Random Testing via Runtime Abstract Interpretation

ZAIN K AAMER✉, University of Pennsylvania, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA

Property-based testing of C programs can be automated by synthesizing random input generators from separation-logic specifications. Existing work in this space, such as the Bennet testing tool, uses randomized backtracking search, generating random values and checking them against constraints, backtracking on failure. Although this approach performs well on simple recursive heap structures, it struggles as constraints grow more complex, particularly when they involve pointer arithmetic—as, for example, in the many forms of specialized storage allocators that arise in low-level systems software. Existing work uses targeted optimizations and heuristics to satisfy specific classes of constraints, but this requires continual expansion as new special cases arise, resulting in complex tools.

We reframe generation as the iterative refinement of abstract domain elements, where sampling a concrete value is the final refinement. By applying abstract interpretation at runtime to obtain an abstract element, we obtain a lightweight form of constraint solving and propagation that enables randomized testing of programs with complex preconditions. We identify three strategies for applying abstract interpretation: (1) *speculative refinement*, refining abstract elements before sampling based on immediately following constraints, (2) *corrective refinement*, calculating "desired" abstract elements from information gleaned from failed constraints, and (3) *cascading propagation*, propagating information from failures to components of compound expressions. We formalize these ideas in a generator DSL whose monadic semantics are parametric over abstract domains.

We implement this DSL in a new tool called Lucas and evaluate it on sixteen workloads: the six original case studies from the Bennet paper, six position-independent data structures, and four free-list allocators. Comparing configurations with and without refinement, we find that refinement finds bugs in all four allocators and in two of the position-independent data structures that Bennet-style random backtracking fails to find.

CCS Concepts: • **Software and its engineering → General programming languages**; **Software testing and debugging**.

Additional Key Words and Phrases: Random Testing, Property-based Testing, Separation Logic, Abstract Interpretation

ACM Reference Format:
Zain K Aamer and Benjamin C. Pierce. 2026. Random Testing via Runtime Abstract Interpretation. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 381 (October 2026), 29 pages. https://doi.org/10.1145/3839513

1 Introduction

Property-based testing (PBT) is a popular method for random testing. Initiated by the QuickCheck library in Haskell [17], PBT has since been deployed in a wide range languages and tools, from interactive theorem provers (such as Rocq [33] and Isabelle [9]) and functional languages (such as OCaml [53] and Scala [42]) to low-level languages including C [57].

PBT depends on *input generators* to produce random values that can be used to evaluate whether an *executable property* holds of the system under test. These generators must often be written manually, in particular when the property under test has a sparse precondition, so that simple

Authors' Contact Information: Zain K Aamer, University of Pennsylvania, Philadelphia, USA, zaamer@seas.upenn.edu; Benjamin C. Pierce, University of Pennsylvania, Philadelphia, USA, bcpierce@cis.upenn.edu.

ACM 2475-1421/2026/10-ART381
https://doi.org/10.1145/3839513

rejection sampling—blindly generating random inputs and throwing away those that fail the precondition—wastes most of its time generating and discarding invalid tests.

A promising alternative is to use an executable variant of *separation logic* [49] for specifying preconditions. Separation logic is already used by a large number of verification tools for low-level languages, including Infer [10], VST [6, 11, 58], VeriFast [31], Viper [40], Gillian [23, 35], and CN [47]. While separation logic is primarily used for full-blown correctness proofs, there has also been some work on using separation-logic specifications to inform test input generation, most recently with Bennet [3], which synthesizes input generators from CN preconditions. Bennet runs on top of Fulminate [7], a lower-level tool that instruments C functions based on preconditions and postconditions. Bennet provides *input generators* for sparse preconditions while Fulminate checks *executable postconditions*, allowing us to automatically apply PBT to a C function with a CN specification.

The primary bottleneck in this process turns out to be the speed at which useful inputs can be generated. Bennet works by converting CN preconditions into a dedicated generator DSL, which is optimized and compiled into a testing harness in C. The compiled generators use randomized backtracking search: they incrementally check generated values against relevant constraints and, after a violation, regenerate the relevant parts of the input. While backtracking is lightweight and performs well for simple data structures, it struggles as the number of constraints increases, especially those involving complex bitvector formulas. Bennet uses targeted optimizations and heuristics to handle specific constraint classes, such as linear inequalities and alignment requirements, but this approach requires continual expansion and has complicated the system over time. For example, Bennet pattern-matches on expressions of the form `x < e`, where all variables in `e` are defined before `x`, and it restricts the generated values of `x` to be less than `e`. It includes many similar patterns to account for different types of casts. Also, in addition to logical constraints (`x == y + z`), Bennet must deal with resource constraints, which correspond to exclusive memory ownership in the original specification ("`p` points to 8 bytes of exclusively owned memory"). When an assignment requires more memory than is currently allocated, Bennet reports a dedicated allocation failure, requiring a separate case in the DSL's semantics.

The observation motivating the present work is that both logical and resource constraints can be captured by abstract domains. Rather than handling allocation failures separately, as Bennet does, we model the allocation state around a pointer—how many bytes must be valid before and after it—as an abstract domain. Generation then becomes *iterative refinement* of abstract elements, where each constraint refines the element from which values are sampled and a concrete value is the final refinement, and we can apply *runtime abstract interpretation*, using backward abstract transformers to obtain an abstract element from which sampling is more likely to satisfy a given constraint. This process acts as a form of lightweight constraint solving and propagation, allowing us to test programs with complex ownership involving pointer arithmetic, such as allocators. We identify three separate strategies for applying abstract interpretation: (1) *speculative refinement*, refining the abstract element before sampling based on the immediately following constraints, (2) *corrective refinement*, obtaining "desired" abstract elements from failed constraints and carrying them back as information, and (3) *cascading propagation*, propagating information from failures to components of compound expressions.

We formalize these ideas in a new generator DSL whose monadic semantics are parametric over an abstract domain, building upon Bennet's original semantics. This scheme naturally extends to products of the allocation domain with others, such as wrapped intervals [25, 41] and congruences [29]. For example, wrapped intervals can solve linear inequalities without resorting to narrow optimizations, while congruences can handle alignment constraints. In particular, speculative refinement replaces Bennet's pattern-based specializations with backward abstract transformers

supplied by the selected domains. Thanks to the allocation abstract domain, the special casing from the Bennet DSL largely disappears.

We implement the new DSL in a tool called Lucas, reusing the CN and Bennet frontends. We evaluate Lucas's bug-finding performance under various configurations, enabling and disabling different refinement strategies, both on the six original Bennet case studies and on ten new case studies: six position-independent data structures and four free-list allocators.

In summary, our contributions are:

- We demonstrate how runtime abstract interpretation can perform lightweight constraint solving and propagation for random input generation. We frame generation as iterative refinement of abstract elements and identify three strategies for applying abstract interpretation: speculative refinement (refining before sampling), corrective refinement (obtaining desired abstractions from failed constraints), and cascading propagation (propagating failure information to subexpressions) (Section 2).
- We formalize these ideas in a new generator DSL, generic over abstract domains, that builds on Bennet's original monadic semantics. We show how allocation requirements can be captured by an abstract domain and combined via reduced products with domains such as wrapped intervals and congruences to handle more complex constraints (Section 3).
- We describe Lucas, an implementation of this DSL that reuses the CN and Bennet frontends (Section 4), and evaluate its bug-finding performance against a Bennet-style baseline on the six original Bennet case studies plus ten new workloads, featuring six position-independent data structures and four free-list allocators (Section 5).

We conclude with related work (Section 6) and a discussion of limitations and future work (Section 7).

2 Overview

In this section, we motivate our work by walking through a CN predicate describing "intrusive" linked lists and showing how Fulminate [7] can check this predicate against a concrete heap. We then derive a random generator from this predicate using the translation strategy from the original Bennet tool. We illustrate why Bennet's backtracking search struggles to satisfy the predicate and how runtime abstract interpretation addresses the issue.

2.1 Checking that a Heap Satisfies a Precondition

Consider a function that swaps the integers referenced by two pointers. Both pointers must be valid pointers to integers. In separation logic, we can express this requirement as $\exists n, m.\ p \mapsto n * q \mapsto m$. The separating conjunction ($*$) asserts that the heap consists of two disjoint parts: one where `p` maps to `n` and one where `q` maps to `m`. While this may not be strictly necessary for this swap, it allows for convenient compositional reasoning in program verification. Given a concrete heap, one way to check this condition would be to enumerate partitions of the heap into two disjoint parts, one for each conjunct. If the wrong partitioning is selected, the checker would have to backtrack, potentially searching over exponentially many partitions.

Furthermore, checking existentially quantified formulas can also involve backtracking: checking a heap against $\exists p.\ p \mapsto 27$ could require iterating over every address in the heap, and within a larger formula one may need to backtrack among several candidate pointers. Fortunately, in many useful specifications, the existentially quantified variables are uniquely determined by the values of the function arguments and the heap.

```
void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}
```

For example, given $\exists v.p \mapsto v$, a concrete heap, and the value of p, we can simply look up the value stored at p. This restriction, combined with limiting disjunctions to if-statements, is enforced by the syntax of CN's specification language. Banerjee et al. [7] showed that this restricted syntax allows a formula to be checked effectively against a heap at runtime.

To understand CN's syntax, let's return to the formula $\exists\, v.\, p \mapsto v$, where p is given. We'll assume for this example that heaps store only integers. We would like to transform this formula into a function, which captures the intuition for taking a concrete heap and reading the value of v from it.

The type of the points-to relation is $\mapsto$: $\texttt{Heap} \to \texttt{Loc} \to \mathbb{Z} \to \mathbb{B}$. For all types A, the set of functions of type $\texttt{A} \to \mathbb{B}$ is isomorphic to $\mathcal{P}(\texttt{A})$. Therefore, the type of $\mapsto$ is equivalent to $\texttt{Heap} \to \texttt{Loc} \to \mathcal{P}(\mathbb{Z})$. However, since there is a single value at a given location in a heap (or none at all), this is equivalent to $\texttt{Heap} \to \texttt{Loc} \to \texttt{Option}\ \mathbb{Z}$, which captures the ability to take a concrete heap and read the value stored in it.

We can solve our problem with separating conjunctions and exponential partitions by removing the Heap argument and replacing our option monad ($\texttt{Option}\ \mathbb{Z}$) with a state-option monad ($\texttt{MarkedHeap} \to \texttt{Option}\ (\mathbb{Z}, \texttt{MarkedHeap})$). The points-to operation takes a heap and, if the given location is present in the domain of the heap and was previously unmarked, marks it as "owned," returning the updated heap along with the value read from that location. If a location is already marked, we return None, guaranteeing exclusive ownership. This new points-to function, of type $\texttt{Loc} \to \texttt{MarkedHeap} \to \texttt{Option}\ (\mathbb{Z}, \texttt{MarkedHeap})$, captures the intuition of CN's read-write ownership primitive, **RW**.[1] Any use of the **RW** primitive is equivalent to the grounded existentials we've discussed, where they are uniquely determined by the arguments and the heap. As such, CN does not have general existential quantifiers, only allowing new values to be introduced via **RW**. Now checking $\texttt{p} \mapsto \texttt{n} * \texttt{q} \mapsto \texttt{m}$ is just calling **RW** on p, marking it in the heap, and returning n, then calling **RW** on q on the updated heap. If $\texttt{p} = \texttt{q}$, the second call sees q is already marked and returns None. This guarantees the disjointness implied by the separating conjunction.

In CN's syntax, the specification for swap uses **take** constructs, which can be thought of as monadic binds in a language like Haskell. Each **take** passes the MarkedHeap state resulting from its body to the following lines, exiting early with None if the body returned None. The first **take** binds the value at p to n, and the second binds the value at q to m. The postcondition says that, after swap, the value at p equals the old value of q and vice versa. The logical constraints (n == m_after and m == n_after) are evaluated as guards, returning None if false.

```
requires
  take n = RW<int>(p);
  take m = RW<int>(q);
ensures
  take n_after = RW<int>(p);
  take m_after = RW<int>(q);
  n == m_after; m == n_after;
```

Now, let us consider a more complicated example: an implicit free list allocator. An implicit free list organizes a contiguous buffer into a sequence of blocks, each beginning with a header that stores the block's size and whether it is allocated or free. There is no explicit linked list of free blocks. Instead, the allocator traverses the list implicitly by advancing from one header to the next using the size field. The C struct and the CN datatype we need are shown below.

