ll.get(n)| =0
  ...
   else
    //havoc constructor
    ∃ n, n < l.size() ∧ l.get(n) =0
```

As shown in this example there are many cases where we can finitize the access formula but in some cases, like the `map`$_{|x|}$`(ll)` or the `havoc` branches, we cannot fully eliminate the use of quantified expressions.

4.3 Simply Quantified Formula

Common approaches to handling quantified formula in SMT solvers often involve the use of instantiation triggers or other heuristics. Unfortunately, these approaches suffer from poor performance and solver instability [33]. Further, they fundamentally introduce incompleteness into the engine and prevent the construction of models. As model generation is one of our key objectives we want to have a decision procedure for these formula.

As described in Section 3 the BosqueIR language specifies the `Nat` type as a fixed width integer and is represented using the theory of fixed-width bitvectors. With this choice of integer encoding, and our fixed set of quantification forms, we can ensure that all quantified variables are scoped to just bitvector values which puts the formula in the theory of quantified bitvector formula (QBVF). If we look at the *map* formula from the `someof` implementation we see it has the following structure:

$$\exists n, n < ll.size() \wedge |ll.get(n)| = 0$$

As the n in this formula is the only quantified variable and it is a bitvector, it is clear this formula is in the QBVF fragment. As such it is both decidable and there exist efficient techniques exist for solving these problems in practice [42, 56].

4.4 ISequence Lemma

The combination of algebraic datatypes and simple quantifiers is sufficient to encode many container operations in the BosqueIR standard library. However, subset based computations like `filter` are still problematic. These computations have four properties that define their semantics. For a `List`$_{Int}$ `l`, a predicate `p`, and the output list `lp`:

(1) $x \in l \wedge p(x) \Rightarrow x \in lp$

736 (2) $x \in lp \Rightarrow p(x) \land x \in l$
737 (3) The multiplicity of $x \in lp$ and $x \in l$ are equal
738 (4) The order of elements in lp matches l

739 The first 2 conditions can be easily specified using simple quantifiers but the multiplicity and
740 ordering properties are more complex. Instead we present an auxiliary datastructure, an `ISequence`
741 which is a integer indexed sequence of bitvector values and an uninterpreted function $\texttt{iseq}_p\texttt{(List}$
742 $\texttt{l)} = s$, with the following constraints:

743
744 (1) $\forall n \in [0, size(l)), p(\texttt{get}(l, n)) \Rightarrow \exists i \in [0, \texttt{size}(s))$ s.t. $s[i] = n$
745 (2) $\forall i \in [0, \texttt{size}(s)), p(\texttt{get}(l, s[i]))$
746 (3) $\forall i \in [0, \texttt{size}(s)), s[i] \in [0, \texttt{size}(l))$
747 (4) $\forall i, j \in [0, \texttt{size}(s)), i < j \Rightarrow s[i] < s[j]$

748 Assertions 1, 2 enforce that any indecies in the `List l` satisfying the predicate must appear in
749 the `ISequence s` and that every element in `s` contains an index in `l` that satisfies the predicate
750 – matching the subset requirements. Assertions 3, 4 enforce the size, multiplicity, and ordering
751 constraints by limiting the positions and range of values that can appear in the result `ISequence`.
752 These constraints all fall into the QBVF fragment and are dischargable by a SMT solver.
753 With this auxiliary operation filter is defined simply as a constructor of `List * ISequence`.
754 We can also use the `iseq` function to compute the result of the `countIf` functor and, as the
755 underlying function uses are identical to constructing the `ISequence` for filter it is trivial to show
756 that `l.filter(p).size() == l.countIf(p)`. We use a similar form of auxiliary operations to
757 implement sorting, uniqueness, and join constructs.
758 Returning to our running example, the definition of `le` would be equivalent to the formula:

759
```
760 concat3(
761   const(1, 2, 3),
762   filter<10(map|x|(havoc(σ1)), iseq<10(map|x|(havoc(σ1)))),
      fill(c, 0)
    )
```
763
764 Now suppose we want to try and prove that our running example function always returns true –
765 that is $\texttt{le.someOf}_{=0}\texttt{()}$ is `false` is unsatisfiable. If we try to check this formula we will find that the
766 proof fails, but since the formula is decidable, we can instead ask the solver to produce a model that
767 satisfies the formula when the function returns false – that is $\texttt{le.someOf}_{=0}\texttt{()}$ is `false` is satisfiable.
