Bennet: Randomized Specification Testing for Heap-Manipulating Programs

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

Property-based testing (PBT), widely used in functional languages and interactive theorem provers, works by randomly generating many inputs to a system under test. While PBT has also seen some use in low-level languages like C, users in this setting must craft all their own generators by hand, rather than letting the tool synthesize most generators automatically from types or logical specifications. For low-level code with complex memory ownership patterns, writing such generators can waste significant amounts of time.

CN, a specification and verification framework for C, features a streamlined presentation of separation logic that is specially tuned to present only "easy" logical problems to an underlying constraint solver. Prior work on the Fulminate testing framework has shown that CN's streamlined specifications can also be checked effectively at run time, providing an oracle for testing whether a memory state satisfies a pre- or postcondition.

We show that the restricted syntax of CN is also a good basis for deriving *generators* for random inputs satisfying separation-logic preconditions. We formalize the semantics for a DSL describing these generators, as well as optimizations that reorder when values are generated and propagate arithmetic constraints. Using this DSL, we implement a property-based testing tool, Bennet, that generates and runs random tests for C functions annotated with CN specifications. We evaluate Bennet on a corpus of programs with CN specifications and show that it can efficiently generate bug-revealing inputs for heap-manipulating programs with complex preconditions.

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

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

1 Introduction

Property-based testing (PBT), popularized by QuickCheck [24], allows users to test a piece of software against an executable specification on a large number of randomly generated inputs. Initially developed for Haskell, PBT has now been applied to many other languages, including Python [54], Erlang [7, 67], Scala [62], OCaml [84], and many others—even to low-level languages like C [86]. It has also been applied to proof assistants [12, 45, 64] to assist formal verification.

PBT relies on *generators* for random inputs to a system under test. Many times this has to be manually written, but we would like to be able to synthesize these from our specifications automatically. To achieve this, we need a specification language that (1) can capture complex memory ownership patterns, (2) is reasonably quick to check at runtime, and (3) facilitates automatic generation of valid inputs. We argue that (a particular presentation of) *separation logic* fits the bill.

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

ACM 2475-1421/2025/10-ART413
https://doi.org/10.1145/3764115

Separation logic [78] is an important tool for describing the complex memory ownership of heap-manipulating programs. It has been used by a number of verification tools, including Infer [16], VST [5, 17, 89], VeriFast [40], Viper [61], Gillian [27, 55], and CN itself [74]. While separation logic is an ideal method of describing valid heaps, it has rarely been used for testing. One reason might be the difficulty in checking whether a memory state satisfies a given separation logic formula, which is EXPTIME in the general case [11]. Various automatic verification tools, like VeriFast [40] and Gillian [27, 55] have applied input/output mode checks to restrict formulas to a fragment that is more amenable to proof automation. Instead of these mode checks, CN uses a restricted syntax that forces specifications to be written in a style that always satisfies these mode checks.

Banerjee et al. [9] showed that CN's syntax is also amenable to efficient runtime checking. Their instrumentation tool, Fulminate, inserts checks into functions with CN specifications, alerting the user when a precondition or postcondition has been violated on a given program run. If the function under test is given an input satisfying its precondition, Fulminate can detect whether the function's result satisfies the desired postcondition. The motivating insight of the present paper is that CN's syntax for predicates and specifications is *also* useful for deriving randomized generators for test inputs satisfying the function's precondition.

We define a transformation from the syntax of CN specifications to a closely corresponding generator DSL. We define a DSL that makes instantiation—and the location to backtrack to—explicit for each argument, global, and memory value generated, and we formalize it with a monadic semantics. We can synthesize generators, from expressions in the DSL, that perform backtracking search, attempting to regenerate pieces of the input when they cannot satisfy the precondition.

Our generator synthesis algorithm is structured like an optimizing compiler, with two key optimization passes. First, we *lift constraints*, where a generator for values satisfying `P||Q` becomes two generators, one for values satisfying `P` and the other for values satisfying `Q`, plus a random choice between the two. Second, we conservatively *reorder when variables are generated*, to backtrack earlier if necessary and decrease non-determinism. (For example, consider the numerical constraint `x == y + z`. If `x` is generated after `y` and `z`, generation is simply assigning the value of `y + z` to `x`. If `x` is generated first, we would either have to rewrite the constraint to calculate `y` or `z`, or else randomly generate values and backtrack.) We implement the generator synthesis algorithm in a tool, dubbed Bennet, that compiles our DSL into C, allowing us to generate test cases for CN programs and use Fulminate to check for specification violations.

We evaluate the bug-finding ability of Bennet on various case studies using an extension of the Etna benchmarking framework [83]. We compare Bennet against AFL++ [26], a popular fuzzing tool. We show the generality of our method by applying it to a large collection of examples from the CN tutorial [75] and showing that it can generate precondition-satisfying inputs for all but one of the current examples. To evaluate Bennet's effectiveness in practice, we carry out bug-finding studies on several more difficult examples, including complex data structures (binary search trees and AVL trees), a resource management simulation, and a system for managing access to secret keys. To assess the importance of the lifting and reordering optimizations, we perform an ablation study on several of the most challenging case studies.

In summary, our main contributions are:

- We demonstrate that CN's restricted syntax, already known to be good for proof automation and runtime checking, is also useful for deriving random generators for complex heap shapes (Section 2).
- We formalize a generator DSL that echoes the structure of the CN specification language and give it a monadic semantics (Section 3). We define optimizations on this DSL that propagate

constraints and reorder when values are generated to reduce the likelihood of backtracking, significantly improving generation speed.
- We implement our DSL by compiling it to C, encoding complex control flow into relatively simple and efficient C code (Section 4).
- We evaluate the bug-finding effectiveness of our generators on several case studies, including data structures (BSTs and AVL trees) and small examples of critical systems code (an airport simulation and a real-world key management system), showing them outperforming AFL++ (Section 5). We also demonstrate that our optimization passes give significant benefits via an ablation study.

We close by discussing related work (Section 6) and limitations and future directions (Section 7).

2 Overview

We introduce CN specification language by example, review how it can be used to check whether a heap is valid [9], and sketch an alternative interpretation of CN statements as generators of arguments and heaps.

2.1 Checking Heaps Against Specifications

We begin by showing how to check if a specific heap satisfies a logical precondition or postcondition.

Suppose we have a simple function that takes a pointer to an integer and doubles its value in-place.

```
void times2(int *p) {
    int n = *p;
    int m = n + n;
    *p = m; }
```

Since signed overflow/underflow are undefined behavior in C, we want to ensure that p is a valid `int` pointer and that n doubled does not overflow/underflow. In most program logics, a precondition for this function might look like this:

$$\exists \mathsf{n}.\ \mathsf{p} \mapsto \mathsf{n}\ \wedge\ \mathsf{INT32\_MIN} \leq \mathsf{n} + \mathsf{n} \leq \mathsf{INT32\_MAX}$$

In English, this says that there exists an integer n at the location pointed to by p, where $n + n$ is a valid 32-bit integer. Given a concrete heap and asked if it is a model of this formula, one could simply look at the value at p in the heap to get the value of n. This lookup requires no backtracking, but checking that existentially quantified formulas are satisfied can require backtracking in general. (E.g., to check whether a given heap satisfies the formula $\exists \mathsf{p}.\ \mathsf{p} \mapsto 42$, we must examine every location in the heap.) To support efficient validation of formulas against heaps, we would like to restrict attention to those that can be validated without backtracking. Some verification tools use semantic checks such as mode analysis for this purpose. CN uses syntactic restrictions.