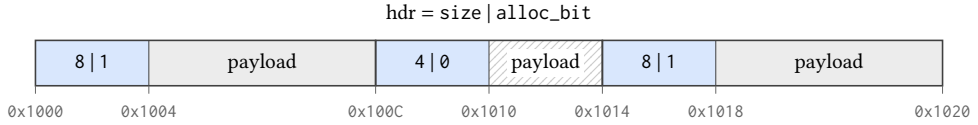
```
struct block_header { unsigned int raw_size; }

datatype block_list = { Nil {},
  Cons { u32 size, boolean is_free, datatype block_list tl } }
```

[1] Current versions of CN call this primitive **RW** (read-write permission), while earlier versions, including the one used by Aamer and Pierce [3], called it **Owned**.

We will define a predicate, `BlockList`, which will take a pointer to the first element and a pointer to the end of the allocation region, recursively building a list of blocks.

First, let us understand the memory layout. Each block's header packs the block's payload size and an allocation bit into a single `unsigned int`, effectively forming an intrusive linked list over the raw memory. The diagram below shows a buffer containing two allocated blocks and one free.



The header contains the size of the node (excluding the header itself). The last bit is used to mark whether or not the block has been allocated. We can write a CN predicate that walks the block list:

```
predicate (datatype block_list) BlockList(
  pointer p, pointer end
) {
    if (p >= end) {
        assert(p == end);
        return Nil {};
    } else {
        take H = RW<block_header>(p);
        let size = H.raw_size & (~1u32);
        let is_free = (H.raw_size & 1u32) == 0u32;
        let q = array_shift<char>(p, sizeof<block_header>);
        let payload_size = is_free ? size : 0u32;
        take Payload = each (u64 i; i < (u64)payload_size) {
            RW<char>(array_shift<char>(q, i))
        };
        let next = array_shift<char>(q, (u64)size);
        take Rest = BlockList(next, end);
        return Cons{size: size, is_free: is_free, tl: Rest}; } }
```

This predicate uses several CN constructs beyond the **take**/**RW** pattern introduced earlier. CN's pointer arithmetic uses **array_shift**<char>(p, n), which is equivalent to the C expression ((**char***)p) + n. It also uses **each**, which expresses iterated ownership:
take Payload = **each** (u64 i; i < n) { **RW**<char>(...) } takes ownership of n consecutive bytes, binding the result to Payload. When a block is allocated (is_free is false), payload_size is zero and the loop claims nothing. When it is free, the loop claims ownership of every payload byte.

Consider the heap we showed in the diagram, a contiguous buffer at address 0x1000 containing three blocks, two allocated and one free. We check BlockList with p = 0x1000 and end = 0x1020.

Since p < end, we take the else branch. We take ownership of the header via **RW**<block_header>(0x1000), marking 0x1000–0x1004 as owned and binding H to {raw_size = 9}. We compute size = 9 & ~ 1 = 8 and is_free = ((9 & 1) = 0) = false, so the block is allocated. Since is_free is false, payload_size = 0 and the **each** loop iterates zero times. We compute q = 0x1004 and next = 0x1004 + 8 = 0x100C, then recurse.

In the second call with p = 0x100C, we again take the else branch. The header at 0x100C yields raw_size = 4, giving size = 4 and is_free = true (the alloc bit is 0). Now payload_size = 4, so

the **each** loop takes ownership of 4 bytes at `0x1010`–`0x1014`. We compute `next` = `0x1010` + 4 = `0x1014` and recurse.

In the third call with `p` = `0x1014`, the header yields `raw_size` = 9, so `size` = 8 and `is_free` = `false`. The **each** loop again iterates zero times. We compute `next` = `0x1018+8` = `0x1020`. Recursing with `p` = `0x1020`, we find `p` ≥ `end` and `p` = `end`, so the assertion passes and we return `Nil {}`.

Unwinding the recursion, the predicate returns the blocks: `[(8, false), (4, true), (8, false)]`. All memory has been marked with no overlapping ownership and all assertions satisfied, so the predicate is satisfied.

2.2 Generating Heaps with Lucas

We have now seen how to *check* whether a concrete heap satisfies a predicate. What about *generating* a heap that satisfies it?

Building on the intuition of Banerjee et al. [7], Aamer and Pierce [3] observed a natural interpretation of CN predicates as generators. They defined a generator DSL capturing this intuition and a translation from CN specifications to the DSL. We reuse the basic syntax and translation from CN, but provide it with novel semantics and extend the syntax.

When translating, each **take**/**RW**, which reads from the heap and marks ownership, becomes a **let***/**arbitrary** that generates a random value and an assignment (`:=`) that writes it to the heap. Assertions remain assertions, and pure expressions are unchanged. The translated generator for `BlockList` is shown below.

```
generator (datatype block_list) GenBlockList(
  pointer p, pointer end
) {
    if (p >= end) {
        assert(p == end);
        return (Nil {});
    } else {
        let* H = arbitrary<block_header>();
        block_header p := H;
        let size = H.raw_size & (~1u32);
        let is_free = (H.raw_size & 1u32) == 0u32;
        let q = array_shift<char>(p, sizeof<block_header>);
        let payload_size = is_free ? size : 0u32;
        map (u64 i; i < (u64)payload_size) {
            let* c = arbitrary<char>();
            char array_shift<char>(q, i) := c;
            return c
        };
        let next = array_shift<char>(q, (u64)size);
        let* rest = GenBlockList(next, end);
        return (Cons { size: size, is_free: is_free, tl: rest });
    }
}
```

Compared with the predicate in Section 2.1, **take** `H` = **RW**`<block_header>(p)` became **let*** `H` = **arbitrary**`<block_header>()` followed by `block_header p := H`, which generates a

random header and writes it to the heap (checking that it does not overlap previous assignments). The **each** loop became a **map**, which generates and assigns each byte of the payload, while everything else remains the same.

This generation process is captured in a monad that threads the heap state through computations and supports exceptions for backtracking. When a constraint is violated or an assignment fails, the generator backtracks to an earlier choice point and retries. In order to generate pointers, we allocate a large buffer when the testing harness begins and randomly choose locations contained within that buffer. This allows us to easily randomize the placement of data in memory. Then a given pointer `p` is valid if it lies within the buffer.[2] We do this so that we can get a good shuffling of the order of pointers, as opposed to simply relying on the non-determinism of `malloc`.

Suppose we call `GenBlockList(p, end)` with `p = 0x0800` and `end = 0x1020`, where `0x0800` lies outside our pre-allocated pointer generation buffer. Since `p < end`, we enter the `else` branch and generate a random header `H` via **`let*`** `H = ` **`arbitrary`**`<block_header>()`. We then try to write it with `block_header p := H`, but the assignment fails because `0x0800` is not valid memory. At this point we need `p` to point to at least **`sizeof`**`(block_header)` bytes of writable memory. We backtrack, carrying that allocation requirement to the caller, which regenerates `p` as `0x1000` (inside the buffer) and calls `GenBlockList` again.

Now consider the same situation two recursive calls deep: block A at `0x1000` and block B at `0x100C` have already been generated, and `next = ` **`array_shift`**`<char>(q, size)` has landed at an address outside the buffer. The allocation failure is not about the original `p` but about the `size` chosen for block B one level up. We need to propagate this requirement back through the call stack. Rather than special-casing **`array_shift`**, we treat allocation requirements as an abstract domain (made concrete in the next subsection). We fix block B's `size` and apply backward abstract interpretation to determine what memory `q` needed. We use overapproximate backward abstract interpretation, where, given a constraint, it determines an abstract element for each variable such that all satisfying assignments are possible. For example, if block B's `size` is `4` and `q` is **`array_shift`**`<char>(p, ` **`sizeof`**`<block_header>)`, then `next` is **`array_shift`**`<char>(q, 4)`. If we know that `next` must be a valid pointer to 4 bytes of memory, then we can calculate that `q` must point to at least 8 bytes of valid memory, and then `p` must point to at least `8 + sizeof<block_header>` bytes of valid memory. This is cascading propagation.

Suppose instead the assignment at `0x1000` succeeds: we mark bytes `0x1000`–`0x1004` as assigned (the header) and obtain a random `H` with, say, `raw_size = 0xFFFE`, so the block is free. After masking, `size = 0xFFFE` is far larger than the buffer. The **`map`** loop tries to write `0xFFFE` payload bytes starting at `0x1004`, which immediately fails. We backtrack to the **`let*`** where `H` was generated and try again.

What if `size` were merely large enough to overshoot `end`? For example, `size = 24` places the next block at `0x101C`, which is fine, but a further recursive call finds `p = 0x1038 > end`, and the assertion **`assert`**`(p == end)` fails. Blind backtracking would discard all generated blocks and start over. Instead, we carry an interval for `size` (e.g. $[0, \texttt{end} - \texttt{q}]$) back to the choice point, so the next attempt picks a value that keeps the block list within bounds. It is important to note that this is not relational. Rather, we substitute the concrete values of `end` and `q` before applying backward abstract interpretation. This is corrective refinement.

Finally, suppose the allocator enforces a minimum block size (**`assert`**`(size >= 4)` after computing `size`). Rather than generating arbitrary headers and discarding those that violate the constraint, we apply backward abstract interpretation *before* sampling: the assertion gives us a lower bound on

[2]Some functions may depend on a pointer being `malloc`'d, so that it may be `free`'d. Fulminate does not support such specifications, and since we use it as our checker, we can't either.

`size`, and from `size = H.raw_size & (~1u32)` we derive an interval for `raw_size` that satisfies the constraint. We then sample `raw_size` directly from that interval, avoiding wasted attempts. This is speculative refinement.

2.3 The Allocation Domain

We next show how allocation information forms an abstract domain. This domain is *non-relational*, meaning that each variable has its information tracked independently. We then define a *basis*, which shows how to abstract a single variable. An element of the abstract domain would then be a map from variable names to elements of the basis.

An assignment in the DSL to an address implies two requirements: (1) the entire assignment lies within allocated memory and (2) no other assignment overwrites it.

For each **`RW`** that appears in the CN specification, a corresponding assignment is generated for placing a value at that location. In order to maintain the requirement that **`RW`**'d locations are non-overlapping, we ensure that our assignments also don't overlap (i.e., no byte is written to twice). This requirement is naturally handled by the randomness of pointer generation.

The first, however, is a constraint on the amount of memory that must be writable before and after a given pointer. That is, for a pointer `p`, what are the maximum `n` and `m` such that `p + n` is a valid pointer and `p - m` is too? Since pointers are just bit vectors, the range of valid memory locations is much smaller than all possible values of a pointer.

The requirement for a pointer to have `n` bytes allocated after (or before) it can be captured by a non-relational abstract domain. For each pointer, we track a pair (b, a) representing the number of bytes that must be valid *before* and *after* the address. These pairs form a lattice (Figure 1): the top element $(0, 0)$ imposes no allocation requirement, and elements grow more precise as `b` or `a` increases, with $\bot$ representing any unsatisfiable requirement where $b + a$ exceeds the maximum allocation size. This lattice will be the "allocation basis".

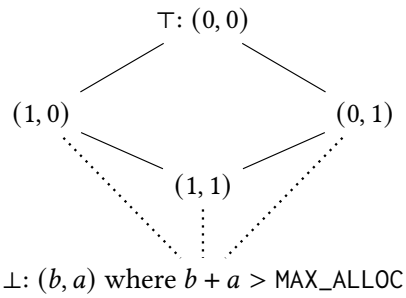


Fig. 1. Allocation basis. The top element $(0, 0)$ has no requirements on valid addresses. Pairs (b, a) represent bytes valid before and after an address. The bottom element represents any unsatisfiable allocation where $b + a >$ `MAX_ALLOC`.

If an assignment fails due to the address `p` not being able to have 8 bytes written to it, then the allocation info corresponding to `p` would be $(0, 8)$. If we had no prior information about `p`, it would've been $\top$, so the failure moves `p` down the lattice, restricting its possible values. When a constraint requires more memory around a pointer than is already available, that constraint's requirement must be lower on the lattice. We can then take the meet to obtain an element of the basis that the pointer should be sampled from.

Both b and a are non-negative, representing *how many* bytes are required, not offsets. This means that addresses can potentially overlap, which reduces the amount of bookkeeping Lucas must do. Instead, disjoint ownership is checked for when assignments are made. If an overlap occurs at the point of an assignment, a failure causes backtracking, leading to a new pointer being generated. In practice, this performs well at satisfying disjoint ownership.