768 Applying the algebraic rules for `someof` simplifies this to the checks that each of the three sublists
769 is false. The first sublist is always false as it is constant and does not contain 0. The `args` sublist can
770 easily be made false for `someOf` check by setting it to be the `empty` list. Finally, we can make the
771 `fill` list false for the predicate by setting the input parameter `c` to 0 as well. If we update the code
772 to, say assign $\texttt{lb = List}_{Int}\texttt{::fill(c + 1, 0)}$, then the proposition that the method returning
773 false is unsatisfiable will reduce to the $\lor$ of the results on the three sublists and now the fill option
774 will always be true and the proof will succeed.
775

776 ## 5 COVER SET METHOD FOR INDUCTIVE REASONING

777 In the previous section we built a specialized encoding for the most common container operations
778 in BosqueIR programs. However, there are some functors that we cannot handle fully even with
779 quantified templates. Additionally, we want to imbue the BSQChk system with a powerful and
780 robust methodology for proving the safety of programs which use general recursion. Remarkably
781 there exists such a method, the *cover set method* (Section 5.1) for mechanizing induction [58]. This
782 technique is powerful enough to support our needs wrt. the remaining reduce style container
783 functors and also generalizes to handle a wide range of inductive proofs. In addition to the generality
784

of the cover set method it also has the desirable property that it does not require an costly iterative unrolling-and-check algorithm [45, 53] and instead only performs a single guarded clause expansion.

5.1 Cover Set Background

The cover set method for mechanizing induction was introduced in 1988 [58] in an equtaional programming language framework in the paradigm of term rewriting approach for automating inductive reasoning. Function definitions in that framework are given in ML style in a recursive fashion, for different cases for constructing a data type on which the function is defined. As a simple illustration, the data type Natural is defined by a constant 0 and a successor function s with the implicit property that $0 \neq s(x)$ and s is free to imply that $s(x) = s(x) \implies x = y$. A binary function + is recursively defined as:

$$x + 0 = x$$
$$x + s(y) = s(x + y)$$

The cover set method computes induction schema from recursively defined terminating function definitions. The key idea is to use the well-founded ordering used to establish a proof of termination of the function definition to (i) select variables(s) in a function call to perform induction and (ii) generation of induction hypotheses for instantiation of induction variables from recursive calls of the function in its body, which are guaranteed to be lower in the well-founded order. The variable(s) on which a function definition recurses is selected for generating induction scheme and hence performing induction.

The above definition of + is terminating over natural numbers. Since the definition is recursing on the second argument, the induction variable to be chosen is the second argument y of +. In the first case when $y = 0$, there is no recursive call to + in the right side; this corresponds to the basis step of an induction proof in which the second argument is instantiated to be 0. In the second case when the second argument is not equal to 0 and equal to $s(y)$ for some y, there is a recursive call to + with y as its second argument, so the induction hypothesis is generated by instantiating the second argument to be y and the conclusion goal is generated by instantiating the second argument to be $s(y)$. The reader would notice that this induction scheme is precisely the principle of mathematical induction on natural numbers.

The cover set method formalizes this approach to automating proofs by induction, addressing three important aspects in inductive proofs: (i) choice of well-founded ordering and (ii) variables to perform induction on, and (iii) the induction hypothesis (hypotheses) to be generated. It consists of a finite set of tuples corresponding to each case in the function definition; the first component in the tuple is the condition on the input under which the function computes the result, and the second component is a finite set of instantiations of the input for generating different induction hypothesis for each recursive call; if there is no recursive call, then no instantiation for any induction hypothesis is generated and is thus left as the empty set; if there are multiple recursive calls, then there are multiple instantiations.

5.2 Revisiting Add

To show the details of the coverset construction and proof we revisit the definition of add from Section 2.4 (defined using infix notation for simplicity).

```
function +(x: Peano, y: Peano): Peano {
  switch(y) {
    case Zero => x
    case Succ@{z} => Succ@{x + z}
  }
}
```

834 Assuming the definition is terminating and is complete, a cover set is generated from the function
835 definition as follows: For each case of the switch statement, there is a tuple consisting of a boolean
836 condition for which the case expression applies; this boolean condition is the conjunction of the
837 boolean condition for the case and the negation of the conjunction of the conditions of all the cases
838 before it. The second component of the tuple is a finite (possibly empty) set of substitutions serving
839 as instantiations for generating induction hypotheses. From add, the cover set includes two tuples:
840 $\{< y = 0, \{\} >, < y = s(w) and not(y = 0), \{w\} >\}$.