Suppose we rewrite the body of the `times2` precondition to use a predicate, `canDoublePointee`,

$$\mathsf{canDoublePointee}(\mathsf{p},\mathsf{n}) =\ \mathsf{p} \mapsto \mathsf{n} \wedge \mathsf{INT32\_MIN} \leq \mathsf{n} + \mathsf{n} \leq \mathsf{INT32\_MAX}$$

and rewrite the precondition itself as $\exists n.\ \mathsf{canDoublePointee}(\mathsf{p}, n)$. The type of `canDoublePointee` is $\mathsf{Loc} \to \mathbb{Z} \to \mathsf{Heap} \to \mathbb{B}$, taking a memory location and an integer, and returning a proposition on heaps ($\mathsf{Heap} \to \mathbb{B}$). Rearranging, we can write the type as $\mathsf{Heap} \to \mathsf{Loc} \to \mathbb{Z} \to \mathbb{B}$.

The key insight into CN's syntax is that, for all types A, the set of functions of type $A \to \mathbb{B}$ is isomorphic to the powerset of A. A function f of type $\mathsf{A} \to \mathbb{B}$ corresponds to the subset $\mathsf{A}' = \{\mathsf{a} \in \mathsf{A} \mid \mathsf{f}(\mathsf{a})\}$ of satisfying inputs to f.

Assuming that heaps store only integers, consider the type of the points-to relation, $\mapsto$, namely $\mathsf{Heap} \to \mathsf{Loc} \to \mathbb{Z} \to \mathbb{B}$. By the above isomorphism, this type is equivalent to $\mathsf{Heap} \to \mathsf{Loc} \to \mathcal{P}(\mathbb{Z})$. But, since there can only be a single value at a given location in a heap (or none at all), the type is really $\mathsf{Heap} \to \mathsf{Loc} \to \mathsf{Option}\ \mathbb{Z}$, i.e., the type of functions from heaps and locations into values in the $\mathsf{Option}\ \mathbb{Z}$ monad.

Going further, suppose we want to check that some heap satisfies a separation-logic assertion $P * Q$. This assertion can be read as "This heap consists of two disjoint parts, one satisfying p and one satisfying q"—that is, to check naively that a given heap satisfies it, we would need to guess a partition of the heap into two disjoint parts and recursively check that one satisfies p and the other q. There are exponentially many ways to make this split, so clearly we need a better idea to make checking efficient.

Suppose we remove our initial Heap argument and replace our option monad (Option $\mathbb{Z}$) with a state-option monad (MarkedHeap $\rightarrow$ Option ($\mathbb{Z}$, MarkedHeap)). That is, instead of expecting a singleton heap in our points-to function, we take a heap of any size and mark the given location, if present in the domain of the heap and previously unmarked, as "owned," returning the updated heap along with the value read from the given location. To guarantee unique ownership, we return None if a location is already marked. This new points-to function (of type Loc $\rightarrow$ (MarkedHeap $\rightarrow$ Option ($\mathbb{Z}$, MarkedHeap))) captures the intuition of CN's unique-ownership primitive, **Owned**.

Returning to our example, CN's syntax allows canDoublePointee to be written like this:

```
predicate integer canDoublePointee(pointer p) {
  take n = Owned(p);
  let r = n + n;
  MINi32() <= r; r <= MAXi32();
  return n; }
```

It assigns the return value of **Owned** to n using the **take** construct, which is semantically a monadic bind—that is, it passes along the MarkedHeap state resulting from the body of the **take** to the remainder of the predicate, aborting the remainder with None if the body returns None). This captures the guarantees of a separating conjunction within CN. The **take** is followed by an ordinary **let**, two logical constraints, and a monadic return, which will return Some(n) if evaluation reaches that point. The logical constraints are simply evaluated, returning None if they are false.

```
requires
  take n = canDoublePointee(p);
ensures
  take m = Owned<int>(p);
  m == n + n;
```

In the full specification for times2, the precondition says that the caller of times2 must provide a pointer p and a heap such that evaluating canDoublePointee(p) on this heap yields Some(n) for some n. The postcondition then promises that evaluating canDoublePointee(p) in the heap at the end of the call to times2 will yield Some(m), where $m = n + n$.

For a more interesting example, let's define an inductive predicate for lists of positive integers,

```
predicate (datatype int_list) PosIntList(pointer p) {
  if (p == NULL) { return (Nil {}); }
  else { take n = Owned<struct list_node>(p);
         assert (n.head > 0);
         take rest = PosIntList(n.tail);
         return (Cons {hd: n.head, tl: rest}); }}
```

where the C type list_node and the CN type int_list are defined as:

```
struct list_node = { int head; struct list_node* tail; }
datatype int_list = { Nil {}, Cons { i32 hd, datatype int_list tl } }
```

Computationally, this "predicate" behaves like a function, taking in a pointer as an argument and constructing a result belonging to the algebraic data type int_list with constructors Nil and

`Cons`. Each time it is called, the predicate checks whether `p` is `NULL`. If so, it returns the empty list. Otherwise, it reads the value pointed to by `p`, claiming ownership (and checking that it has not already been claimed) of this heap cell, and asserts that the value stored in this node is positive, before performing a recursive call to read out the rest of the list.

For example, suppose we want to check `PosIntList(p)`, where `p` is `0x40`, against the following concrete marked heap.

```
{ 0x10 ↦ (false, (struct list_node) { .head = 5, .tail = 0x00 }),
  0x40 ↦ (false, (struct list_node) { .head = 10, .tail = 0x10 }) }
```

We begin by checking if `p` is **`NULL`**; it is not, so we take the **`else`** branch. We then attempt to take ownership the memory block from `p` (inclusive) to `p + sizeof(list_node)` (exclusive), which succeeds since it is unmarked in the heap; then we mark those locations as owned. We check that `n.head`, which is 10, is greater than zero. It is. Then we perform a recursive call, passing in `n.tail` (`0x10`). We once again take the **`else`** branch of the **`NULL`** check and mark the heap to claim ownership. Finally, on the second recursive call, with argument `0x00`, we take the first branch of the **`if`**, return back up the call stack, and produce the following result:

```
Cons { hd: 10, tl: Cons { hd: 5, tl: Nil {} } }
```

2.2 Generating Heaps from Specifications

At this point, we have seen how to *check* whether a given heap satisfies a given separation-logic formula in CN's restricted syntax. What about the converse problem of *generating* random heaps satisfying the separation-logic precondition of a function under test.

Naively, since we can check concrete heaps efficiently, we could consider generating completely random heap states, checking if they match the precondition, and discarding those that don't. However, while this generate-and-test method can be effective in very simple cases, it is not applicable for CN in general: the space of memory states is far too large for a randomly chosen heap to have a nontrivial chance of satisfying a sparse precondition like "`p` points to the root of a valid binary search tree."

A better approach is to consider a new interpretation of CN predicates as random *generators* of satisfying inputs. This idea has been applied previously to specifications of functional data structures [34, 42, 44, 68]. Our novel contribution is showing that it can be extended to separation-logic specifications of C heap structures.

For example, we can automatically derive the following `GenPosIntList` generator from the `PosIntList` predicate defined above:

```
generator (datatype int_list) GenPosIntList(pointer p) {
  if (p == NULL) { return (Nil {}); }
  else { let* n = arbitrary<struct list_node>();
         struct list_node p := n;
         assert (n.head > 0);
         let* rest = GenPosIntList(n.tail);
         return (Cons {hd: n.head, tl: rest}); } }
```

The first line in the **`else`** branch is a **`let`**`*`, which introduces a variable `n` bound to an arbitrary **`struct`** `list_node`. For integer types, **`arbitrary`** samples from a uniform distribution. For pointer types, it chooses randomly between generating `NULL` or a new heap pointer (via `malloc`). An arbitrary value for a struct is recursively defined as a struct built from arbitrary members.