Formally, meets and joins in this basis are defined component-wise:

$$(b_1, a_1) \sqcap (b_2, a_2) = \begin{cases} (\max(b_1, b_2),\ \max(a_1, a_2)) & \text{if } \max(b_1, b_2) + \max(a_1, a_2) \leq \texttt{MAX\_ALLOC} \\ \bot & \text{otherwise} \end{cases}$$

$$(b_1, a_1) \sqcup (b_2, a_2) = (\min(b_1, b_2),\ \min(a_1, a_2))$$

For example, if we have an allocator that stores header information before the pointer to the data, we'd need **`sizeof`**`(node_header)` bytes before it. If it were 4 bytes, we would want $(4, 0)$. If we also had 32 bytes of data, we'd want $(0, 32)$ for that. Together their meet would be $(4, 32)$.

To sample from an element of this basis, recall that the buffer's address is non-deterministic, so we do not know its location ahead of time. Given a buffer of size N starting at address s, the concretization of (b, a) is $\gamma((b, a)) = \{s + i \mid b \leq i \leq N - a\}$. That is, it is the set of addresses within the buffer that have at least b valid bytes before them and a valid bytes after. The concretization of $\top = (0, 0)$ is the entire buffer, and that of $\bot$ is $\emptyset$.

Thus, for a specific run of Lucas, the allocation domain is effectively an interval. Applying backward abstract interpretation to the result of an **`array_shift`** —turning allocation information about its result into information about its arguments —just involves basic arithmetic. If **`array_shift`**`<char>(p, 10)` must belong to $(4, 8)$, then `p` must belong to $(0, 18)$ (`max(0, 4 - 10)` and `max(0, 8 + 10)`). When two pointers are constrained to be equal, we take the meet of their requirements and impose it on both. Casts between pointer and integer types leave allocation requirements unchanged.

3 Abstracting Generator Semantics over Abstract Domains

Having given the intuition for the Lucas DSL, we now make its syntax and semantics concrete. Following the style of Aamer and Pierce [3], we use a monadic semantics. We also define program transformations for enabling each of our refinement strategies.

3.1 DSL Grammar

We use the metavariables `cty` for C types and `bty` for CN types. The DSL's complete syntax is:

```
c ::= Logical(e) | Owned(e, n)| OwnedRange(e1, n, e2)
e ::= x | n | e1 == e2 | e1 < e2 | e1 || e2 | array_shift<cty>(e1, e2) | ...
g ::= arbitrary<cty>() | pick([g1, ..., gm]) | cty e := e'; g' | let* x = g; g'
    | GenName(x1,...,xn) | if (e) { g1 } else { g2 } | assert(e); g' | return e
    | map (bty x; e) { g } | return_domain(e) | assert_domain(c1, ..., cm); g'
    | arbitrary_domain<cty>(c1, ..., cm) | arbitrary_domain_prop<cty>(c1, ..., cm)
gen ::= generator bty GenName(bty1 x1, ..., btyn xn){ g }
```

The last four forms (**`return_domain`**, **`arbitrary_domain`**, **`arbitrary_domain_prop`**, **`assert_domain`**) and constraints, `c`, are novel additions. The remainder are adapted from Bennet [3], though with novel semantics. We refer the reader to the Bennet paper for the translation from CN to this DSL.

These novel variants will be created from the simpler form (e.g., converting **assert** to **assert_domain**) via optional transformations defined alongside our monadic semantics. Making them optional allows the user to choose refinement strategies and manage the overhead they add to generators. Additionally, designing the DSL in this way means that future work has the freedom to be more selective about when our new strategies are used. For example, one could use speculative

refinement for one variable (use **`arbitrary_domain`**), but not another (use `arbitrary`), as the rest of the DSL is agnostic to the refinement strategies in use.

The constraints, `c`, are also new, capturing what the immediately following constraints expect for the new variants covered. A `Logical(e)` constraint requires the pure boolean expression `e` to hold. An `Owned(e, n)` constraint requires `e` to evaluate to the address of an `n`-byte cell that is valid and has not been assigned to. An $\texttt{OwnedRange}(\texttt{e}_1, \texttt{n}, \texttt{e}_2)$ constraint requires the same of each `n`-byte cell at the addresses $\texttt{e}_1, \texttt{e}_1 + \texttt{n}, \ldots$ strictly below $\texttt{e}_2$.

The user never writes these constraints themselves. When the transformations that introduce our novel forms are applied, a simple scan-ahead static analysis computes the set of constraints. Starting from the statement being rewritten, it walks forward over the generator body and collects a `Logical(e)` constraint from each **`assert`**`(e)`, an `Owned(e, sizeof(cty))` constraint from each assignment `cty e := e'`, and an `OwnedRange` constraint from each loop that assigns every cell of a contiguous array range—the compiled form of an iterated separating conjunction whose guard restricts the index to a contiguous range. The analysis only collects constraints whose variables are already in scope, and it stops at any **`pick`**, **`if`**, or **`return`**.

The `e` are "pure" expressions supported by CN and Fulminate, such as pointer arithmetic, inequalities, and boolean operators. The pure expression **`array_shift`**`<int>(p, 10)` is equivalent to the C expression **`(int*)`**`p + 10`.

3.2 Abstract Domains

Our semantics are parameterized by a two-layer non-relational abstract domain: *basis* elements describe possible values of one variable, while elements of the full *abstract domain* map variables to basis elements. Variables are thus tracked independently, without relationships between them [38].

Each basis (e.g., intervals, congruences) must implement the following signature, where `B` is the carrier type for elements of the basis:

$$\begin{aligned}\texttt{Basis} = \{\ &\texttt{top} :: \texttt{B},\ \texttt{bottom} :: \texttt{B},\ \texttt{meet} :: \texttt{B} \to \texttt{B} \to \texttt{B},\ \texttt{join} :: \texttt{B} \to \texttt{B} \to \texttt{B},\\ &(=) :: \texttt{B} \to \texttt{B} \to \texttt{Bool},\ (\leq) :: \texttt{B} \to \texttt{B} \to \texttt{Bool},\\ &\texttt{backward} :: \texttt{Constraint} \to \texttt{Sym} \to \texttt{Option}\ (\texttt{Map}\ \texttt{Sym}\ \texttt{B}) \to \texttt{B}\ \}\end{aligned}$$

Let $\gamma : B \to \mathcal{P}(\texttt{Val})$ be the concretization function for a basis `B`, mapping a basis element to the set of concrete values it represents. The partial order $(\leq)$ must agree with concretization, so $b_1 \leq b_2$ implies $\gamma(b_1) \subseteq \gamma(b_2)$, and `bottom` and `top` are the least and greatest elements, concretizing to the empty set and the set of all values. Importantly, `meet` and `join` must over-approximate intersection and union of the concretization, meaning $\gamma(b_1) \cap \gamma(b_2) \subseteq \gamma(\texttt{meet}(b_1, b_2))$ and $\gamma(b_1) \cup \gamma(b_2) \subseteq \gamma(\texttt{join}(b_1, b_2))$. When a basis is a lattice, these are the exact meet and join, but bases need not be lattices; for example, wrapped intervals [25] provide an *over-meet* and *over-join* that satisfy our requirements.

Finally, `backward` is the backward abstract transformer for a single constraint. Given a constraint and a symbol `x` (and optionally an abstract state, mapping symbols to basis elements), `backward` returns a basis element containing all values of `x` that can satisfy the constraint, making it an overapproximate analysis. That is, if some assignment maps `x` to v and satisfies the constraint, then v lies in the concretization of the returned element. When an abstract state is supplied, only assignments consistent with it need be considered, namely those mapping each variable to a value represented by its element.

For example, given the constraint $x < 10$ over an unsigned integer, `backward` may return the interval $[0, 9]$, but not $[0, 8]$. Thus speculative refinement never removes a satisfying value, provided sampling has support over every concrete value the resulting element represents. This is a completeness property of the sampling space, not the source of generator soundness. Every

$$
\begin{aligned}
&[\![\textbf{let*}\ \mathtt{x} = \mathtt{return(\_domain)?\ e}; \mathtt{g_{rest}}]\!] = \lambda \mathtt{env}.\ \lambda \mathtt{state}.\\
&\quad \textbf{let}\ \mathtt{v} = \mathtt{evalPure(e[state.vars])}\ \textbf{in}\\
&\quad \textbf{match}\ [\![\mathtt{g_{rest}}]\!]\ (\mathtt{env})(\mathtt{state}\{\mathtt{vars} \leftarrow \mathtt{state.vars} \cup \{\mathtt{x} \mapsto \mathtt{v}\}\})\ \textbf{with}\\
&\quad \mid \mathtt{Error\ (is\_young\ , xbs)}\ \textbf{when}\ \mathtt{x} \in \mathtt{dom(xbs)} \Rightarrow\\
&\qquad \textbf{let}\ \mathtt{xbs'} = \mathtt{Map.delete(x,\ xbs)}\ \textbf{in}\\
&\qquad \mathtt{Error\ (is\_young,\ refineReturn(x, e, state, xbs(x)) \cup xbs')}\\
&\quad \mid \mathtt{result} \Rightarrow \mathtt{result}
\end{aligned}
$$

$$
\mathtt{refineReturn}^{\mathtt{return}}(\mathtt{x}, \mathtt{e}, \mathtt{state}, \mathtt{b_x}) = \{\mathtt{y} \mapsto \mathtt{SpecialBasis.top} \mid \mathtt{y} \in \mathtt{freeVars(e)}\}
$$

$$
\begin{aligned}
&\mathtt{refineReturn}^{\mathtt{return\_domain}}(\mathtt{x}, \mathtt{e}, \mathtt{state}, \mathtt{b_x}) =\\
&\quad \textbf{let}\ \mathtt{basisFor(y)} =\\
&\qquad \textbf{let}\ \mathtt{c} = \text{“}\mathtt{x == e}\text{”}[\mathtt{Map.delete(y, state.vars)}]\ \textbf{in}\\
&\qquad \mathtt{SpecialBasis.backward(c,\ y,\ Some\ \{x \mapsto b_x\})}\ \textbf{in}\\
&\quad \{\mathtt{y} \mapsto \mathtt{basisFor(y)} \mid \mathtt{y} \in \mathtt{freeVars(e)}\}
\end{aligned}
$$

Fig. 2. Monadic Semantics of **let*** _ = **return**(_domain)? _

sampled candidate is still checked against the original constraint, and a violating candidate is rejected or causes backtracking. Corrective refinement and cascading propagation occur only after failures, so they cannot prevent a failure-free execution from generating any otherwise reachable satisfying input.

For Lucas, we work with `SpecialBasis`, the reduced product of any number of bases, which the user selects via the CLI, with the allocation basis of Section 2.3. We also generate the functions

$$\mathtt{getAllocInfo} : \mathtt{Basis} \to \mathtt{AllocInfo} \qquad \mathtt{ofAllocInfo} : \mathtt{AllocInfo} \to \mathtt{Basis}$$

where `AllocInfo` is the basis for allocation information.

The full abstract domain is then a map `AbstractState = Map Sym SpecialBasis`, with all operations handled element-wise. A variable absent from the map is treated as `SpecialBasis.top`.

3.3 Monadic Semantics

We define the semantics for our generator DSL as a monad. Following the previous subsection, we parameterize everything over `SpecialBasis`. The semantics is defined by a translation to the reader-state-exception monad `Gen a`, given below.

```
GenState = { seed :: RandomSeed, heap :: MarkedHeap, vars :: Map Sym cnPExp }
GenEnv a = { ctx :: GenCtx a, allocBounds :: Loc × Loc }
GenCtx a = Sym → [a] → Gen a
Result a = Success a | Error (Bool × AbstractState)
Gen a = GenEnv a → GenState → Result (a × GenState)
```

Some of the generator state overlaps with the Bennet DSL [3]. The state, `GenState`, holds a random seed (`RandomSeed`), a marked heap (`MarkedHeap`) tagging each location as assigned or not, and a map from argument variables to values, stored as pure CN expressions (`cnPExp`).

The environment (`GenEnv`) includes a context (`GenCtx`) storing each generator's name and arguments for calling it, and `allocBounds`, the first and last usable address of the allocation domain.

```
[[cty e_addr ≔ e_val ; g]] = λenv. λstate.
  let p = evalPure (e_addr[state.vars]) in
  let (lo, hi) = env.allocBounds in
  let isAllocated = lo ≤ p ∧ p + sizeof(cty) ≤ hi in
  if not isAllocated then
    let x_p = pointerVar (e_addr) in
    let b = SpecialBasis.backward(Owned(e_addr, sizeof(cty)), x_p, None) in
    if b = SpecialBasis.bottom then
      Error (true, {x ↦ SpecialBasis.top | x ∈ freeVars(e_addr)})
    else
      Error (true, {x_p ↦ b})
  else if isMarked (p, sizeof(cty), state.heap) then
    Error (true, {x ↦ SpecialBasis.top | x ∈ freeVars(e_addr)})
  else
    let value = evalPure (e_val[state.vars]) in
    let heap′ = heapAssign (p, cty, value, state.heap) in
    [[g]] (env) (state{heap ← heap′})
```

Fig. 3. Monadic Semantics of Assignment

`Result`'s `Error` case carries a boolean flagging whether it is "young" (has not backtracked before a blamed non-deterministic value) and an abstract state restricting what resampling may select. The monad, `Gen a`, maps the environment and state to `Success`, carrying a value of type `a` and an updated state, or `Error` on failure. Pure expressions are evaluated by `evalPure : cnPExp → cnPExp`, which we reuse from Fulminate's internals.