841 To prove of the associativity of:

842

$$x + (y + z) = (x + y) + z$$

843
844 Observe the variables y and z appear as second arguments in subterms with +; however, only z
845 appears as the second argument on both sides, so it is heuristically preferred.
846 The first tuple of the cover set for + generates the subgoal:

847

$$z = 0 \Rightarrow x + (y + z) = (x + y) + z,$$

848
849 since there is no induction hypothesis. The second tuple generates the subgoal:

850

$$((z = s(w) \wedge (z \neq 0)) \wedge (x + (y + w) = (x + y) + w)) \Rightarrow x + (y + z) = (x + y) + z.$$

851

852
5.3 Recursively defined Functors on Containers

853
854 When proving inductive facts we switch from the constructor based definitions in Section 4
855 and instead use inductive definitions for every container operation. This simplifies the inductive
856 reasoning that is required and can be trivially done as an equality assertion of the two definitions
857 at the same time we are introducing the other cover set conjectures.
858 We illustrate below some examples on containers using functors such as map and reduce with
859 inductive definitions. Consider a data type *list* generated using constructors *nil* and *cons*. Define a
860 binary function append on lists, a unary function rev and a unary function length with the usual
861 semantics:

```
862 function append(l1: IntList, l2: IntList ): IntList {
863   switch(l) {
864     case empty => l2,
865     case cons@{h1, tl1} => cons(h1, append(tl1, l2))
866   }
867 }
868
869 function length(l: IntList ): Nat {
870   switch(l) {
871     case empty => 0,
872     case cons@{h, tl} => length(tl) + 1
873   }
874 }
875
876 function rev(l: IntList ): IntList {
877   switch(l) {
878     case empty => l,
879     case cons@{h, tl} => append(rev(tl), cons(h, nil)),
880   }
881 }
```

877 The above definitions are terminating and the associated coverset for append is: $\{(l1 = empty, \{\}), (l1 \neq empty \wedge l1 = cons(h1, tl1)), \{tl1\})\}$.
878
879

```
880 function reduce(l: IntList, op: fn(_: Int, _: Int) -> Int, id: Int): Int {
881   switch(l) {
882     case empty => id
```

```
      case cons@{h, tl} => op(h, reduce(tl, op, id))
    }
  }
```

With *sum* recursively defined:

```
function sum(l: IntList): Int {
  switch(l) {
    case empty => 0
    case cons@{h, tl} => h + sum(tl)
  }
}
```

Using the coverset method we can show that $reduce(l, fn(x, y) => x + y, 0) = sum(l)$.

The cover set associated with the above terminating definition of *sum* is $\{(l = empty, \{\}), (l \neq empty \wedge l = cons(h, tl)), \{tl\})\}$. Using the cover set method, two goals are generated to prove the conjecture:

$$l = empty \Rightarrow reduce(l, +, 0) = sum(l)$$
$$l \neq empty \wedge l = cons(h, tl) \wedge reduce(tl, +, 0) = sum(tl) \Rightarrow reduce(l, +, 0) = sum(l).$$

The first subgoal, after substituting l by *empty*, is $reduce(empty, +, 0) = sum(empty)$, which simplifies by the definitions of *sum* and *reduce* to $0 = 0$ which is valid.

The induction hypothesis generated from the coverset is $l \neq empty \wedge l = cons(h, tl)) \wedge reduce(tl, +, 0) = sum(tl)$; the conclusion of the second subgoal, after substituting $cons(h, tl)$ for l, is $reduce(cons(h, tl), +, 0) = sum(cons(h, tl))$ which simplifies by the definitions of *reduce* and *sum* to $h+reduce(tl, +, 0) = h+sum(tl)$; with the use of the induction hypothesis $reduce(tl, +, 0) = sum(tl)$, the subgoal is proved.