The second line of the **else** performs an assignment to the under-construction heap, checking that the assignment is valid—that is, that it satisfies two conditions: there must be enough space, and none of the memory can have been assigned to before. (Each assignment corresponds to an **Owned** in the original predicate, so assigning to the same location twice would violate the disjointness requirement of **Owned**.) Here, we check that the memory cells from `p` to `p + sizeof(struct list_node) - 1` are allocated and that none of these bytes have been assigned to. If both of these conditions are true, `n` is assigned to `p`. Otherwise, the generator will backtrack to where the pointer was generated and either allocate more memory or generate a different pointer.

If the **assert** in the next line fails because `n.head` is less than 1, the generator will backtrack to where `n` was generated and try again. Finally, we make a recursive call with our newly generated tail pointer and use the result to build our return value.

To implement a generator that does all this, we need to build a heap while checking that locations are assigned at most once; for this we use a marked heap, as we did above for checking heap predicates. We also need to signal when **assert**s fail, so it makes sense to begin with a state-option monad, as we did for checking specifications. We also need a source of randomness to allow for making random choices. This results in a more complex state, `GenState` = { `seed` :: `RandomSeed`, `heap` :: `MarkedHeap` }, for a state-option monad, `Gen′ a` = `GenState` ⟶ `Option` (a, `GenState`). (The prime indicates that we will need to generalize it a bit more, below.)

As a concrete example, let's see how the two-list-cell heap that we saw in the previous subsection might be generated. Suppose we call `GenPosIntList(p)`, where `p` is `0x40`, with the heap { `0x40` ↦ (`false`, `(char)0`) }. Here, `0x40` only points to a single byte of memory, which has not been assigned to yet (marked `false`). As a result, assigning to `0x41` would be an out-of-bounds access. (We assume that `0x40` points to zero for simplicity of presentation; in reality, our final generator in C uses `malloc` and so does not zero memory.)

Since `p` is non-null, we start by taking the **else** branch of the conditional. For the **let*** and **arbitrary**, suppose we generate `(struct list_node) { .head = -5, .tail = 0x00 }`.

Assuming **int**s are 4 bytes and pointers are 8 bytes, the next line wants to assign `n` to locations `0x40` through `0x4F` (a single `struct list_node` requires 16 bytes due to padding). But our current heap only has `0x40` allocated. Therefore, we must backtrack to the caller of `GenPosIntList` to reallocate `p` to point to at least 16 bytes.

Our current generation monad, `Gen′`, simply returns `None` when it fails, providing no information about the cause of the failure. But here we need to tell the generator that generated `p` how many should actually be allocated. We introduce two new types, `Result` and `Failure`, where `Result` is like `Option`, except it has carries a value of type `Failure` instead of `None`. We enrich the monad to communicate this information to where `p` was allocated. We change the state-option monad, `Gen′`, to a reader-state-exception monad, `Gen a` = `GenEnv a` ⟶ `GenState a` ⟶ `Result` (a × (`GenState a`)), where `GenEnv` contains an element of `Option Failure`.

After backtracking and reallocating `p` with 16 bytes, we replay the same choices, putting us back at the "`:=`" with `n` equal to `(struct list_node) { .head = -5, .tail = 0x00 }`, but with a different heap: { `0x40` ↦ (`false`, `(char)0`), `0x41` ↦ (`false`, `(char)0`), …, `0x4F` ↦ (`false`, `(char)0`) }. This time the assignment succeeds, resulting in the heap { `0x40` ↦ (`true`, `(struct list_node){ .head = -5, .tail = 0x00 }`) }.

Next we come to the **assert**, which checks if `n.head` (-5) is positive and fails. We then backtrack to the assignment, un-marking `0x40` to `0x4F`, before backtracking to the **let***. Since the assertion that failed referenced `n`, we try regenerating `n`. Note that we wanted to make the same choices after reallocating for the previous failure, since we did not want reallocation to change our generator's distribution. However, for this failure we want to make different choices, because we are generating a different value for `n` and that might lead to a different path being taken. Suppose, this time, we generate (**struct** list_node) { .head = 10, .tail = 0x10 }, where 0x10 is allocated via malloc. After we perform the assignment again, we have the heap:

{ 0x10 ↦ (false, (char)0),
0x40 ↦ (true, (**struct** list_node) { .head = 10, .tail = 0x10 }) }.

This time the assertion passes, so we continue with the recursive call. Similar to what we just walked through, this call will backtrack to reallocate 16 bytes for 0x10. Eliding the details, suppose that the result of the recursive call is Cons { hd: 5, tl : Nil {} }. The final result is then Cons { hd: 10, tl: Cons { hd: 5, tl : Nil {} } }, with this heap:

{ 0x10 ↦ (true, (**struct** list_node) { .head = 5, .tail = 0x00 }),
0x40 ↦ (true, (**struct** list_node) { .head = 10, .tail = 0x10 }) }

3 From Specifications to Generators

The scheme sketched in Section 2 is sufficient to transform a precondition into a generator that is guaranteed to produce only values and heap states satisfy the precondition. However, if this scheme is applied naively, the exact shape of the original predicate can make a huge difference in the efficiency of the derived generator, and the most intuitive way to write the predicate does not necessarily yield a generator that minimizes backtracking. To do better, we need to be able to reason about the structure of the predicate and transform it into a more generator-friendly version. Accordingly, we introduce a lower-level DSL that, in particular, separates generating values from placing them in memory and introduces additional forms that are useful for optimizations, like **pick**, which randomly selects from a list of generators.

Internally, Bennet follows roughly the structure of an optimizing compiler, with a frontend, an intermediate representation, optimization passes, code generation, and a runtime system. First, we convert CN specifications and predicates to the generator DSL (Sections 3.1 and 3.2; we define the DSL and its monadic semantics—the interesting part—in Section 3.1 and the translation from CN into the DSL in 3.2). Second, we optimize generators by lifting logical constraints into the DSL (Section 3.3) and reordering variables (Section 3.4). Third, we add additional information to our DSL to ease our lowering to C (Section 3.5). Finally, we lower to C and link with some runtime support code (Section 4).

3.1 Generator DSL

We gave a high-level sketch of our generator DSL in Section 2.2; we now make it precise. We use the metavariable cty for C types and bty for CN types. The complete syntax is:

$$
\begin{aligned}
\mathsf{e} &::= \mathsf{x} \mid \mathsf{n} \mid \mathsf{e}_1 == \mathsf{e}_2 \mid \mathsf{e}_1 < \mathsf{e}_2 \mid \texttt{!e} \mid \mathsf{e}_1 \texttt{ || } \mathsf{e}_2 \mid \mathsf{e}_1 \,\&\&\, \mathsf{e}_2 \mid \mathsf{e}_1 \texttt{ implies } \mathsf{e}_2 \mid \mathsf{e}_1 \mathrel{?} \mathsf{e}_2 : \mathsf{e}_3 \\
&\quad \mid \texttt{array\_shift<cty>}(\mathsf{e}_1, \mathsf{e}_2) \mid \ldots \\
\mathsf{g} &::= \textbf{arbitrary}\texttt{<cty>()} \mid \textbf{uniform}\texttt{<cty>()} \mid \textbf{alloc}\texttt{()} \mid \textbf{pick}\texttt{([}\mathsf{g}_1\texttt{, ..., }\mathsf{g}_m\texttt{])} \\
&\quad \mid \mathsf{GenName}(\mathsf{x}_1 : \mathsf{e}_1, \ldots, \mathsf{x}_n : \mathsf{e}_n) \mid \textbf{return}\ \mathsf{e} \mid \texttt{cty e := e'; g'} \mid \textbf{let}\ \texttt{x = e; g'} \\
&\quad \mid \textbf{let*}\ \texttt{x = g; g'} \mid \textbf{let}\texttt{(n)*}\ \texttt{x =} \mathsf{g}_1 ; \mathsf{g}_2 \mid \textbf{assert}\texttt{(e); g'} \mid \textbf{if}\ \texttt{(e)}\ \{ \mathsf{g}_1 \}\ \textbf{else}\ \{ \mathsf{g}_2 \} \\
\mathsf{gen} &::= \textbf{generator}\ \texttt{bty GenName(bty}_1\ \mathsf{x}_1\texttt{, ..., bty}_n\ \mathsf{x}_n\texttt{)\{ g \}}
\end{aligned}
$$

The grammar is built up from a set of "pure" expressions e that do not interact with the heap. We don't need to go into much detail about these, since computing terms is done via Fulminate. They include standard arithmetic operators as well as logical implication (written implies) and pointer arithmetic (written array_shift because it is usually used with pointers to arrays). The pure expression array_shift<int>(p, 10) is equivalent to the C expression (**int***)p + 10.