The semantics of the DSL is a function $[[\cdot]]$ from DSL terms to `Gen cnPExp`. All generators produce CN values, represented as constant pure expressions. The rest of this subsection explains how this function is defined.

We now define the semantics of each construct in turn. Each `_domain` variant shares the structure of its base form and differs only in a named refinement helper, so we present each pair as one parameterized definition, giving the helper's variants below the main rule.

3.3.1 Cascading Propagation. When the right-hand side of a **`let*`** is **`return`** `e` or **`return_domain`** `e`, the value of `x` is deterministic (Figure 2). The pure expression `e` is evaluated with the current variable bindings, and the result `v` is bound to `x` before the rest of the generator is run. If the rest of the generator fails, blaming `x`, we cannot regenerate `x` since it is fully determined by other variables. Instead, we assign blame to the free variables of `e` via `refineReturn`. We keep `is_young` unchanged, as it is deterministic. Its role is explained under **`arbitrary`**.

The two variants differ only in `refineReturn`. Plain **`return`** discards all refinement information, only identifying the variables to blame with `SpecialBasis.top`. The **`return_domain`** variant improves on this by propagating abstract information backward through the expression. For each free variable `y` in `e`, we substitute every other variable with its value to get `e′` and then invoke `SpecialBasis.backward` on the equation `x == e′`, seeded with the basis that `x` was blamed with,

$$
\begin{aligned}
&[\![\textbf{assert}(\texttt{\_domain})?(c_1, \ldots, c_m); g]\!] = \lambda \texttt{env}.\ \lambda \texttt{state}.\\
&\quad \textbf{if}\ \texttt{evalPure}\ (c_1[\texttt{state.vars}]) \wedge \cdots \wedge \texttt{evalPure}\ (c_m[\texttt{state.vars}])\\
&\quad \textbf{then}\ [\![g]\!]\ (\texttt{env})\ (\texttt{state})\\
&\quad \textbf{else}\\
&\qquad \textbf{let}\ F = \{i \mid \neg\ \texttt{evalPure}\ (c_i[\texttt{state.vars}])\}\ \textbf{in}\\
&\qquad \texttt{Error}\ (\texttt{true},\ \{x \mapsto \texttt{basisFor}(x) \mid x \in \texttt{freeVars}(\{c_i \mid i \in F\})\})
\end{aligned}
$$

$$\texttt{basisFor}^{\texttt{assert}}(x) = \texttt{SpecialBasis.top}$$

$$\texttt{basisFor}^{\texttt{assert\_domain}}(x) = \bigsqcap_{i=1}^{m} \texttt{SpecialBasis.backward}(c_i[\texttt{state.vars}],\ x,\ \texttt{None})$$

Fig. 4. Monadic Semantics of **assert**(_domain)?

to compute a more precise abstract element for y. This reduces the waste of information and should decrease the number of failed regenerations upstream.

When the user enables cascading propagation in Lucas, it replaces all **let*** x = **return** e; g' with **let*** x = **return_domain** e; g'.

3.3.2 Corrective Refinement. Corrective refinement aims to improve the information learned from constraint failures. We'll begin with the assignment statements, which are necessary for Lucas to generate inputs.

We assume an interface over MarkedHeap as follows.

$$\texttt{isMarked} :: \texttt{Loc} \to \texttt{Size} \to \texttt{MarkedHeap} \to \texttt{Bool},$$

$$\texttt{heapAssign} :: \texttt{Loc} \to \texttt{CType} \to \texttt{Value} \to \texttt{MarkedHeap} \to \texttt{MarkedHeap}$$

The function isMarked checks whether any bytes in the range have already been marked, and heapAssign marks and writes a value of the given type to the pointer. Bytes are marked when they are assigned to, so by avoiding double-marking, we ensure non-overlapping assignments.

The assignment expression (Figure 3) evaluates the target address and checks that the required bytes are both unmarked and allocated. If the bytes are not allocated, it uses SpecialBasis.backward on the Owned constraint to refine the address's pointer variable. Here pointerVar, given a pure expression, returns the name of the variable the address is computed from—either a pointer-typed variable or, generalizing Bennet [3], a bitvector variable cast to a pointer. As in Bennet, we require that address expressions contain a single such variable. If that refinement is SpecialBasis.bottom—no choice of pointer could satisfy the assignment—it instead blames all of the address's free variables with SpecialBasis.top. If the bytes have already been marked, it throws a young failure blaming the address's free variables with SpecialBasis.top. Otherwise, it evaluates the value, writes it to the heap with heapAssign, and continues.

For logical constraints, **assert** and **assert_domain** (Figure 4) evaluate their constraints and, if they fail, throw a young failure mapping each free variable to a basis element via basisFor. The constraints list of **assert_domain** is computed via the scan-ahead analysis in Section 3.1. Plain **assert** discards all refinement information, using SpecialBasis.top. The **assert_domain** variant identifies which constraints failed and blames the free variables that appear in those constraints. For each blamed variable, it uses SpecialBasis.backward on all constraints to compute a more precise basis element.

The most interesting case is when the right-hand side of a **let*** has the form **arbitrary**<cty>(), where we can exploit failure information to refine what we sample (Figure 5). To lighten the

```
[[let* x = arbitrary<cty>(); g_rest]]
    = fst(retryArb(max_tries, x, SpecialBasis.top, SpecialBasis.top, cty, [[g_rest]]))

retryArb(n, x, b, b_tmp, cty, rest) =
   get >>= λoldState.
   sample cty b_tmp >>= λv. λenv. λstate.
   match rest (env)(state{vars ← state.vars ∪ {x ↦ v}}) with
   | Error (is_young , xbs) when x ∈ dom(xbs) ⇒
      let alloc = SpecialBasis.getAllocInfo(xbs(x)) in
      let is_bottom = (Error (false, Map.delete(x, xbs)), Some (xbs(x))) in
      if alloc ≠ AllocInfo.top then
         let b'_tmp = SpecialBasis.meet (b, SpecialBasis.ofAllocInfo(alloc)) in
         if b'_tmp = SpecialBasis.bottom then is_bottom else
         let state' = oldState{seed ← oldState.seed} in
         retryArb(n, x, b'_tmp, b'_tmp, cty, rest) (env) (state')
      else if n > 0 then
         let b'_tmp = SpecialBasis.meet (b, xbs(x)) in
         if b'_tmp = SpecialBasis.bottom then is_bottom else
         let b' = if is_young then b'_tmp else b in
         let state' = oldState{seed ← newSeed(state.seed)} in
         retryArb(n − 1, x, b', b'_tmp, cty, rest) (env) (state')
      else
         (Error (false, Map.delete(x, xbs)), None)
   | result ⇒ (result, None)
```

Fig. 5. Monadic Semantics of **let*** _ = **arbitrary**<cty>()

handling of state, we use the standard state-monad operation `get`, which returns the current state, and the Haskell notation >>= for monadic bind, of type `Gen a → (a → Gen b) → Gen b`. We also use `sample cty b`, which draws a concrete value of C type `cty` from the concretization of `b`. The function `retryArb` maintains two basis elements: `b`, the *accumulated* refinement that persists across retries, and b_{tmp}, a *temporary* refinement that may incorporate less reliable information, meaning that it depends on values that were generated after this point. Both begin at `SpecialBasis.top`. When the rest of the generator fails blaming `x`, three cases arise:

(1) **Allocation refinement.** If `getAllocInfo(xbs(x))` yields informative allocation information (not `AllocInfo.top`), it is met into the persistent basis `b` (which also becomes the new b_{tmp}) and we retry *without* decrementing `n`, replaying the same random seed. We do not want to bias our generators toward small (in terms of memory usage) inputs, which is why we replay the same choices and do not decrement the number of tries.
(2) **Standard refinement.** Otherwise, if retries remain (`n > 0`), we meet the accumulated basis `b` with `xbs(x)` to obtain a refined b'_{tmp}. If the error is *young*—meaning it has not been caught and rethrown by an intermediate **let***—the abstract information is fresh, so we also update the

$$
\begin{array}{l}
[\![\textbf{let}\star\ \mathsf{x} = \textbf{arbitrary\_domain}(\_\mathsf{prop})?\texttt{<cty>}(\mathsf{c}_1,\ \ldots,\ \mathsf{c}_m); \mathsf{g}_{\mathsf{rest}}]\!] = \lambda \mathsf{env}.\ \lambda \mathsf{state}. \\
\quad \textbf{let}\ (\mathsf{b}, \mathsf{E}) = \mathsf{meetBases}_{\mathsf{i}=1}^{\mathsf{m}} \\
\qquad (\mathsf{SpecialBasis.top})\ \mathsf{SpecialBasis.backward}(\mathsf{c}_\mathsf{i}[\mathsf{state.vars}], \mathsf{x},\ \mathsf{None})\ \textbf{in} \\
\quad \textbf{if}\ \mathsf{E} \neq \emptyset\ \textbf{then}\ \mathsf{Error}\ (\mathsf{true},\ \{\mathsf{y} \mapsto \mathsf{SpecialBasis.top} \mid \mathsf{y} \in \mathsf{E}\}) \\
\quad \textbf{else match}\ \mathsf{retryArb}(\mathsf{max\_tries}, \mathsf{x}, \mathsf{b}, \mathsf{b}, \mathsf{cty}, [\![\mathsf{g}_{\mathsf{rest}}]\!])\ \textbf{with} \\
\qquad \mid (\mathsf{result},\ \mathsf{None}) \Rightarrow \mathsf{result} \\
\qquad \mid (\mathsf{result},\ \mathsf{Some}\ (\mathsf{b}')) \Rightarrow \\
\qquad\quad \textbf{let}\ (\_, \mathsf{E}') = \mathsf{meetBases}_{\mathsf{i}=1}^{\mathsf{m}}(\mathsf{b}') \\
\qquad\qquad \mathsf{SpecialBasis.backward}(\mathsf{c}_\mathsf{i}[\mathsf{state.vars}], \mathsf{x},\ \mathsf{None})\ \textbf{in} \\
\qquad\quad \textbf{if}\ \mathsf{E}' \neq \emptyset \\
\qquad\quad \textbf{then}\ \mathsf{propagate}(\mathsf{x}, \mathsf{b}', \mathsf{E}', \mathsf{state}) \\
\qquad\quad \textbf{else}\ \mathsf{result}
\end{array}
$$

$$
\mathsf{propagate}^{\mathsf{arbitrary\_domain}}(\mathsf{x}, \mathsf{b}', \mathsf{E}', \mathsf{state}) = \mathsf{Error}\ (\mathsf{true},\ \{\mathsf{y} \mapsto \mathsf{SpecialBasis.top} \mid \mathsf{y} \in \mathsf{E}'\})
$$

$$
\begin{array}{l}
\mathsf{propagate}^{\mathsf{arbitrary\_domain\_prop}}(\mathsf{x}, \mathsf{b}', \mathsf{E}', \mathsf{state}) = \\
\quad \textbf{let}\ \mathsf{hint} = \mathsf{Some}\ \{\mathsf{x} \mapsto \mathsf{b}'\}\ \textbf{in} \\
\quad \textbf{let}\ \mathsf{basisFor}(\mathsf{y}) = \mathsf{fst}(\mathsf{meetBases}_{\mathsf{i}=1}^{\mathsf{m}}(\mathsf{SpecialBasis.top}) \\
\qquad \mathsf{SpecialBasis.backward}(\mathsf{c}_\mathsf{i}[\mathsf{state.vars}], \mathsf{y}, \mathsf{hint}))\ \textbf{in} \\
\quad \mathsf{Error}\ (\mathsf{true},\ \{\mathsf{y} \mapsto \mathsf{basisFor}(\mathsf{y}) \mid \mathsf{y} \in \mathsf{E}'\})
\end{array}
$$

Fig. 6. Monadic Semantics of **let*** _ = **arbitrary_domain**(_prop)?<_>(_)

persistent basis b to $\mathsf{b}'_{\mathsf{tmp}}$. If the error is not young, b is left unchanged, since the information may be stale after having passed through other backtrack points. We then retry with a fresh seed and one fewer attempt.