5.4 Integration into Z3

A direct proof in Z3 can be obtained by computing instantiations of the universal variables in the definitions of sum and reduce above. The coverset can be used to find instantiations of the universal variables: in the first subgoal, terms (reduce l + 0) and (sum l) each simplify under the condition (= l empty) to 0 and 0, respectively. Similarly, in the second subgoal, (reduce tl + 0) and (sum tl) also simplify under the condition (and (not (= l empty)) (= l (cons h tl))) by instantiating the second cases of the definitions of reduce and sum to (+ h (reduce tl + 0)) and (+ (sum tl)) respectively, on which the induction hypothesis from the cover set applies. This is one major advantage of the cover set method that often appropriate instantiations can be generated from the cover set itself.

An indirect proof can be obtained by proving the unsatisfiability of the negated conjunction of the two subgoals generated using the coverset, with the definitions of sum and reduce.

5.5 Extensions

In a given conjecture, many function symbols may occur and, additionally, the same function symbol may occur with different sets of arguments. Each of these function symbols has an associated cover set from which an induction scheme can be generated. Then each of these induction schemes could be a candidate for attempting a proof by induction of the conjecture.

Given that SMT solvers are very efficient in handling very large formulas, it is feasible to simultaneously generate alternative subgoals generated from different induction schemes with the objective of trying them all with the hope that at least one would succeed.

5.6 Model Generation

The cover set method can also be used to generate a counter-example (model) from a false conjecture. In a proof attempt, one of the subgoals is likely not to succeed. If its variables are instantiated in a systematic way, a counter-example can be generated.

Consider a false conjecture, that `length(filter(l, p)) == length(l)`, about a functor filter defined as:

```
function filter(l: IntList, p: pred(_: Int) -> Bool): IntList {
  switch(l) {
    case empty => empty
    case cons@{h,tl} => if(p(h)) then cons(h, filter(tl, p)) else filter(tl , p)
  }
}
```

Choosing the coverset defined by filter, there are three subgoals generated. The first is $l = empty \Rightarrow length(filter(l, p)) = length(l)$ which can be easily proved by substituting empty for l and then using the definitions of length and filter.

The second (failing) subgoal is: $(l \neq empty \wedge l = cons(h, tl) \wedge \neg(ph) \Rightarrow length(filter(tl, p)) = length(tl)) \Rightarrow length(filter(l, p)) = length(l)$.

Substituting for cons(h, tl) for l in the conclusion of the subgoal and simplifying using the definitions of filter and length gives: $length(filter(tl, p)) = length(tl) + 1$ which after using the induction hypothesis gives $length(tl) = length(tl) + 1$ which is a contradiction.

A counter-example is generated in which l = cons(h, tl), tl = empty and h is such that p(h) = false.

6 RELATED WORK

The approach to programming language design and checking presented in this paper represents a novel way to view and build on many longstanding research areas. From the language design standpoint we started from a blank slate and, for every design choice, asked how each feature impacted the analysis problem. From the tooling perspective we looked to the strengths and limitations of existing symbolic analysis, testing, and verification methodologies, and then, looked at how the new language semantics would allow us to build on these strengths and eliminate various weaknesses.

6.1 Language Design:

The goal of the BosqueIR language design was to enable the construction of powerful and practical analysis and developer tools. The approach taken was to identify language features that introduce difficult to reason about constructs and eliminate them.

Loops and Recursion: Loops are a foundational control flow construct in the *Structured Programming* paradigm [10, 15, 21]. However, precisely reasoning about them requires the construction of *loop invariants*. Despite substantial work on the topic [18, 32, 50, 51] the problem of automatically generating precise loop invariants remains an open problem. Instead, inspired by the empirical results in [1] which show most loops are actually just encoding a small set of common idioms, we entirely exclude loops from the BosqueIR language in favor of a comprehensive set of functors. Explicit recursion is still allowed, although discouraged, as a fall-back for algorithms where it is fundamentally needed or cannot be expressed with the standard set of functors. Thus, we still need to deal with the problem of verification conditions for the functors and inductive proofs for recursive calls but, as shown in Section 4 and Section 5 with clever encoding strategies, these problems are tractable.

981 *Identity and Equality:* Equality is a complicated concept in programming [43]. Despite this
982 complexity it has been under-explored in the research literature and is often defined based on
983 historical precedent and convenience. This can result in multiple flavors of equality living in a
984 language that may (or may not) vary in behavior and results in a range of subtle bugs [22] that
985 surface in surprising ways.