The semantics for the DSL is defined by a translation to the reader-state-exception monad `Gen a`:

```
GenState = { seed :: RandomSeed, heap :: MarkedHeap, vars :: Map Sym cnPExp }
GenEnv a = { failure :: Option Failure, ctx :: GenCtx a }
Failure = Allocation Sym Int | Assertion (Set Sym)
GenCtx a = Sym → [a] → Gen a
Result a = Success a | Error Failure
Gen a = GenEnv a → GenState a → Result (a × (GenState a))
```

A generator's state (`GenState`) consists of a random seed (`RandomSeed`) and a marked heap (`MarkedHeap`) that tags each location with a boolean value. We also need a map from variables to values, stored as pure CN expressions of type `cnPExp`.

The generation environment (`GenEnv`) includes an optional `Failure` cause, to support more intelligent regeneration after backtracking. These come in two forms:

- *Allocation failures.* Since CN predicates can perform pointer arithmetic, it is difficult to know ahead of time exactly how many bytes should be allocated in the heap for a given pointer. If we guess wrong, we must be able to backtrack and reallocate the pointer to point to a larger block of memory. In this case, we need to know the pointer that needs to be reallocated and how much memory is required, resulting in the failure value `Allocation "p" 101`.
- *Assertion failures.* When logical constraints are not satisfied, we must also backtrack. In this case, many variables may be to blame. We take all the variables referenced by a failing logical constraint as the argument to the `Assertion` value. For example, if `x / 2 == y / 3` is a logical constraint that evaluates to false, the failure would be `Assertion {"x", "y"}`.

The generation environment also includes a context `GenCtx` to allow calls to other generators; for each, it stores the name of the generator and a list of arguments.

The `Error` case stores information about why a failure occurred, allowing the generator to be more targeted about what values to regenerate or how much to reallocate. The monad, `Gen a`, takes the environment and state, producing `Success` with a value of type `A` and an updated state on success and producing `Error` on failure.

Finally, pure expressions are effect-less calculations denoting booleans, integers, maps, and structs that may appear at the leaves of DSL expressions. To evaluate them in the definitions below, we use the function `evalPure : cnPExp → cnPExp`, which takes pure expressions and produces pure expressions. This corresponds to reusing some of Fulminate's internals in our implementation.

The semantics of the DSL is a function $\mathcal{M}[\![\cdot]\!]$ from DSL terms to `Gen cnPExp`. All our generators produce CN values, represented as constant pure expressions.

The rest of this subsection explains how this function works for the various cases of the DSL syntax.

3.1.1 Basic Constructs. The **if**-**else** and **return** constructs do not affect ownership and do not introduce new values, so their translations into the monad are simple. We substitute any variables with their values from the state—this substitution is written [`state.vars`].

$$\mathcal{M}[\![\textbf{if}\ (\mathrm{e})\ \{\ \mathrm{g}_1\ \}\ \textbf{else}\ \{\ \mathrm{g}_2\ \}]\!] = \lambda\mathrm{env}.\ \lambda\mathtt{state}.$$
$$\textbf{if}\ \mathtt{evalPure(e[state.vars])}\ \textbf{then}\ \mathcal{M}[\![\mathrm{g}_1]\!](\mathtt{env})(\mathtt{state})\ \textbf{else}\ \mathcal{M}[\![\mathrm{g}_2]\!](\mathtt{env})(\mathtt{state})$$
$$\mathcal{M}[\![\textbf{return}\ \mathrm{e}]\!] = \lambda\mathrm{env}.\ \lambda\mathtt{state}.\ \mathtt{Success}\ (\mathtt{evalPure(e[state.vars])}, \mathtt{state})$$

3.1.2 Primitive Values. Next, we need to be able to generate arbitrary values of all C types. We begin with integers and pointers, then discuss structs.

$$
\begin{array}{l}
\mathcal{M}[\![\mathbf{alloc}()]\!] = \lambda\mathsf{env}.\ \lambda\mathtt{state}. \\
\quad \mathbf{match}\ (\mathsf{env}.\mathtt{failure})\ \mathbf{with} \\
\quad |\ \mathsf{Some}\ (\mathtt{Allocation}\ \_\ \mathsf{n}) \\
\qquad \mathbf{let}\ (\mathsf{p}, \mathsf{heap}') = \mathsf{heapAlloc}(\mathsf{n})(\mathtt{state}.\mathsf{heap})\ \mathbf{in} \\
\qquad \mathsf{Success}\ (\mathsf{p}, \mathtt{state}\{\mathsf{heap} \leftarrow \mathsf{heap}'\}) \\
\quad |\ \_ \Rightarrow \\
\qquad \mathbf{let}\ (\mathsf{n}, \mathsf{seed}') = \mathtt{uniformSample}(1, \mathtt{nullInEvery})(\mathtt{state}.\mathsf{seed})\ \mathbf{in} \\
\qquad \mathbf{if}\ \mathsf{n} = 0\ \mathbf{then} \\
\qquad\quad \mathsf{Success}\ (\mathbf{NULL}, \mathtt{state}\{\mathsf{seed} \leftarrow \mathsf{seed}'\}) \\
\qquad \mathbf{else} \\
\qquad\quad \mathbf{let}\ (\mathsf{p}, \mathsf{heap}') = \mathsf{heapAlloc}(\mathtt{sizeof}(\mathtt{intmax\_t}))(\mathtt{state}.\mathsf{heap})\ \mathbf{in} \\
\qquad\quad \mathsf{Success}\ (\mathsf{p}, \mathtt{state}\{\mathsf{seed} \leftarrow \mathsf{seed}', \mathsf{heap} \leftarrow \mathsf{heap}'\})
\end{array}
$$

Fig. 1. Pointer generation (**alloc**)

Integers. To generate integers, the DSL provides a primitive, **uniform**<cty>() that samples from the uniform distribution on integers of the given type. The function isIntTy returns true when its argument is an integer type and isSignedTy returns true when its argument is signed.

$$
\begin{array}{l}
\mathcal{M}[\![\mathbf{uniform}\langle\mathsf{cty}\rangle()]\!] = \lambda\mathsf{env}.\ \lambda\mathtt{state}. \\
\quad \mathbf{let}\ (\mathtt{minVal},\ \mathtt{maxVal}) = \mathsf{getBounds}(\mathtt{isSignedTy}(\mathsf{cty}),\ \mathtt{sizeof}(\mathsf{cty}))\ \mathbf{in} \\
\quad \mathbf{let}\ (\mathsf{v},\ \mathsf{seed}') = \mathtt{uniformSample}(\mathtt{minVal},\ \mathtt{maxVal})(\mathtt{state}.\mathsf{seed})\ \mathbf{in} \\
\quad \mathsf{Success}\ (\mathsf{v},\ \mathtt{state}\{\mathsf{seed} \leftarrow \mathsf{seed}'\})
\end{array}
$$

The uniformSample function takes the bounds of a range and returns a function of type RandomSeed → (cnPExp, RandomSeed). The returned cnPExp is an integer constant sampled from the provided range. We use getBounds to get the range of possible values for a given signedness and number of bytes.