(3) **Exhaustion.** We allow a user to set the maximum number of tries (`max_tries`) that backtracking can have, which is by default 25. When no retries remain, the error is propagated with x removed from the blamed variables.

Corrective refinement depends on the failure information learned from violated constraints being used to refine the abstract element from which the next sample is drawn. This should guide generation toward values more likely to satisfy the constraints.

3.3.3 Speculative Refinement. However, we can do more than just handle failures. With speculative refinement, we can consider constraints before we even generate a value.

The function $\mathsf{meetBases}_{\mathsf{i}=1}^{\mathsf{m}}(\mathsf{b}_\emptyset)\ f(\mathsf{i})$ takes a starting basis element $\mathsf{b}_\emptyset$ and iteratively meets it with the results of applying f to each index. For each constraint, if the meet would result in `SpecialBasis.bottom`, it instead stores the free variables that appeared in that constraint. It then returns a pair containing the result of the meets and the free variables that appeared in any constraints that resulted in bottom.

Both **arbitrary_domain** and **arbitrary_domain_prop** (Figure 6) take a list of constraints $\mathsf{c}_1, \ldots, \mathsf{c}_m$ that mention the variable being generated. Before sampling, the constraints are analyzed via `meetBases` and `SpecialBasis.backward` to compute a refined basis b. If any constraint reduces to bottom (i.e., the set E is non-empty), an error is raised immediately, blaming the free

```
[[let* x = g_inner ; g_rest]] = retryLet(max_tries, x, [[g_inner]], [[g_rest]])

retryLet(n, x, inner, rest) =
  get >>= λoldState.
  inner >>= λv. λenv. λstate.
  match rest (env)(state{vars ← state.vars ∪ {x ↦ v}}) with
  | Error (_ , xbs) when x ∈ dom(xbs) ⇒
    if n > 0 then
      let state' = oldState{seed ← newSeed(state.seed)} in
      retryLet(n − 1, x, inner, rest) (env) (state′)
    else
      Error (false, Map.delete x xbs)
  | result ⇒ result
```

Fig. 7. General Semantics of **let***

variables of the conflicting constraints. Otherwise, `retryArb` is invoked with the refined basis instead of `SpecialBasis.top`, so that the first sample is already drawn from a restricted domain.

When `retryArb` aborts because a refinement met to bottom, it returns the residual basis b' that `x` was last blamed with, and the constraints are re-evaluated against b'. If any now yield bottom (set E'), the variable can no longer satisfy them, so `propagate` is called to assign blame upstream.

The two variants differ only in `propagate`. Plain **arbitrary_domain** discards all refinement information, blaming variables with `SpecialBasis.top` (analogous to $\texttt{refineReturn}^{\texttt{return}}$). The **arbitrary_domain_prop** variant improves on this by running `SpecialBasis.backward` with a *hint*—the residual basis b' for `x`—to compute a tighter basis for each blamed variable (analogous to $\texttt{refineReturn}^{\texttt{return\_domain}}$).

When the user enables speculative refinement in Lucas, it replaces all `let* x = arbitrary<cty>(); g'` with `let* x = arbitrary_domain<cty>(c₁, ..., cₘ); g'`, where $c_1, \ldots, c_m$ are the constraints that the scan-ahead analysis of Section 3.1 collects from the body following the **arbitrary**, keeping only those that mention the **let***'s name. There is an additional user setting which instead replaces them with the `_prop` variant to further enable backward propagation of blame.

3.3.4 Remaining Pieces. Since we are in a monad, **return** is simply a monadic return. The **if**-statements are standard **if**-statements. The **pick** performs random sampling of branches without replacement (shuffle the branches and if the previous one failed, try the next). The **map** form evaluates its inner generator for each index that satisfies the guard and returns a dictionary from indices to values; if a failure occurs, it does not backtrack between elements but exits the loop.

The general semantics of **let*** is shown in Figure 7. The translation of **let*** when the right-hand side is any other generator form invokes `retryLet`, saving the current state, evaluating the inner generator to obtain a value `v`, and then evaluating the rest of the generator with `v` assigned to `x`. If `rest` produces an error that blames `x` (i.e., $x \in \texttt{dom}(\texttt{xbs})$), we regenerate `x`. It resets to the old saved state with a new random seed and recurses with one fewer attempt remaining. When we run out of attempts, the error is propagated upward, with `x` removed from the blame set. The error is also marked as no longer young (`false`), since it has backtracked past a non-deterministic value that was blamed. In all other cases, where it is a success or an error that does not blame `x`, the result

passes through unchanged. Note that this general case performs no abstract domain refinement. Since the inner generator is opaque, we can only retry with a fresh seed upon failure.

3.4 Sized Generation

Inspired by Bennet [3] and QuickCheck [17], we define a notion of "sized generation". For each test case, a positive `size` is chosen, which constrains the depth of recursive calls and the values generated by **`arbitrary`**. For intervals, sizing is simple: we sample from `[0, size - 1]` for unsigned values and `[-size + 1, size - 1]` for signed values. We generalize this for our abstract domains by saying that for unsigned values, we sample from the `size` values closest to 0, and for signed values, from the $2 \times \texttt{size} - 1$ values closest to 0.

More formally, let $\gamma : A \to \mathcal{P}(\mathsf{BV}_n)$ be the concretization function for an abstract domain A over n-bit values. For an abstract element $\alpha \in A$ and a positive integer `size`, the sized concretization $\gamma_{\texttt{size}}(\alpha)$ selects the k values in $\gamma(\alpha)$ closest to zero, as shown on the right, where $|v|$ is the absolute value of v and $k = \min(\texttt{size}, |\gamma(\alpha)|)$ for unsigned types and $k = \min(2 \times \texttt{size} - 1, |\gamma(\alpha)|)$ for signed types. When $\alpha = \top$, this recovers the standard QuickCheck-style intervals: $[0, \texttt{size} - 1]$ for unsigned and $[-\texttt{size} + 1, \texttt{size} - 1]$ for signed types. We implement samplers from this set for each domain and their reduced products with the allocation domain.

$$\gamma_{\texttt{size}}(\alpha) = \underset{\substack{S \subseteq \gamma(\alpha) \\ |S| = k}}{\arg\min} \sum_{v \in S} |v|$$

To manage depth, we follow Bennet's approach exactly. Recursive generators have a size argument added. If the path has a single recursive call, the size argument is `size - 1`, while a path with n recursive calls has a size of `size / n`. Exhausting the budget raises a depth exception, causing generation to backtrack and choose a path that avoids the recursive call. The size argument for the initial call is determined per input generation attempt.

3.5 Replacing Bennet's Special Cases

Bennet specializes generation by pattern-matching syntactic forms. For an inequality such as `x < y + z`, where `y` and `z` are already defined, it restricts the values generated for `x`. The pattern requires a variable on one side, casts need extra patterns, and resource requirements such as insufficient allocation use dedicated failure types. Each new constraint form requires another pattern or failure case in the DSL.

Lucas instead recovers these specializations through speculative refinement. Its scan-ahead analysis sees the upcoming constraint `x < y + z` and substitutes the already-generated values of `y` and `z`. The wrapped-interval backward transformer then recovers the upper bound for `x`, reproducing Bennet's specialization without a dedicated inequality pattern. Congruences refine alignment constraints like `x % 8 == 0`, tristate numbers refine bitmasks like `(x & 0xff) == 0x40`, and reduced products combine them. Adding support means defining reusable backward transformers rather than enumerating the patterns each constraint may appear in.

4 Implementation

Lucas's generator synthesis algorithm uses CN's frontend to parse preconditions, then turns them into generators using Bennet's frontend. These synthesized generators are compiled to C, using a new runtime we've developed. For our PRNG, we use SplitMix [52] instead of the 64-bit Mersenne Twister [36, 43], which Bennet used, as we found SplitMix to be faster.

Each basis provides backward tests for the comparisons it can refine and backward transformers for the operations it can soundly invert. The wrapped-interval basis tests equality and inequality using the rules of Gange et al. [25], and inverts negation, addition, subtraction, and casts. The congruence basis follows Granger [29], adapted to machine arithmetic by reducing each modulus

by $\gcd(\cdot, 2^w)$ at width w. It inverts addition, subtraction, negation, multiplication, shifts, division, and remainders, so a remainder constraint refines its argument to a congruence class. The tristate-number basis [20] refines through the bitwise operations and constant shifts. For the allocation basis, see Section 2.3. Since the user can choose many combinations of abstract domains, we use that selection to synthesize a reduced product in C, which is then linked with the runtime library at test time.

5 Evaluation

Our evaluation is designed to answer the following Research Questions:

- **RQ1 (Overhead):** How much does it cost to apply all of our refinement strategies in situations where Bennet-style naive generation already succeeds at finding bugs?
- **RQ2 (New Bug Finding Abilities):** Do our refinement strategies find bugs that Bennet-style generation does not?
- **RQ3 (Ablation):** Are all of the refinement strategies and abstract domains essential, or can some be dropped without affecting bug-finding performance?

5.1 Experimental Setup

Our bug-finding experiment for RQ1 and RQ2 compares two basic configurations of Lucas. The first, *Everything Off*, disables all of our refinement strategies and every abstract domain except the allocation domain, which is always enabled, as it is necessary for generating valid pointers. With the refinement strategies disabled, Lucas is a reimplementation of the approach used by Bennet [3], so we use this configuration as our Bennet-style baseline in place of the original—built on the newest version of Fulminate along with various implementation fixes and upgrades—letting us focus on the impact of our refinement strategies rather than of implementation details. The second configuration, *Everything On*, enables all our refinement strategies (cascading propagation, corrective refinement, and **`arbitrary_domain_prop`**, the propagating variant of speculative refinement) and all abstract domains (wrapped intervals, congruences, tristate numbers, and allocation).

Outside of these experiments, the refinement strategies and the domains other than allocation are opt-in, enabled via CLI flags. Both configurations use the default backtracking budget of 25 tries (`max_tries`), the same as Bennet.

Our experiments have two parts: a bug-finding study and an ablation study. For the first, we ran Lucas in the *Everything Off* and *Everything On* configurations across sixteen workloads, split into two groups: workloads where enabling refinement does not increase the solved-task count, where we consider the overhead it adds, and workloads where it does, where we look at how bug-finding ability improves. For the second, we take the most difficult case studies from the first, ablate each major component, and record how it impacts bug-finding effectiveness.

For all of our benchmarking, we use Etna [51], an evaluation tool for property-based testing tools that automatically injects and tests *mutants*. A mutant is an incorrect version of the program under test. For each mutant, we specify the functions that can reach the mutated code and for which at least one precondition-satisfying input causes Fulminate to report the resulting specification violation. Each pair of mutant and function is a *task*. We run up to 10 trials for each task—fewer when we stop early, as described below—with a timeout of two minutes. For each task, we report the arithmetic mean of the number of backtracks, the number of discards (generation attempts that failed and were thrown away), the number of precondition-satisfying inputs tested, and the time taken to find the bug. In all of our bug-finding tables, these means (with standard error) are taken over the tasks that every configuration solves, so that all configurations are compared on the same set of tasks. Configurations that solve nothing on a workload are dropped rather than allowed to

shrink this common set. Backtracks and discards measure wasted search effort. Unlike wall-clock time, they are largely independent of implementation constants and hardware, changing only with the timeout.

If a configuration is able to detect the bug in all completed trials, we say that it *solves* the task. If it fails to generate a single valid (precondition-satisfying) input in at least one trial, we say that the task is *unsatisfied*. If neither condition holds (i.e., it generates at least one valid input in all completed trials, but fails to detect the bug in at least one trial), we say that the task is *unsolved*.

For practicality, we skip a task's remaining trials once any trial fails to generate a valid input (already unsatisfied) or misses the bug (already cannot be solved). This affects neither the classifications nor the aggregate statistics, which are computed only over solved tasks, which always run all of their trials. An unsolved task may still find the bug given more time, as it is generating *some* valid inputs. An unsatisfied task, by contrast, does effectively no testing, as inputs are not being generated.

Since specification development involves frequently modifying preconditions, resulting in new generators being synthesized, we include the time taken to synthesize, compile, and link the generators in the time-to-bug numbers. We run our benchmarks on a MacBook Pro with an M3 processor and 36GB of RAM.