986 The notion of identity based on memory allocation values also introduces the need to track
987 this property explicitly in the language semantics – introducing the need for alias analysis and
988 a strong distinction between pass-by-value and pass-by-ref semantics. This is another problem
989 that has been studied extensively over the years [19, 20, 28, 36, 37, 52] and remains an open and
990 challenging problem. As with loops, our approach is to eliminate these challenges and complexities
991 by eliminating the use of referential identity and restricting equality to a fixed set of data types
992 where equality is simply term equality.
993
994 *Mutability:* Mutability is known to be a challenge when reasoning about code. It introduces a
995 host of problems including the need to perform *strong updates* (*i.e.* retracting previously asserted
996 facts) and computing *frames* [48]. There are a number of useful techniques, including various
997 *ownership type* systems [7, 8], or *linear type* systems [17, 54, 55], for managing or isolating these
998 issues which may be valuable to incorporate into the BosqueIR language in the future.
999
1000 *Determinism:* Indeterminate behaviors, including undefined, under specified, or non-deterministic
1001 behavior, require a programmer or analysis tool to reason about and account for all possible out-
1002 comes. While truly undefined behavior, *e.g.* uninitialized variables, has disappeared from most
1003 languages, there is a large class of underspecified behavior, e.g. sort stability, map/dictionary enu-
1004 meration order, *etc.*, that remains. The inclusion of non-deterministic operations also results in
1005 code that cannot be reliably tested (flakey tests), where failing witness inputs may sometimes not
1006 fail, and as each non-deterministic choice point introduces a case split for reasoning, can greatly
1007 increase the cost of analyzing a codebase. These increase the complexity of the development process
1008 and, as time goes on, are slowly being seen as liabilities that should be removed [6] so for the
1009 BosqueIR we have taken this to the logical conclusion and fully specify the behavioral semantics
1010 of every language operation.
1011
1012
6.2 Program Analysis

1013 Given the rich literature on program analysis our focus in this section is how the language features,
1014 logic encodings, and features of the BSQChk compare with other approaches.
1015
1016 *Symbolic Verification:* The BSQChk checker is explicitly designed with *full verification* of correct-
1017 ness as a non-goal. The *confidence boosting* hierarchy in Section 1 specifically includes outcomes that
1018 are not verified and the workflow in Figure 2 does not have a path for manually adding lemmas or
1019 other manual proof steps. The use of powerful proof systems with extractable code [12], languages
1020 with proof assistants like [31], or dependently typed languages [14, 46] show that it is practical to
1021 produce fully verified software in some domains today. However, using these languages/techniques
1022 require substantial manual effort and expertise including knowledge of the underlying theorem
1023 provers behavior, and the ability to formally model the desired behavior of the software. This places
1024 the use of these systems beyond the what is practical or cost effective for most applications. Work
1025 on Liquid Types [49] and the Ivy language [35] represent interesting approaches to verification
1026 by enforcing that the logic used in the types/code remains in decidable fragments of logic. This
1027 reduces the expertise and manual work required but does not eliminate it entirely and restricts the
1028 languages to problems which can be expressed entirely in the supported logical fragments.
1029

In contrast the BSQChk checker does not require a separate proof language or the manual insertion of lemmas, and supports arbitrary code in the BosqueIR language. However, an interesting question is how the proofs produced by full verification techniques can be ingested into the BSQChk logical representation to enable fully-verified foreign function integration (FFI).

Cover Set Method and Term Rewriting: The cover set method [58] is motivated by the success in the use of the induction scheme supported in ACL2 [26] based on proving termination of lisp functions in which induction hypotheses are computed from recursive calls in their definition. The cover set method generalizes that idea from lists to arbitrary data types generated by a finite set of constructors in an equational programming language framework; further it also allows the use of sophisticated syntactic termination orderings. The cover set method as adapted to BosqueIR however functions more similarly to the induction scheme generation in ACL2 due to the nonequational structure of the programming language. The Imandra system [45] is a novel design that combines SMT reasoning with ACL2 style proving to support industrial uses.

Abstract Interpretation: Abstract interpretation and related dataflow analyses [41] are at the other end of the spectrum from full symbolic verification. This framework for proving properties of programs is very different since it is based on approximating the behavior a concrete program by that of an abstract program working on an abstract domain with the requirement, any property of the abstract program is indeed a property of the concrete program. This framework relies heavily on identifying a suitable abstract domain, implementing the approximation of the concrete operations on the abstract domain, and most importantly, defining a widening operator on the abstract domain for approximating the looping structure in finitely many steps leading to a fixed point. The analysis is done using forward collecting semantics. The choice of abstract domain and the associated widen operator are critical in the ability to prove useful properties of concrete properties of programs.