Pointers. Pointer types are more complicated than integers—we cannot just naively choose from among all representable values—so we define a dedicated syntactic form for them in the DSL, written **alloc**().

Consider a CN specification that contains **Owned**<char>(p). Initially, we would like the corresponding generator in the DSL to allocate 1 byte and store an arbitrary char at p in the generated heap. However, suppose we later encounter part of the specification that includes the constraint **Owned**<int>(array_shift<char>(p, 100)). Then we need p to point to at least 101 bytes. It seems difficult to know, when we first allocate p, how many bytes we actually need to allocate. Perhaps we could perform a sophisticated interprocedural analysis to overestimate the amount of memory needed; there is no ⊤ element here, though, since we can't allocate infinite memory, which complicates widening in such an analysis. So, instead, we use backtracking to reallocate more memory later, when needed, and omit a size argument to **alloc**.

We track this information via the failure field in the environment. Failures that require allocation look like Allocation x n, where x is the name of the base pointer and n is the number of bytes required. The heapAlloc function takes a number of bytes and returns a function of type MarkedHeap → MarkedHeap that allocates the requested number of bytes. (See Figure 1.)

```
arbitrary<cty>() when isIntTy(cty) = uniform<cty>()
arbitrary<cty*>() = alloc()
arbitrary<struct tag>() =
    let* fresh₁ = arbitrary<cty₁>();  ···
    let* freshₙ = arbitrary⟨ctyₙ⟩();
    return (struct tag {.x₁ = fresh₁, ..., .xₙ = freshₙ})
```

Fig. 2. **arbitrary**

When handling an `Allocation` failure, it will allocate at least the number of bytes needed. Otherwise, it chooses `NULL` with probability `1/nullInEvery` (where `nullInEvery` is a user setting), and the rest of the time creates a pointer to at least one byte (via `malloc`-ing the size of `intmax_t`).

In the case of an `Assertion` failure, the default branch is taken, since it means we want a different pointer, such as wanting **`NULL`**.

Structs. Next we define the DSL form **`arbitrary`**, which can generate values of any C type (except, for the moment, unions, as CN does not support them yet). We introduce a new construct, **`let`**`*`, a generalization of **`let`** that allows generators on the right side of the equals, rather than just pure expressions. It is explained carefully later; intuitively, it marks every location where a value was made by the generator and so where generator failures need to be handled.

We define **`arbitrary`** in terms of **`uniform`**, **`alloc`**, and **`let`**`*`. For the struct case, the struct's definition is **`struct`** `tag { cty`$_1$ `x`$_1$`; ...; cty`$_n$ `x`$_n$`; }`. We desugar any **`typedef`**s. The function `isIntTy` returns whether its argument is an integer type and `isSignedTy` returns whether its argument is a signed integer type. For primitive types, we use the respective primitive generators. We generate arbitrary structs by recursively generating arbitrary values for the members.

3.1.3 Satisfying Logical Constraints. Logical constraints on generated values are written **`assert`**`(e)`. If we find we have made a logical constraint false at some point during the generation process, we backtrack and regenerate values, beginning with the most recently generated. To avoid regenerating values that have nothing to do with the logical constraint that failed, we represent such failures as `Assertion xs`, where `xs` is the set of all variables that could be responsible for the failure.

$$\begin{aligned}&\mathcal{M}[\![\textbf{assert}(\mathtt{e});\ \mathtt{g}]\!] = \lambda\mathtt{env}.\ \lambda\mathtt{state}.\\ &\quad \textbf{if}\ \mathtt{evalPure(e[state.vars])}\\ &\quad \textbf{then}\ \mathcal{M}[\![\mathtt{g}]\!](\mathtt{env})(\mathtt{state})\\ &\quad \textbf{else}\ \mathtt{Error\ (Assertion\ (freeVars(e)))}\end{aligned}$$

If a logical constraint evaluates to false, the `Assertion` exception is thrown, carrying the set of variables in the constraint. This only approximates the variables that are truly to blame for a failure. It excludes variables that have previously constrained variables involved in the failing assertion. For example, `y` could be constrained to be less than `z`, which is `INT_MIN`, making `x < y` impossible to satisfy. However, when `x < y` fails, only `x` and `y` would be blamed. This approximation has worked well in practice; we leave more sophisticated analysis to future work.

3.1.4 Satisfying Resource Constraints. Next, let's consider resource constraints.

Ownership Primitives. When checking a CN predicate against an existing heap, **`Owned`** returns the value stored at a location. Furthermore, **`Owned`** implies guarantees about the memory it reads from; it ensures that `e` evaluates to a valid address (no SEGFAULT) and it does not overlap with memory claimed by a previous **`Owned`** call. Conversely, when generating a heap to satisfy **`Owned`**`<cty>(p)`, we need to fill in the heap locations that `p` points to so that they contain an arbitrary generated

```
𝓜[[cty e_addr := e_val; g]] = λenv. λstate.
  let p = evalPure(e_addr[state.vars]) in
  if not isUnmarked(p, sizeof(cty), state.heap) then
    Error (Assertion (freeVars(e_addr)))
  else if not isAllocated(p, sizeof(cty), state.heap) then
    let parent = pointerVar(e_addr) in
    Error (Allocation (parent) (p − parent + sizeof(cty)))
  else
    let value = evalPure(e_val[state.vars]) in
    let heap′ = heapAssign(p, cty, value)(state.heap) in
    𝓜[[g]](env)(state{heap ← heap′})
```

Fig. 3. Memory Assignment

value of type cty. Also, if the locations being assigned to are not valid (allocated) addresses, or if they have already been assigned to by a previous assignment, we want to backtrack.

The *assignment* expression in our DSL (Figure 3) serves this purpose. It evaluates the address that will be assigned to and checks that the necessary number of bytes starting at that address are unmarked. If not, it returns an assertion failure, triggering backtracking to generate a different (non-overlapping) pointer. Otherwise, it checks that the target bytes are allocated and, if not, returns an allocation failure to reallocate the necessary bytes. (Since the address could be an array_shift, it extracts the parent pointer, which is what needs to be reallocated with a larger number of bytes.) Finally, it evaluates the value to store, stores it in the heap, and continues with the remaining generator g.

The function pointerVar, given a pure expression, returns the name of a pointer-typed variable that appears in it. For example, given char array_shift<char>(p, 10) := ..., we can determine that p must reallocate at least 11 bytes (array_shift<char>(p, 10) - p + sizeof(char)). To make this operation sensible, we require that address expressions contain a single pointer-typed variable. All the addresses we've encountered in existing CN code are either specific pointers or offsets from pointers.

We assume the existence of three functions over marked heaps. First, isAllocated(p, sz, heap) checks that at least sz bytes are available at p. Second, isUnmarked(p, sz, heap) checks that all of the bytes between p and p + sz are unmarked. Finally, heapAssign(p, cty, v) produces a function of type MarkedHeap → MarkedHeap that marks and assigns v of type cty to p.

If any of the memory we're trying to assign to has already been marked, we backtrack on all the free variables in the address expression. This allows regenerating pointer offsets, as well as the pointer itself. If any of the memory isn't allocated, we return an Allocation error with the parent pointer. The size in this case is the bytes between the parent and the address of interest, plus the size of the value we want to assign. Finally, if the memory is allocated and has not been assigned to, we evaluate the expression on the right and assign it to p. We update the heap and call the rest of the generator.