5.2 Case Studies

Table 1 summarizes our benchmark suite, consisting of sixteen workloads in four groups. The six critical code and recursive data structures workloads are from [3]. The six position-independent data structures and four free-list allocators are new.

Table 1. Workloads. Lines of code (LoC) is of the mutant-free program, including specifications. LoC for SortedList, BST, and AVL is the median across all versions.

Workload	LoC	Mutants	Functions	Tasks
Critical Code				
RingQueue	194	4	2	4
Runway	281	5	2	6
MKM	865	2	1	2
Recursive Data Structures				
SortedList	108	6	2	6
BST	576	26	4	32
AVL	716	24	3	26
Position-Independent Data Structures				
PI-SLL	665	14	9	14
PI-DLL	681	18	9	18
PI-CLL	821	27	11	27
PI-SortedList	763	23	10	23
PI-BST	891	18	6	21
PI-AVL	1,089	31	8	61
Free List Allocators				
IFL	913	11	5	24
IFLC	1,070	17	6	36
EFL	1,119	8	5	24
EFLC	1,351	11	5	31
Total	12,103	245	88	355

Critical Code. Three of the Bennet case studies have small pieces of code with specifications that involve logical constraints. The first is a ring queue, which uses an underlying ring buffer [30]. The second is an airport runway simulation, which tracks planes waiting to arrive and depart [48]. Its key difficulty is a complex validity constraint which tightly restricts the set of valid states. The third is a mission key management system (MKM), which manages private keys [24].

Recursive Data Structures. The other three Bennet case studies are standard "recursive data structures with constraints." The sorted-list case study is actually made up of three versions with different approaches to how the constraints are written. The BSTs and AVL trees are both made up of two versions of specifications.

Position-Independent Data Structures. We developed six position-independent (PI) workloads, which include singly-linked (PI-SLL), doubly-linked (PI-DLL), circular (PI-CLL), and sorted singly-linked (PI-SortedList) lists, BSTs (PI-BST), and AVL trees (PI-AVL). In a position-independent layout, the entire structure is contained

Table 2. Bold indicates the lowest mean and highest solved count per workload, except ties. S/U/UN denotes solved in every trial, unsolved, and unsatisfied (no valid input generated in some trial).

Workload	Configuration	Backtracks	Discards	Inputs	Time (s)	S/U/UN
RingQueue	Everything On	4.7 ± 1.1	0.0 ± 0.0	3.4 ± 0.6	0.32 ± 0.00	4/0/0
	Everything Off	**4.1 ± 1.0**	0.0 ± 0.0	**3.0 ± 0.4**	**0.22 ± 0.00**	4/0/0
Runway	Everything On	**5.0 ± 2.4**	0.0 ± 0.0	**28.4 ± 12.2**	3.89 ± 1.34	6/0/0
	Everything Off	6.5 ± 3.3	0.0 ± 0.0	39.2 ± 19.4	**0.61 ± 0.04**	6/0/0
MKM	Everything On	**35.5 ± 25.9**	0.0 ± 0.0	**21.1 ± 14.4**	0.44 ± 0.00	2/0/0
	Everything Off	36.5 ± 31.2	0.0 ± 0.0	22.4 ± 18.4	**0.37 ± 0.00**	2/0/0
SortedList	Everything On	**671.8 ± 398.0**	0.6 ± 0.3	10.5 ± 4.1	0.28 ± 0.00	6/0/0
	Everything Off	26.0k ± 21.9k	**0.2 ± 0.1**	**8.2 ± 2.8**	**0.22 ± 0.02**	6/0/0
BST	Everything On	**578.2 ± 155.1**	9.7 ± 2.8	30.0 ± 7.8	0.46 ± 0.02	32/0/0
	Everything Off	13.2k ± 3.8k	**2.0 ± 0.7**	**16.9 ± 4.1**	**0.33 ± 0.00**	32/0/0
AVL	Everything On	**23.0k ± 18.6k**	573.0 ± 525.9	552.6 ± 492.5	1.73 ± 0.99	26/0/0
	Everything Off	1.1M ± 857.7k	**54.0 ± 46.3**	**150.4 ± 125.5**	**0.96 ± 0.35**	26/0/0
PI-SLL	Everything On	**26.0k ± 10.6k**	**20.5 ± 8.0**	**3.1 ± 1.4**	0.50 ± 0.03	14/0/0
	Everything Off	284.2k ± 149.9k	924.2 ± 565.1	105.1 ± 70.1	**0.48 ± 0.09**	14/0/0
PI-SortedList	Everything On	**46.3k ± 20.3k**	**36.9 ± 15.4**	**4.7 ± 2.6**	0.70 ± 0.11	23/0/0
	Everything Off	626.2k ± 352.1k	1.5k ± 788.5	140.3 ± 72.8	**0.68 ± 0.19**	23/0/0
PI-BST	Everything On	**10.5k ± 2.7k**	**5.2 ± 1.6**	2.9 ± 1.0	11.27 ± 2.92	13/8/0
	Everything Off	256.4k ± 51.6k	6.8 ± 1.3	**2.7 ± 0.5**	**0.63 ± 0.05**	**21/0/0**
PI-AVL	Everything On	**16.0k ± 2.3k**	**10.8 ± 2.1**	**5.2 ± 1.1**	15.11 ± 1.78	37/24/0
	Everything Off	1.3M ± 469.7k	78.0 ± 38.7	18.3 ± 7.9	**1.08 ± 0.20**	**55/6/0**

within a single buffer, with nodes pointing to each other via offsets into the buffer rather than via absolute pointers. This layout allows the structure to be easily serialized, copied, and freed. These preconditions can be difficult to satisfy because ownership must be established by traversing the buffer through these offsets to build up the data structure, adding a layer of indirection.

Free-List Allocators. We also developed four free-list allocators, including an implicit free list (IFL)—the running example of Section 2—the same allocator with boundary tags to support coalescing (IFLC), and explicit free list variants of both (EFL and EFLC). We focus on storage allocators for our evaluation because their invariants combine pointer arithmetic and constraints with variably sized payloads.

5.3 RQ1: Should Everything Always Be On?

Aamer and Pierce [3] demonstrated the bug-finding ability of naive random backtracking generation for CN specifications. As we consider new refinement strategies, we need to ask whether they are cheap enough to be unconditionally enabled, or whether they need to be used more sparingly.

To address this question, we collect the workloads on which enabling our strategies did not increase the solved-task count in Table 2. This includes all the original Bennet case studies (where all bugs are already found with the *Everything Off* configuration), plus some of the position-independent data structures, specifically those with acyclic references (neither doubly-linked nor circular linked lists), where the *Everything Off* configuration also performs well.

The *Everything On* configuration eventually finds all the bugs that *Everything Off* does except for PI-BST and PI-AVL, which now time out on some tasks, and it is slower for many of them. For these two workloads, the performance degrades so much that some bugs that were found by *Everything*

Off are no longer found by *Everything On*. While *Everything On* solves 13 of PI-BST's 21 tasks and 37 of PI-AVL's 61 tasks, *Everything Off* solves all 21 PI-BST tasks and 55 PI-AVL tasks.

The runway simulation is the extreme example among the workloads that both configurations fully solve. There, the more costly strategies lead to additional overhead, with *Everything On* averaging 3.89s to find the bug, against 0.61s with *Everything Off*. At the same time, the refinement strategies result in decreased backtracking with *Everything On*, even where wall-clock time increases—from 1.1M to 23.0k on the AVL tree, and from 1.3M to 16.0k on PI-AVL. That is, the strategies succeed at their goal of eliminating wasted search. The added time comes from the cost of the refinement machinery itself, not from additional searching.

While the overhead for some of the Bennet case studies could doubtless be reduced by further optimizing the implementation, even a faster implementation would not eliminate the slowdown entirely, because refinement changes the dynamics of generation. The combined effects of refinement on generation and the rest of the testing pipeline can substantially increase time-to-bug, for example, raising the mean on PI-BST from 0.63 to 11.27 seconds.

Our refinement strategies also shift the output distribution of our generators, by focusing on satisfying constraints given the choices already made. For example, suppose we generate $x \in \{0, \ldots, 8\}$ and $y \in \{0, \ldots, 9\}$, with the constraint $x < y$. Without refinement, a fixed choice $x = k$ satisfies the constraint with probability $(9 - k)/10$, favoring smaller values of x ($x = 0$ has nine satisfying choices of y, while $x = 8$ has only one). With speculative refinement and wrapped intervals, once $x = k$ is chosen we restrict y to $\{k + 1, \ldots, 9\}$, so every choice of x completes to a satisfying pair and its distribution becomes uniform rather than weighted by number of satisfying partners. This shifts bug-finding performance, and can make it worse. If bugs are mostly triggered by inputs the unrefined, weighted distribution favors—small x here—the uniform distribution samples them less often, and time to find a bug can rise.

To summarize: Most of the time, *Everything On* is competitive with *Everything Off*, imposing only minor performance penalties on the majority of the workloads. However, on the harder position-independent case studies, it *does* make a significant difference. This suggests that leaving the refinement strategies off by default is the right choice.

5.4 RQ2: Do We Find Bugs That Naive Generation Cannot

The experiments above show that our refinement strategies can impose substantial overhead on workloads for which naive random backtracking generation already works well. Of course, we hope that these strategies also make testing effective on programs where plain backtracking search fails. To address this, we consider our four allocators and the position-independent doubly-linked and circular linked lists—see Table 3.

The clearest difference is on the free-list allocators. With *Everything Off*, Lucas can't even generate a single precondition-satisfying input. As a result, it is unable to perform any testing of the 115 total tasks. In stark contrast, *Everything On* generates valid inputs for every task and solves all 24 tasks for IFL, 32 of 36 for IFLC, 17 of 24 for EFL, and 13 of 31 for EFLC. Thus, the refinement strategies do not merely decrease the number of backtracks or improve the speed of generation. They make testing possible where Bennet-style generation completely fails.

The allocator specifications are particularly challenging for naive generation because there are many levels of constraints. For IFL, each randomly generated header determines the size of the payload, which impacts the location of the following header, so the values generated must fit within the allocation and partition the buffer. IFLC adds a footer to every block that must agree with its header, adding more constrained values. The addition of the footer also increases the alignment required for each block, making each block's alignment constraint less likely to be satisfied. The explicit variants store pointers inside each free block for faster traversal of the free list. Together,

Table 3. Bold marks the lowest mean and highest solved count per workload, except ties. S/U/UN denotes solved in every trial, unsolved, and unsatisfied (no valid input generated in some trial).

Workload	Configuration	Backtracks	Discards	Inputs	Time (s)	S/U/UN
IFL	Everything On	**18.7k ± 11.5k**	**1.2k ± 605.4**	**179.8 ± 100.1**	**1.47 ± 0.65**	**24/0/0**
	Everything Off	—	—	—	—	0/0/24
IFLC	Everything On	**20.6k ± 4.9k**	**791.3 ± 189.4**	**42.8 ± 8.2**	**0.91 ± 0.06**	**32/4/0**
	Everything Off	—	—	—	—	0/0/36
EFL	Everything On	**89.4k ± 34.7k**	**2.5k ± 786.5**	**259.1 ± 116.4**	**4.40 ± 1.95**	**17/7/0**
	Everything Off	—	—	—	—	0/0/24
EFLC	Everything On	**54.7k ± 21.1k**	**1.6k ± 681.0**	**44.4 ± 19.5**	**5.13 ± 2.15**	**13/18/0**
	Everything Off	—	—	—	—	0/0/31
PI-DLL	Everything On	**14.0k ± 3.2k**	**11.1 ± 2.6**	**1.4 ± 0.3**	**0.47 ± 0.01**	**18/0/0**
	Everything Off	6.8M ± 4.6M	18.6k ± 12.5k	1.9k ± 1.3k	6.17 ± 3.96	10/8/0
PI-CLL	Everything On	**56.4k ± 19.3k**	**5.6 ± 1.4**	**1.0 ± 0.0**	1.45 ± 0.35	**25/2/0**
	Everything Off	1.1M ± 296.2k	170.4 ± 39.0	1.2 ± 0.1	**0.92 ± 0.16**	9/18/0

these pointers must form a consistent doubly-linked free list, adding even more constraints. Finally, EFLC combines both the boundary-tag and explicit-list constraints. At this point our refinement strategies begin to struggle. (Even so, Lucas was able to find bugs for some of the unsolved tasks in some of the trials, suggesting that it would perform better with only a modestly larger timeout.)