These analyses generally trade, large amounts of, precision for scalability. Although some analyses have been successfully used for verification, such as [38], they generally produce many false positives and care must be used in when/where these analyses are deployed [13]. Our current experience with the BSQChk prover is that it can spend considerable time checking assertions that could easily be discharged by a more efficient data flow [29] or numeric analysis [30].

Symbolic Execution and Fuzzing: While verification and abstract interpretation are generally focused on over approximation to show that certain program states are infeasible, symbolic execution [2, 4, 9, 23, 27] and concrete fuzzing (white, grey, or black box) [16, 57] focus on exploring possible executions looking for error states. As discussed in Section 2, as long as the error of interest has a small scope property then these techniques are quite effective. However, in cases where the error requires a large number of path expansions, where the input must be large, or where there complex constraints on the input that block accessibility to the error, these techniques become substantially less effective. Since the full encoding in the BosqueIR tool can see both forward constraints from the input validation and backwards constraints from the error context it does not suffer from the same limitations and can easily find witness inputs even when the small context hypothesis does not hold.

Incorrectness Logic and Under Approximate Analysis: Incorrectness Logic [44] and other under approximate approaches [5], that cannot prove the absence of an error but instead can prove the presence of a fault, represent an interesting and recent development in the design space of program analysis. These systems look to fuse the power of symbolic representations to capture many concrete states while under (rather than over) approximating reachability. The goal is to build tools that provide a "no false positives" guarantee while still finding as many bugs as practical. A

1079 property that the BSQChk also has as part of the workflow that maximizes *actionable* information
1080 for the developer.
1081 Interestingly, one of the motivations for introducing Incorrectness Logic is that (p. 4) "...the exact
1082 reasoning of the middle line of the diagram [strongest post semantics] is definable mathematically
1083 but not computable (unless highly incomputable formulae are used to describe the post)." However,
1084 as shown in this paper, this middle line of exact and decidable semantics is practical to compute
1085 in most cases when the language semantics are designed appropriately. Further, by encoding the
1086 exact semantics in a decidable fragment of logic, the BosqueIR language and BSQChk checker
1087 provide the best of both the verification and fault detection worlds (it satisfies both correctness and
1088 incorrectness logics).
1089
1090
7 CONCLUSION

1091 This paper presented an approach to programming language design that was centered around the
1092 co-design of the language and an encoding into decidable fragments of logic that are efficiently
1093 dischargeable using modern SMT solvers. The resulting language, BosqueIR, and the BSQChk
1094 automated checking tool show that this is an effective technique. The encoding is both semantically
1095 precise and complete for most of the language (and has effective heuristics for the incomplete
1096 fragments). This enables the BSQChk checker to provide *actionable* results to a developer for builtin
1097 errors as well as user specified asserts, data invariants, or pre/post conditions.
1098 We are actively working with collaborators at Morgan Stanley to apply this methodology to
1099 the example code and regulatory examples that are currently implemented in the Morphir frame-
1100 work [39]. Our initial experience has been very positive with some small bugs, such as in our
1101 introductory example, found and the tool showing excellent performance in practice. Our big
1102 challenge at this point is the lack of explicit assertions in most of the code which limits us to
1103 checking for predefined error classes such as invalid casts of artihmatic overflow. We plan to
1104 investigate how to make including assertions simpler for developers and running the checker on
1105 larger quantities of code.
1106 The ability to effectively reason about the precise behavior of an application creates opportunities
1107 for not only a powerful suite of checker tools but also for program synthesizers, code optimization
1108 tools and accelerator architecture support, and application lifecycle management systems. As a
1109 result we hope this new approach to thinking about programming and programming languages
1110 will lead to massively improved developer productivity, increased software quality, and enable a
1111 new golden age of developments in compilers and developer tooling.
1112
1113
ACKNOWLEDGMENTS

1114 Thanks to Stephen Goldbaum and Attila Mihaly for their insights into the Morgan Stanley Morphir
1115 Framework and their input on earlier iterations of the BosqueIR language.
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, Premkumar T. Devanbu, Mark Marron, and Charles A. Sutton. Mining semantic loop idioms. *IEEE Transactions on Software Engineering*, 44, 2018.