User-defined Predicates. One might expect calls to user-defined predicates to simply translate to calls to user-defined generators. Due to backtracking, there is a bit more to it. (See Figure 4.)

Suppose we define predicates F and G. If we generate x = y = −1, calling F(x + y) would fail. We then need to backtrack out of the body of F, but the exception Assertion {n} contains n,

$$
\begin{aligned}
&\mathcal{M}[\![\texttt{PredName}(\texttt{x}_1 : \texttt{e}_1, \ldots, \texttt{x}_\texttt{n} : \texttt{e}_\texttt{n})]\!] = \lambda\texttt{env}.\ \lambda\texttt{state}.\\
&\quad \textbf{let } [\texttt{v}_1, \ldots, \texttt{v}_\texttt{n}] = [\texttt{evalPure}(\texttt{e}_1[\texttt{state.vars}]), \ldots, \texttt{evalPure}(\texttt{e}_\texttt{n}[\texttt{state.vars}])] \textbf{ in}\\
&\quad \textbf{match } (\texttt{env.ctx})(\texttt{PredName})([\texttt{v}_1, \ldots, \texttt{v}_\texttt{n}]) \texttt{ with}\\
&\quad \mid \texttt{Error (Allocation y sz) } \textbf{when } (\texttt{for some i}, \texttt{x}_\texttt{i} = \texttt{y}) \Rightarrow\\
&\qquad \textbf{let } \texttt{p = pointerVar(e\_i)} \textbf{ in}\\
&\qquad \texttt{Error (Allocation p } (\texttt{sz} + \texttt{evalPure}(\texttt{v}_\texttt{i} - (\texttt{p}[\texttt{state.vars}]))))\\
&\quad \mid \texttt{Error (Assertion ys)} \Rightarrow\\
&\qquad \texttt{Error (Assertion } \{\texttt{z} \mid \texttt{y} \in \texttt{ys} \land \texttt{y} = \texttt{x}_\texttt{i} \land \texttt{z} \in \texttt{freeVars}(\texttt{e}_\texttt{i})\})\\
&\quad \mid \texttt{result} \Rightarrow \texttt{result}
\end{aligned}
$$

Fig. 4. Call to user-defined predicate

which does not appear in the precondition or the function's arguments. We want `Assertion` {n} to become `Assertion` {y, z}.

```
generator i32 F(i32 n) {
  assert (n > 0);
  return n; }
generator i32 G() {
  let* x = uniform<i32>();
  let* y = uniform<i32>();
  let* z = F(x + y);
  return z; }
```

To achieve this, we tag all calls to generators derived from user-defined predicates with their argument names, so that backtracking failures can have their symbols renamed to the new scope. We provide the semantics of calls below. It looks up `PredName` in `env.ctx`, passing in the arguments. If the called generator returns a failure, the blamed variables are renamed from the callee's context to the caller's context.

As in the definition of assignment, due to the use of `pointerVar`, we require that pointer arguments involve a single pointer-typed variable. Suppose a generator has an argument `p`, which we're backtracking to reallocate 10 bytes. As we backtrack to the caller, what if the argument for `p` is `array_shift<int>(q, 5)`? Then not only do we need to rename `p` to `q` in the failure, but we need to add 5 to the number of bytes needed. This is because `q` is getting reallocated now, not `p`.

3.1.5 Backtracking Destinations. It remains to define semantics for one construct, **`let`**`*` (see Figure 5). We've noted that **`let`**`*` allows generators on its right side and that this is where newly generated values are introduced. Accordingly, **`let`**`*` is responsible for handling failures.

Recall that there are two type of failures: assertion failures, when a different value is needed, and allocation failures, when more memory needs to be allocated.

Assertion Failures. In the first case, we want to generate new values, so we use `newSeed`, a function which takes the current `RandomSeed` and produces a new independent one. However, we don't want to keep trying forever on the same variable. Consider the logical constraint `x < y` over two arguments of type `int32_t`, when `x = INT32_MAX`. In this case, it is impossible to generate a `y` which is greater than `x`. If `y` is generated after `x`, it will be reached by backtracking first, and so a satisfying solution would require backtracking further, to `x`. So we define a maximum number of assertion regenerations allowed. The count is reset every time a variable defined earlier is backtracked to. We reflect this in the syntax as **`let`**`(n)*`, where `n` is permitted number of regenerations. Actually, **`let`**`*` is syntactic sugar for **`let`**`*(max_regenerations)`, where `max_regenerations` is a user-configurable setting, defaulting to 25. Plain **`let`** `x = e` is syntactic sugar for **`let`**`(0)* x =` **`return`** `e`. Since **`return`** is deterministic, regenerating `x` would be result in the same value, so we set the number of backtracks allowed to zero.

$$
\begin{array}{l}
\mathcal{M}[\![\textbf{let}(\mathtt{n})\texttt{*}\ \mathtt{x} = \mathtt{g}_{\mathtt{inner}}; \mathtt{g}_{\mathtt{rest}}]\!] = \\
\quad \mathtt{ask} >\!\!>= \lambda\mathtt{oldEnv}. \\
\quad \mathtt{get} >\!\!>= \lambda\mathtt{oldState}. \\
\quad \mathcal{M}[\![\mathtt{g}_{\mathtt{inner}}]\!] >\!\!>= \lambda\mathtt{v}.\ \lambda\mathtt{env}.\ \lambda\mathtt{state}. \\
\quad \mathtt{match}\ \mathcal{M}[\![\mathtt{g}_{\mathtt{rest}}]\!](\mathtt{env})(\mathtt{state}\{\mathtt{vars} \leftarrow \mathtt{state.vars} \cup \{\mathtt{x} \mapsto \mathtt{v}\}\})\ \textbf{with} \\
\quad |\ \mathtt{Error}\ (\mathtt{Allocation}\ \mathtt{x}'\ \mathtt{sz})\ \textbf{when}\ \mathtt{x} = \mathtt{x}' \Rightarrow \\
\qquad \textbf{let}\ \mathtt{env}' = \mathtt{oldEnv}\{\mathtt{failure} \leftarrow \mathtt{Some}\ (\mathtt{Allocation}\ \mathtt{x}'\ \mathtt{sz})\}\ \textbf{in} \\
\qquad \textbf{let}\ \mathtt{state}' = \mathtt{oldState}\{\mathtt{seed} \leftarrow (\mathtt{state.seed})\}\ \textbf{in} \\
\qquad \mathcal{M}[\![\textbf{let}(\mathtt{n})\texttt{*}\ \mathtt{x} = \mathtt{g}_{\mathtt{inner}}; \mathtt{g}_{\mathtt{rest}}]\!](\mathtt{env}')(\mathtt{state}') \\
\quad |\ \mathtt{Error}\ (\mathtt{Assertion}\ \mathtt{xs})\ \textbf{when}\ \mathtt{x} \in \mathtt{xs} \Rightarrow \\
\qquad \textbf{if}\ \mathtt{n}\ >\ \mathtt{0}\ \mathtt{then} \\
\qquad\quad \textbf{let}\ \mathtt{env'}\ =\ \mathtt{oldEnv}\{\mathtt{failure} \leftarrow \mathtt{Some}\ (\mathtt{Assertion}\ \mathtt{xs})\}\ \textbf{in} \\
\qquad\quad \textbf{let}\ \mathtt{state'}\ =\ \mathtt{oldState}\{\mathtt{seed} \leftarrow \mathtt{newSeed}(\mathtt{state.seed})\}\ \textbf{in} \\
\qquad\quad \mathcal{M}[\![\textbf{let}(\mathtt{n}\ -\ \mathtt{1})\texttt{*}\ \mathtt{x}\ = \mathtt{g}_{\mathtt{inner}}; \mathtt{g}_{\mathtt{rest}}]\!](\mathtt{env}')(\mathtt{state}') \\
\qquad \textbf{else} \\
\qquad\quad \mathtt{Error}\ (\mathtt{Assertion}\ ((\mathtt{xs} \setminus \{\mathtt{x}\}) \cup \mathtt{freeVars}\ \mathtt{g}_{\mathtt{inner}})) \\
\quad |\ \mathtt{result} \Rightarrow \mathtt{result}
\end{array}
$$

Fig. 5. Monadic Semantics of **let***

Allocation Failures. In the second case, we do *not* want to generate new values. If we made new choices after performing reallocation, taking the exact same path through the generator again would be unlikely. This would skew the generator's distribution by making inputs that require more memory less likely. Instead, we want to allocate more memory than originally but otherwise generate the same values. We do this by saving the `RandomSeed` after generating a value. If an allocation failure causes is to backtrack to a given **let***, we allocate the required amount of memory and then restart the following generation using the stored `RandomSeed`.