A similar pattern appears in the two position-independent linked list examples. Their nodes refer to one another using offsets into a single buffer, which couples the integer value of each offset with allocation bounds and ownership. One thing that makes them simpler than the allocators is that each node in the list has a fixed size, whereas the allocators have payloads of various sizes. However, the additional structure of PI-DLL and PI-CLL makes them more difficult than the other PI case studies, with *Everything Off* solving 10/18 and 9/27 tasks, respectively. In comparison, *Everything On* solves all 18 PI-DLL tasks and 25 of 27 PI-CLL tasks. The new refinement strategies also reduce backtracks and discards by two to three orders of magnitude for PI-DLL and by more than an order of magnitude for PI-CLL.

The allocator workloads still average between 18.7k and 89.4k backtracks per solved task with *Everything On*, showing that generation does not become trivial after overcoming a few constraints. Instead, each backtrack lets the strategies use information from failed constraints to redirect search across several variables. The refinement strategies thereby enable testing where Bennet-style naive generation is ineffective and substantially improve bug-finding on difficult position-independent structures.

5.5 RQ3: Is Each Option Essential?

Having established that using all of our strategies together provides new bug-finding abilities, we should also ask whether all of them are necessary or whether some could be dropped. To answer this, we take our hardest case studies, the allocators, individually ablate the strategies, and disable the non-allocation domains as a group rather than individually.

Since *Everything Off* fails to generate inputs for any of these tasks, we use *Everything On* as our baseline. We individually disable cascading propagation and corrective refinement, replace the propagating speculative refinement with its non-propagating variant, and disable speculative refinement entirely. We also disable the non-allocation domains. The results of these ablations

Table 4. *Without X* disables *X* compared to *Everything On*, *Speculative (Not Prop)* uses plain speculation, and *Only Allocation Domain* retains only the allocation domain. Bold solved counts are below *Everything On*, and underlined counts are above it. S/U/UN denotes solved in every trial, unsolved, and unsatisfied (no valid input generated in some trial).

Workload	Configuration	Backtracks	Discards	Inputs	Time (s)	S/U/UN
IFL	Everything On	4.5k ± 2.0k	293.2 ± 148.2	63.7 ± 32.3	0.66 ± 0.05	24/0/0
	Without Cascading	32.1k ± 13.3k	295.2 ± 136.1	62.0 ± 30.6	1.11 ± 0.36	**22**/2/0
	Without Speculative	3.8k ± 1.5k	465.5 ± 194.4	85.9 ± 36.4	0.62 ± 0.03	24/0/0
	Speculative (Not Prop)	2.5k ± 907.9	279.0 ± 103.1	60.5 ± 22.2	0.62 ± 0.02	**22**/2/0
	Without Corrective	140.4k ± 31.0k	15.9 ± 3.7	47.2 ± 10.0	0.66 ± 0.02	**22**/2/0
	Only Allocation Domain	729.8k ± 136.2k	273.8 ± 68.7	280.2 ± 54.1	2.27 ± 0.34	**23**/1/0
IFLC	Everything On	15.5k ± 4.3k	594.0 ± 164.1	38.3 ± 8.3	0.84 ± 0.05	32/4/0
	Without Cascading	62.8k ± 13.7k	503.9 ± 111.6	34.8 ± 7.6	1.18 ± 0.12	32/4/0
	Without Speculative	13.5k ± 2.8k	452.4 ± 92.1	30.0 ± 6.8	0.89 ± 0.05	**29**/7/0
	Speculative (Not Prop)	14.2k ± 3.8k	547.9 ± 145.6	35.1 ± 7.8	0.83 ± 0.05	32/4/0
	Without Corrective	164.2k ± 49.2k	911.5 ± 272.9	131.2 ± 39.4	0.85 ± 0.06	**29**/7/0
	Only Allocation Domain	3.0M ± 1.1M	26.2k ± 9.2k	1.2k ± 416.0	7.42 ± 2.39	**27**/9/0
EFL	Everything On	6.6k ± 4.9k	218.1 ± 161.1	23.9 ± 19.4	0.82 ± 0.10	17/7/0
	Without Cascading	103.0k ± 98.4k	2.3k ± 2.2k	197.0 ± 191.3	5.76 ± 5.01	**11**/13/0
	Without Speculative	213.0k ± 165.5k	641.7 ± 511.5	42.5 ± 35.7	12.05 ± 9.43	**5**/19/0
	Speculative (Not Prop)	—	—	—	—	**0**/0/24
	Without Corrective	214.6k ± 213.8k	1.9k ± 1.9k	950.2 ± 946.3	1.94 ± 1.28	17/7/0
	Only Allocation Domain	—	—	—	—	**0**/0/24
EFLC	Everything On	—	—	—	—	13/18/0
	Without Cascading	—	—	—	—	**7**/24/0
	Without Speculative	—	—	—	—	**6**/25/0
	Speculative (Not Prop)	—	—	—	—	**0**/0/31
	Without Corrective	—	—	—	—	24/7/0
	Only Allocation Domain	—	—	—	—	**0**/0/31

are presented in Table 4. Since the set of tasks solved by every configuration differs from that of Table 3, the reported means are not comparable across the two tables.

We find that no single ablation maintains *Everything On*'s effectiveness across all the allocators. Removing cascading propagation costs solved tasks on IFL, EFL, and EFLC, and removing speculative refinement costs tasks on IFLC, EFL, and EFLC. Replacing propagating speculation with its non-propagating variant is more damaging still, with EFL and EFLC becoming unsatisfied and IFL losing two solved tasks. This is worse than no speculation at all, since removing speculative refinement entirely still solves five and six tasks, respectively. The reason is that speculative refinement can discover bottom—an empty abstract value, indicating that no concrete value can satisfy the accumulated constraints—before sampling, but the non-propagating variant reports that failure with only `SpecialBasis.top` for the blamed variables. Without speculative refinement, the same inconsistency is discovered later by corrective refinement, whose more detailed refinement information can guide subsequent attempts. Corrective refinement is important in general, despite its cost: removing it reduces the solved count on IFL and IFLC and increases IFL's mean backtracks from 4.5k to 140.4k—over thirty times the wasted search to find the same bugs.

Support for domains beyond allocation is again useful for the hardest preconditions. With only the allocation domain, EFL and EFLC are entirely unsatisfied, and the solved counts for IFL and IFLC fall from 24 to 23 and from 32 to 27, respectively. The combination of wrapped intervals,

congruences, and tristate numbers tracks constraints on sizes, alignments, and bit-level patterns that the allocation domain alone cannot represent, demonstrating the value of generalizing the original allocation-specific approach to additional abstract domains. This experiment does not attempt to isolate the contribution of each individual domain.

EFLC exposes a tradeoff rather than a universally best configuration. The full configuration solves 13 tasks and removing corrective refinement solves 24, yet neither dominates, as the 13 are not a subset of the 24. Since no task is solved by every configuration, the table shows no means for EFLC. Without corrective refinement, generation is cheaper and finds more bugs in simpler functions within the two-minute limit, but for `free` it very often produces the trivial **`NULL`**, leaving `free`'s bugs undetected. The full configuration tests `free` at the cost of slower generation elsewhere. Every option thus improves bug-finding somewhere but can reduce throughput elsewhere, which favors enabling refinement only where naive generation cannot satisfy preconditions and managing configurations spec-by-spec. We lay out a possible selection strategy and its automation in Section 7.

6 Related Work

The key innovation in Lucas is using abstract domains to refine the sampling space of a random generator at runtime. We discuss related work from constraint solving, abstract interpretation, and test generation.

6.1 Constraint Solving and Propagation

Constraint propagation with abstract domains. There has been much prior work on performing constraint propagation with intervals [21] and other abstract domains [8, 34, 37, 55]. Our work is related in that we use refinements of abstract elements to propagate constraints. However, we do it within our generator language, with our refinement strategies controlling when it occurs, rather than propagating blindly. We also rarely save the information propagated, so as to avoid decreasing the diversity of inputs generated.

Constraint-based test generation. Euclide [28] generates tests using a hybrid of finite-domain propagation and dynamic linear relaxations, alternating local filtering with simplex calls that tighten as domains shrink. FocalTest [12, 13] uses a translation to constraint logic programming over finite domains, where propagation is used to prune the search space, aiming to generate test inputs for coverage. Lucas shares the high-level goal of propagation-guided generation but only performs it in certain circumstances, defined by our refinement strategies. We also use abstract domains, reducing the amount of information propagated in exchange for speed.

6.2 Abstract Interpretation

Abstract interpretation [18] classically overapproximates reachable states statically. We instead use backward abstract transformers at runtime to refine the values from which a generator samples. Overapproximation may produce discarded samples but never excludes valid values.

We implement the non-relational domains of wrapped intervals [25], congruences [29], and tristate numbers [20, 22, 56]. Relational domains such as octagons [39] and polyhedra [19] can capture inter-variable constraints but are more expensive.

Lucas's per-variable refinement and blame information fits a non-relational map domain. Relational domains would require new semantics and substantial runtime changes, while a sufficiently expressive one approaches the cost and complexity of a constraint solver.

6.3 Techniques for Satisfying Preconditions

Narrowing and lazy generation. Needed narrowing and related approaches [5, 26, 54] lazily instantiate values only as needed. Lazy SmallCheck [50] uses lazy enumeration for exhaustive bounded

testing, and Claessen et al. [15, 16] adapts this technique to random testing. Adding such laziness to Lucas would be useful. However, the current work would still be necessary, since laziness only reduces how far backtracking traverses—one would still need a way to sample values satisfying the constraints encountered.

Generator synthesis and constrained generation. Existing work does not target low-level languages where resource constraints must be satisfied. Lampropoulos et al. [32] derive generators from inductively defined properties in Rocq, which is a much more restricted language. Palamedes [27] uses deductive synthesis in Lean. These approaches synthesize or define generators that satisfy logical constraints. Lucas has to consider a more difficult problem, which intertwines those constraints with resource constraints, having to reason about pointer arithmetic and ownership.

Separation logic testing. In Java StarFinder and Concolic StarFinder, Pham et al. [44, 45, 46] used a decidable fragment of separation logic along with a solver, in order to generate satisfying inputs for Java programs. We support a larger fragment of separation logic and have to deal with more complex pointer arithmetic, due to targeting C.

Chou [14] generates stateful sequences of calls to CN-specified functions, using Fulminate as a test oracle. If a precondition failure occurs, it's an invalid sequence, and if a postcondition failure occurs, it's a bug. However, it requires a constructor or initialization function to begin the sequence, whereas Lucas does not.

A recent tool, Darcy [2], uses an alternative strategy—an SMT translation—to generate test inputs. Lucas is intended as a potential replacement for Bennet, with the ability to enable or disable the refinement strategies, while Darcy is not. For example, for the ring queue in Section 5, generating 100 inputs for the `push` function takes Bennet and Lucas about 3 seconds, whereas Darcy takes 3 to 3.5 hours, due to the use of a recursive function in the specification. However, it is able to support very complex preconditions by relying on a solver.

7 Limitations and Future Work

By default, only the allocation domain is enabled, since it is necessary for generating valid pointers. Every other domain and refinement strategy is opt-in, selected with CLI flags. The heuristics below give a workable configuration, not necessarily an optimal one.

A conservative approach starts with *Everything Off* and refines specification by specification when Lucas fails to generate valid inputs. Invalid inputs are not the only reason to refine, though: naive generation may produce valid inputs yet explore them ineffectively, as on PI-DLL and PI-CLL, where *Everything Off* solves far fewer tasks than *Everything On*.

To guide these choices, Lucas reports the constraints responsible for the most backtracking and those rarely satisfied within a test attempt—the two major sources of generation failure we found in Bennet [3] and Lucas without refinement strategies. A constraint's syntactic form then suggests a domain: inequalities suggest wrapped intervals, remainder constraints suggest congruences, and bitwise operations suggest tristate numbers. The refinement strategy depends on where the constrained value comes from. We enable speculative refinement for a value generated locally and corrective refinement for a generated argument, so failure information carries back to its generation site. We choose cascading propagation empirically, enabling it for a case study and keeping it when it improves generation.

These rules are only starting points. Domain selection, speculative refinement, and corrective refinement could be automated into per-constraint or per-variable suggestions, though we have no such rule for cascading propagation. Finer granularity could also help, such as converting some **`assert`** to **`assert_domain`**, but not all.

Acknowledgments

We thank our reviewers for their useful feedback. Our work was supported by the NSF under grant *SHF: Medium: Usable Property-Based Testing, NSF #2402449*. Claude Code was used in the development of Lucas, to generate the scripts that render our experimental data as the LaTeX tables presented here, and to style the heap layout diagram in Section 2.