[2] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM*, 2004.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[4] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Thresher: Precise refutations for heap reachability. In *PLDI*, 2013.

[5] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. Racerd: Compositional static race detection. 2018.

[6] V8 doesn't stable sort, 2018. https://bugs.chromium.org/p/v8/issues/detail?id=90.

[7] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, 2002.

[8] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA*, 1998.

[9] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *TACAS*, 2004.

[10] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press Ltd., London, UK, UK, 1972.

[11] Leonardo de Moura, Nikolaj Bjørner, and et. al. Z3 SMT Theorem Prover. `https://github.com/Z3Prover/z3`, 2021.

[12] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. Narcissus: Correct-by-construction derivation of decoders and encoders from binary formats. In *ICFP*, 2019.

[13] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. *Commun. ACM*, 62:62–70, 2019.

[14] F* language, 2021. https://www.fstar-lang.org/.

[15] R. W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19:19–32, 1967.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. PLDI, 2005.

[17] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *OOPSLA*, 2012.

[18] Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In *CAV*, 2009.

[19] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO*, 2011.

[20] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, 2001.

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

[22] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39:92–106, 2004.

[23] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11, 2002.

[24] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. ISSTA, 1996.

[25] Martin Jonáš and Jan Strejček. Is satisfiability of quantified bit-vector formulas stable under bit-width changes? (experimental paper). LPAR, 2018.

[26] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, USA, 2000.

[27] Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. Corral: A solver for reachability modulo theories. In *CAV*, 2012.

[28] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, 2007.

[29] Chris Lattner, Andrew Lenharth, and Vikram S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, 2007.

[30] Vincent Laviron and Francesco Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *VMCAI*, 2009.

[31] K. Rustan M. Leino. Developing verified programs with dafny. In *ICSE*, 2013.

[32] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *APLAS*, 2005.

[33] K. Rustan M. Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In *CAV*, 2016.

[34] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[35] Kenneth L. McMillan and Oded Padon. Deductive verification in decidable fragments with ivy. In *SAS*, 2018.

[36] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. Parallel inclusion-based points-to analysis. In *OOPSLA*, 2010.

[37] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology*, 14:1–41, 2005.

1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225

[38] Antoine Miné. The octagon abstract domain. *Higher Order Symbolic Computation*, 19:31–100, 2006.
[39] Morphir runtime, 2020. https://github.com/finos/morphir.
[40] Yannick Moy, Nikolaj Bjørner, and Dave Sielaff. Modular bug-finding for integer overflows in the large: Sound, efficient, bit-precise static analysis. Technical Report MSR-TR-2009-57, 2009.
[41] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag, Berlin, Heidelberg, 1999.
[42] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. Solving quantified bit-vectors using invertibility conditions. CAV, 2018.
[43] James Noble, Andrew P. Black, Kim B. Bruce, Michael Homer, and Mark S. Miller. The left hand of equals. In *Onward!*, 2016.
[44] Peter W. O'Hearn. Incorrectness logic. In *POPL*, 2019.
[45] Grant Passmore, Simon Cruanes, Denis Ignatovich, Dave Aitken, Matt Bray, Elijah Kagan, Kostya Kanishev, Ewen Maclean, and Nicola Mometto. The Imandra automated reasoning system (system description). IJCAR, 2020.
[46] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in f*. In *ICFP*, 2017.
[47] Juan Pedro Bolívar Puente. Persistence for the masses: Rrb-vectors in a systems language. *Proc. ACM Program. Lang.*, 1(ICFP), 2017.
[48] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
[49] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *PLDI*, 2008.
[50] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Non-linear loop invariant generation using gröbner bases. In *POPL*, 2004.
[51] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *Advances in Neural Information Processing Systems 31*, pages 7751–7762. 2018.
[52] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.
[53] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. SAS, 2011.
[54] Jesse A. Tov and Riccardo Pucella. Practical affine types. In *POPL*, 2011.
[55] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.
[56] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42, 2013.
[57] Michal Zalewski. American Fuzzy Lop (AFL). `https://lcamtuf.coredump.cx/afl/`, 2021.
[58] Hantao Zhang, Deepak Kapur, and Mukkai S. Krishnamoorthy. A mechanizable induction principle for equational specifications. CADE, 1988.