Monadic Semantics. The semantics of **let*** are given in Figure 5. To lighten the handling of state and environment, we rely on some standard operations for reader and state monads: `ask` returns the current environment, and `get`, which returns the current state, just as `ask` returns the environment. We also use the Haskell notation >>= for monadic bind, of type $\mathtt{Gen\ a} \to (\mathtt{a} \to \mathtt{Gen\ b}) \to \mathtt{Gen\ b}$.

In the allocation-failure case, we use the state after calling the inner generator the first time, since **alloc** may use randomness to decide between `NULL` and a valid pointer, but won't when processing an allocation failure. When all regeneration attempts are exhausted, it returns the previous failure, but modifies the set of variables to blame. This allows us to backtrack to deterministic variables (those bound by **let**s whose right-hand sides are pure), where **let** `x = 10 + y` can be desugared into **let**`(0)* x =` **return** `(10 + y)`. In that case, an assertion failure that blames `x` becomes one that blames `y`.

3.2 Translation from CN to the DSL

We next give the grammar for CN and show how to translate CN specifications and predicates to our DSL.

3.2.1 CN. The CN specification language is based on *constraints*, which come in two flavors: *logical constraints* and *resource constraints*. Logical constraints are bare pure expressions, which do not interact with the heap. They include, for example, the inequalities in the `double` function. Resource constraints are the part of CN that constrain the heap. There are two types of resource constraints: the primitive **Owned**`<cty>` and calls to user-defined predicates.

These constraints plus a pure **let** that simply gives a value a name, form statements. Statements can be placed in specifications or inside of clauses, which are used in predicates to allow branching and have return values. CN also allows types to be defined at the specification level, as we saw in the `PosIntList` example. The grammar of specifications is:

```
  resource ::= Owned<cty>(e) | PredName(e_1, ..., e_n)
 statement ::= let x = e; | take x = resource; | e;
statements ::= statement | statement statements
      spec ::= requires statements ensures statements'
    clause ::= statements return e
   clauses ::= clause | if (e) { clause } else { clauses }
      pred ::= predicate bty PredName(x_1, ..., x_n){ clauses }
```

3.2.2 Defining the Translation. We next show how how to translate from CN formulas to the internal DSL.

Basic constructs. Many CN constructs translate directly into the DSL:

```
CN-to-DSL[[if (e) { clause } else { clauses } ]] =
   if (e) { CN-to-DSL[[clause]] } else { CN-to-DSL[[clauses]] }
CN-to-DSL[[return e;]] = return e
CN-to-DSL[[let x = e; statements]] =
   let x = e; CN-to-DSL[[statements]]
CN-to-DSL[[e; statements]] =
   assert(e); CN-to-DSL[[statements]]
```

Next, let's consider resource constraints, which are introduced in CN via **take**. The interesting parts come from the translation of the different sorts of resources that can be taken; the translation of **take** itself is simple.

```
CN-to-DSL[[take x = resource; statements]] =
   let* x = CN-to-DSL[[resource]]; CN-to-DSL[[statements]]
```

We next consider how different sorts of resource constraints are translated.

CN's memory primitive, **Owned**, claims ownership of a piece of memory and returns the value stored there. Therefore, given **Owned**`<cty>(p)`, we would like `p` to point to an arbitrary value of type `cty`.

```
CN-to-DSL[[Owned<cty>(e)]] =
   let* fresh = arbitrary<cty>();
   cty e := fresh;
   return fresh
```

This translation generates a random value of type `cty`, assigns it to `e`, and returns the random value.

We define the translation from CN, assuming that we have a list, $\mathtt{x}_1, \ldots, \mathtt{x}_n$, of the names of the arguments of `PredName`. We annotate the names of the arguments because, in case of a failure, we need to be able to remap the variables referenced in the failure from the callee's context to the caller's context.

$$\mathtt{CN\text{-}to\text{-}DSL}[\![\mathtt{PredName(e_1,\ \ldots,\ e_n)}]\!] = \mathtt{PredName(x_1 : e_1,\ \ldots,\ x_n : e_n)}$$

3.2.3 Specifications and Predicates. We have now defined all we need to take a CN predicate and convert it into a generator that guarantees that its arguments and produced heap satisfy the predicate. Only one small thing remains: when testing functions, we also have to generate their arguments. We define a translation $\mathtt{CN\text{-}to\text{-}DSL}_{\mathtt{spec}}[\![\cdot]\!]$, which takes a specification and a list of argument names and their C types.

$$\begin{aligned}
&\mathtt{CN\text{-}to\text{-}DSL}_{\mathtt{spec}}[\![\mathbf{requires}\ \mathtt{statements}\ \mathbf{ensures}\ \mathtt{statements'}]\!] = \lambda(\mathtt{x_1}, \mathtt{cty_1}), \ldots, (\mathtt{x_n}, \mathtt{cty_n}).\\
&\quad \mathbf{let*}\ \mathtt{x_1} = \mathbf{arbitrary}\langle \mathtt{cty_1} \rangle\mathtt{()};\ \ \cdots\\
&\quad \mathbf{let*}\ \mathtt{x_n} = \mathbf{arbitrary}\langle \mathtt{cty_n} \rangle\mathtt{()};\\
&\quad \mathtt{CN\text{-}to\text{-}DSL}[\![\mathtt{statements}]\!]\\
&\quad \mathbf{return}\ \{\mathtt{x_1 : x_1,\ \ldots,\ x_n : x_n}\}
\end{aligned}$$

With every C function `f` and accompanying CN specification `spec` and arguments $\mathtt{x}_1, \ldots, \mathtt{x}_n$ with types $\mathtt{cty}_1, \ldots, \mathtt{cty}_n$, we create a mapping from `f` to $\mathtt{CN\text{-}to\text{-}DSL}_{\mathtt{spec}}[\![\mathtt{spec}]\!]$. The collection of these mappings, with type `GenCtx cnPExpr`, is the context passed to our semantics.

3.2.4 Inlining and Flattening Generators. Now that can get generators from our predicates, let's try to improve them. An easy way to provide more context and freedom to our optimization passes is to inline non-recursive generators. However, this can result in more complicated control flow, like the generator below.

```
let* x = if (e) { g₁ } else { g₂ }; g′ ⟹
    if (e) { let* x = g₁; g′ } else { let* x = g₂; g′ }
```

Therefore, we perform a flattening pass, where we try to move **`if`**s to the outermost level, turning the generator above into the one on the bottom.

While this has made each branch of the **`if`** have a constant `x`, it'd be even better if they could reason about `y`. If, `y` is defined in the preceding lines before this snippet (it is not an argument to the predicate) we can rewrite the snippet using **`pick`**.

```
g′; if (e) { g₁ } else { g₂ } ⟹
    pick ([(g′; assert(e); g₁), (g′; assert(!e); g₂)])
```

We define **`pick`** equivalently to the `backtrack` combinator from *Ode on a Random Urn* [46], which samples without replacement from a list of generators.