Data Availability Statement

An artifact is available on Zenodo [4], and the up-to-date version of Lucas is publicly available as part of the CN toolchain [1]. The artifact includes Lucas and an extension of Etna [51] with all the case studies described here, along with scripts for running the experiments and constructing the tables.

References

[1] 2025. CN. https://github.com/rems-project/cn

[2] Zain K Aamer, Rini Banerjee, Hiroyuki Katsura, David Kaloper-Meršinjak, Dimitrios J. Economou, Kayvan Memarian, Dhruv Makwana, Neel Krishnaswami, Benjamin C. Pierce, Christopher Pulte, and Peter Sewell. 2026. Code-Specify-Test-Debug-Prove: Flexibly Integrating Separation Logic Specification into Conventional Workflows. *Proc. ACM Program. Lang.* 10, PLDI, Article 200 (June 2026), 27 pages. doi:10.1145/3808278

[3] Zain K Aamer and Benjamin C. Pierce. 2025. Bennet: Randomized Specification Testing for Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 413 (Oct. 2025), 30 pages. doi:10.1145/3764115

[4] Zain K Aamer and Benjamin C. Pierce. 2026. *Artifact for "Random Testing via Runtime Abstract Interpretation"*. doi:10.5281/zenodo.21515133

[5] Sergio Antoy, Rachid Echahed, and Michael Hanus. 2000. A Needed Narrowing Strategy. *J. ACM* 47, 4 (July 2000), 776–822. doi:10.1145/347476.347484

[6] Andrew W. Appel. 2012. Verified Software Toolchain. In *NASA Formal Methods*, Alwyn E. Goodloe, Suzette Person, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum (Eds.). Vol. 7226. Springer Berlin Heidelberg, Berlin, Heidelberg, 2–2. doi:10.1007/978-3-642-28891-3_2

[7] Rini Banerjee, Kayvan Memarian, Dhruv Makwana, Christopher Pulte, Neel Krishnaswami, and Peter Sewell. 2025. Fulminate: Testing CN Separation-Logic Specifications in C. *Proc. ACM Program. Lang.* 9, POPL, Article 43 (Jan. 2025), 33 pages. doi:10.1145/3704879

[8] Sébastien Bardin, Philippe Herrmann, and Florian Perroud. 2010. An Alternative to SAT-Based Approaches for Bit-Vectors. In *Tools and Algorithms for the Construction and Analysis of Systems*, Javier Esparza and Rupak Majumdar (Eds.). Springer, Berlin, Heidelberg, 84–98. doi:10.1007/978-3-642-12002-2_7

[9] Lukas Bulwahn. 2012. The New Quickcheck for Isabelle. In *Certified Programs and Proofs*, Chris Hawblitzel and Dale Miller (Eds.). Springer, Berlin, Heidelberg, 92–108. doi:10.1007/978-3-642-35308-6_10

[10] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, Berlin, Heidelberg, 459–465. doi:10.1007/978-3-642-20398-5_33

[11] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *Journal of Automated Reasoning* 61, 1-4 (June 2018), 367–422. doi:10.1007/s10817-018-9457-5

[12] Mathieu Carlier, Catherine Dubois, and Arnaud Gotlieb. 2010. Constraint Reasoning in FocalTest. In *ICSOFT*. https://inria.hal.science/hal-00699233

[13] Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. 2013. FocalTest: A Constraint Programming Approach for Property-Based Testing. In *Software and Data Technologies*, José Cordeiro, Maria Virvou, and Boris Shishkov (Eds.). Springer, Berlin, Heidelberg, 140–155. doi:10.1007/978-3-642-29578-2_9

[14] Ethan Khan Chou. 2025. *Model-Free Stateful Random Testing*. M.S. University of Maryland, College Park, United States – Maryland. https://www.proquest.com/docview/3297923714/abstract/E7CE0F7F576B400APQ/1

[15] Koen Claessen, Jonas Duregård, and Michał H. Pałka. 2014. Generating Constrained Random Data with Uniform Distribution. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 18–34. doi:10.1007/978-3-319-07151-0_2

[16] Koen Claessen, Jonas Duregård, and Michał H. Pałka. 2015. Generating constrained random data with uniform distribution. *Journal of Functional Programming* 25 (2015), e8. doi:10.1017/S0956796815000143

[17] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. doi:10.1145/351240.351266

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

[19] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Tucson, Arizona) *(POPL '78)*. Association for Computing Machinery, New York, NY, USA, 84–96. doi:10.1145/512760.512770

[20] Edward Cree, Yonghong Song, Daniel Borkmann, John Fastabend, Harishankar Vishwanathan, Andrii Nakryiko, Song Liu, Paul Chaignon, and Nandakumar Edamana. 2026. *tnum.c*. https://github.com/torvalds/linux/blob/v6.19/kernel/bpf/tnum.c Linux kernel source file `kernel/bpf/tnum.c` at tag `v6.19`. Authors ordered by first appearance in the visible GitHub commit history for this file from oldest to newest, excluding KAGA-KOKO because that change only added a license identifier..

[21] Ernest Davis. 1987. Constraint Propagation with Interval Labels. *Artificial Intelligence* 32, 3 (July 1987), 281–331. doi:10.1016/0004-3702(87)90091-9

[22] Guangsheng Fan, Liqian Chen, Banghu Yin, Wenyu Zhang, Peisen Yao, and Ji Wang. 2025. Program Analysis Combining Generalized Bit-Level and Word-Level Abstractions. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA030 (June 2025), 23 pages. doi:10.1145/3728905

[23] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, Part i: A Multi-Language Platform for Symbolic Execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 927–942. doi:10.1145/3385412.3386014

[24] Galois, Inc. 2025. VERSE-OpenSUT. Github. https://github.com/GaloisInc/VERSE-OpenSUT

[25] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2015. Interval Analysis and Machine Arithmetic: Why Signedness Ignorance Is Bliss. *ACM Trans. Program. Lang. Syst.* 37, 1 (Jan. 2015), 1:1–1:35. doi:10.1145/2651360

[26] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 225–234. doi:10.1145/1806799.1806835

[27] Harrison Goldstein, Hila Peleg, Cassia Torczon, Daniel Sainati, Leonidas Lampropoulos, and Benjamin C. Pierce. 2025. The Search for Constrained Random Generators. arXiv:2511.12253 [cs.PL] https://arxiv.org/abs/2511.12253

[28] Arnaud Gotlieb. 2009. Euclide: A Constraint-Based Testing Framework for Critical C Programs. In *2009 International Conference on Software Testing Verification and Validation*. 151–160. doi:10.1109/ICST.2009.10

[29] Philippe Granger. 1989. Static Analysis of Arithmetical Congruences. *International Journal of Computer Mathematics* 30, 3-4 (Jan. 1989), 165–190. doi:10.1080/00207168908803778

[30] John Hughes, Rini Banerjee, and Benjamin C. Pierce. 2024. Adventures in Specification-Based Testing. Invited talk at Isaac Newton Institute Workshop on Big Specification: Specification, Proof, and Testing at Scale.

[31] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods (Lecture Notes in Computer Science)*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, Berlin, Heidelberg, 41–55. doi:10.1007/978-3-642-20398-5_4

[32] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2018. Generating Good Generators for Inductive Relations. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018), 1–30. doi:10.1145/3158133

[33] Leonidas Lampropoulos and Benjamin C. Pierce. 2018. *QuickChick: Property-Based Testing in Coq*. https://softwarefoundations.cis.upenn.edu/qc-current/ Electronic textbook.

[34] Michel Leconte and Bruno Berstel. 2007. Extending a CP Solver with Congruences as Domains for Program Verification. In *Trends in Constraint Programming*. John Wiley & Sons, Ltd, Chapter 21, 337–344. doi:10.1002/9780470612309.ch21

[35] Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 827–850. doi:10.1007/978-3-030-81688-9_38

[36] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Trans. Model. Comput. Simul.* 8, 1 (Jan. 1998), 3–30. doi:10.1145/272991.272995

[37] Laurent D. Michel and Pascal Van Hentenryck. 2012. Constraint Satisfaction over Bit-Vectors. In *Principles and Practice of Constraint Programming*, Michela Milano (Ed.). Springer, Berlin, Heidelberg, 527–543. doi:10.1007/978-3-642-33558-

7_39
[38] Antoine Miné. 2004. *Weakly Relational Numerical Abstract Domains.* Theses. Ecole Polytechnique X. https://pastel.hal.science/tel-00136630
[39] Antoine Miné. 2006. The octagon abstract domain. *Higher-Order and Symbolic Computation* 19, 1 (01 Mar 2006), 31–100. doi:10.1007/s10990-006-8609-1
[40] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Vol. 9583. Springer Berlin Heidelberg, Berlin, Heidelberg, 41–62. doi:10.1007/978-3-662-49122-5_2
[41] Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2012. Signedness-Agnostic Program Analysis: Precise Integer Bounds for Low-Level Code. In *Programming Languages and Systems*. Springer, Berlin, Heidelberg, 115–130. doi:10.1007/978-3-642-35182-2_9
[42] Rickard Nilsson. 2024. ScalaCheck. https://scalacheck.org/.
[43] Takuji Nishimura. 2000. Tables of 64-Bit Mersenne Twisters. *ACM Trans. Model. Comput. Simul.* 10, 4 (Oct. 2000), 348–357. doi:10.1145/369534.369540
[44] Long H. Pham, Quang Loc Le, Quoc-Sang Phan, and Jun Sun. 2019. Concolic Testing Heap-Manipulating Programs. In *Formal Methods – The Next 30 Years*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer International Publishing, Cham, 442–461. doi:10.1007/978-3-030-30942-8_27
[45] Long H. Pham, Quang Loc Le, Quoc-Sang Phan, Jun Sun, and Shengchao Qin. 2018. Testing Heap-Based Programs with Java StarFinder. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 268–269. doi:10.1145/3183440.3194964
[46] Long H. Pham, Quang Loc Le, Quoc-Sang Phan, Jun Sun, and Shengchao Qin. 2019. Enhancing Symbolic Execution of Heap-Based Programs with Separation Logic for Test Input Generation. In *Automated Technology for Verification and Analysis*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.). Springer International Publishing, Cham, 209–227. doi:10.1007/978-3-030-31784-3_12
[47] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying Systems C Code with Separation-Logic Refinement Types. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 1:1–1:32. doi:10.1145/3571194
[48] Christopher Pulte, Benjamin C. Pierce, Cole Schlesinger, and Elizabeth Austell. 2024. CN tutorial. https://rems-project.github.io/cn-tutorial/. [Online; accessed 26-October-2024].
[49] J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. doi:10.1109/LICS.2002.1029817 ISSN: 1043-6871.
[50] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell (Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 37–48. doi:10.1145/1411286.1411292
[51] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C. Pierce, and Leonidas Lampropoulos. 2023. Etna: An Evaluation Platform for Property-Based Testing (Experience Report). *Proceedings of the ACM on Programming Languages* 7, ICFP (Aug. 2023), 878–894. doi:10.1145/3607860
[52] Guy L. Steele, Doug Lea, and Christine H. Flood. 2014. Fast splittable pseudorandom number generators. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 453–472. doi:10.1145/2660193.2660195
[53] Stephen Dolan and Mindy Preston. 2017. Testing with Crowbar. https://github.com/ocaml/ocaml.org-media/blob/master/meetings/ocaml/2017/extended-abstract__2017__stephen-dolan_mindy-preston__testing-with-crowbar.pdf
[54] Andrew Tolmach and Sergio Antoy. 2003. A Monadic Semantics for Core Curry. *Electronic Notes in Theoretical Computer Science* 86, 3 (2003), 16–34. doi:10.1016/S1571-0661(04)80691-1 WFLP 2003, 12th International Workshop on Functional and Constraint Logic Programming (in connection with RDP'03, Federated Conference on Rewriting, Deduction and Programming).
[55] Charlotte Truchet, Marie Pelleau, and Frederic Benhamou. 2010. Abstract Domains for Constraint Programming, with the Example of Octagons. In *2010 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 72–79. doi:10.1109/SYNASC.2010.69
[56] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2022. Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 254–265. doi:10.1109/CGO53902.2022.9741267
[57] Scott Vokes. 2019. theft: property-based testing for C. https://github.com/silentbicycle/theft
[58] Litao Zhou, Jianxing Qin, Qinshi Wang, Andrew W. Appel, and Qinxiang Cao. 2024. VST-A: A Foundationally Sound Annotation Verifier. *Proc. ACM Program. Lang.* 8, POPL, Article 69 (Jan. 2024), 30 pages. doi:10.1145/3632911

Received 2026-03-17; accepted 2026-08-06