Given a list of generators, **`pick`** randomly selects between them. Should the generator selected fail with an assertion failure, **`pick`** will choose a different branch, only backtracking after it has tried all of them.

3.3 Lifting Logical Constraints

We want our optimization passes to have as much information as possible at their disposal. Consider the logical constraint `x == 10 || x == 11`. We could optimize a generator for each case and pick between them, if we could "lift" them to the level of our DSL. Lifting means taking logical operators in pure expressions and moving the logic into the DSL language.

In the case of conjunction, lifting is easy:

assert(e_1 && e_2); g $\Longrightarrow$ **assert**(e_1); **assert**(e_2); g

That is, we can split a conjunction into two separate **assert**s. If x is referred to by e_1 and not by e_2, the reordering pass may later be able to move **assert**(e_2) to before x is generated. Then, if e_2 is false, the generator will do its backtracking before wasting time generating x.

Implication and conditionals are lifted like this:

assert(e_1 implies e_2); g $\Longrightarrow$ **if** (e_1) { **assert**(e_2); g } **else** { g }
assert(e_1 ? e_2 : e_3); g $\Longrightarrow$ **if** (e_1) { **assert**(e_2); g } **else** { **assert**(e_3); g }

Similar to lifting conjunctions, these rewrites don't change the generator's distribution. They simply give the flattening pass (described below) more to do.

We use **pick** for disjunctions.

assert(e_1 || e_2); g $\Longrightarrow$ **pick**([(1, **assert**(e_1); g), (1, **assert**(e_2); g)])

Unlike the previous rewrites, lifting disjunctions does change the distribution. While this was initially a limitation to be fixed, users of Bennet have actually used this fact to skew their generator's distribution. For example, they might write, **assert**(x < INT32_MAX / 2 || x >= INT32_MAX / 2), which is a tautology. If x is uniformly distributed, x < INT32_MAX / 2 will be true about three-fourths of the time. However, due to the lifting, it will make the probability of each case 50%.

While lifting disjunctions has advantages, we can't perform it blindly, like the others. Consider the snippet to the right. Suppose we decide to split the assertion, but RecursivePred always return positive values. In the original form, n would've always passed the assertion. However, after splitting, the branch with **assert**(n < 0) will always fail, leading to needless backtracking.

```
let* n = RecursivePred();
assert (n < 0 || n >= 0);
```

Accordingly, we apply the heuristic of not splitting disjunctions that constrain the results of a non-primitive generator call in both branches. This way, we are never increasing the likelihood of a generator call's result failing some constraint, when it wouldn't have before the rewrite. It is conceivable that we could be more aggressive, using a more complicated heuristic to determine when to split disjunctions, though we have not seen a need to yet. Finally, we rerun the flattening pass, which will pull all the **pick**s and **if**s to the top level.

3.4 Reordering

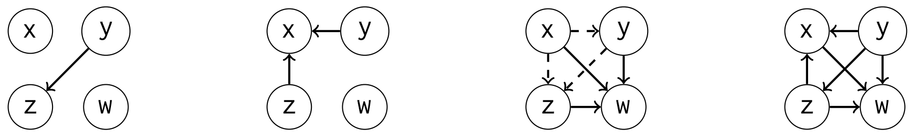
The final optimization pass reorders **let***s, so that, for any logical constraint that is an equality between a variable and an expression (ex: x == e), the variable (x) is generated after the free variables in the expression (e). It also calls recursive generators as early as allowed and moves assertions and assignments as early as it can. This means that, if backtracking is needed, it is known as soon as possible.

For example, consider the generator Foo on the right. We show all the intermediate graphs for Foo in Figure 6.

```
generator (i32, i32, i32) Foo() {
  let* x = uniform<i32>();
  let* y = uniform<i32>();
  let z = 10 + y;
  let w = uniform<i32>();
  assert (x == y + z);
  return (x, y, w); }
```

We begin by building a graph of data dependencies (Figure 6a). Traversing the generators, for each let* x = g, we add an edge from the variables in g to x.

We construct two graphs of preferred orderings, based on constraints. (We build two graphs so that we can prioritize equality constraints over constraints referring to recursive generators, in case of a conflict.)



(a) Data dependencies. (b) Equality dependencies. (c) Original ordering. (d) Resulting DAG.

Fig. 6. Reordering graphs for `Foo`

The first graph is for constraints of the form `x == e` ("equality dependencies"), adding edges from the variables in `e` to `x` (Figure 6b). We skip this if `e` is just a variable `y`, since either `y` or `x` already appears after the other. We do not add an edge if it would make the graph cyclic. Equality constraints are our highest priority when reordering, since they only have one satisfying assignment.

The second graph is for constraints where a local variable (generated from a primitive) is constrained by the result of a recursive generator. We call these "external call dependencies". (There are no recursive generator calls in `Foo`, so we omit this graph.) The reason for this heuristic is that non-primitive generators are slower than primitive ones, so, when backtracking from these constraints, it is faster to regenerate the local variable than call the slower one many times. Once again, we don't add any edges that would introduce a cycle.

We take our graph of data dependencies and add edges from the equality dependencies graph, provided they don't introduce a cycle. For external call dependencies, we add edges that don't introduce a cycle. We now have a dag that respects the data dependencies and tries to respect the constraint dependencies, prioritizing equality dependencies over external call dependencies.

One might wonder if we can just do a topographical sort and use it as our final ordering. Indeed, earlier versions of Bennet did this, but we found that arbitrarily shuffling the order of generation (due to our graphs being under-constrained) made debugging our generators much more difficult. If the user writes `w` after `x` and there is no reason to reorder them, we shouldn't. A side effect of this is that the user can suggest an order of generation to Bennet via the ordering in their predicate.

As a result, we take the ordering of variables in the original generator and add its edges to our graph, provided they don't add a cycle. In Figure 6c, the dashed edges are those that couldn't be added due to introducing a cycle, or already existing in the graph. Figure 6d displays the final graph, which gives a fully constrained ordering: `y`, `z`, `x`, `w`.

Having calculated a variable ordering, we take the **`let`**`*`s from the original generator and place them in this order. We also move assertions and assignments to appear as soon as all the variables they reference are defined. This improves performance, as we check whether constraints are satisfied as soon as all their variables are instantiated, backtracking immediately if necessary, rather than wasting time generating values that will later be discarded. Since we avoid introducing cycles, the order in which we consider constraints affects the final order, as the last edge in a cycle is the one omitted. We do not choose a particular strategy and leave such investigations to future work.

In addition to the two main optimization passes detailed in 3.3 and 3.4, some lighter simplification passes get run between each of the main passes. These include applying CN's internal term simplifier (a partial evaluator for pure terms) and introducing intermediate variables for struct and record members, which gives a bit more flexibility for reordering.

3.4.1 Specialization. Now that we've reordered variables according to equality constraints, we'd like to use them to improve our generators. For all `let* x = uniform()` where there exists a constraint of the form `x == e` and all variables in `e` are defined before `x`, replace it with `let x = e`. We perform a similar specialization for inequalities, via a ranged generator built into our runtime.

3.5 Elaboration

The last stage of the pipeline is to compile our DSL down to C code that can be linked with the system under test together with a test-running harness. Due to the low-level nature of C, we need to reason explicitly about which points can be backtracked to, as well as memory management. We also need to avoid recursion leading to stack overflows and non-termination.

We accomplish this by annotating our DSL in two major steps. First, we add a notion of "sized" generators, inspired by QuickCheck [24] to ensure that our generators terminate and avoid stack overflows. Second, we explicitly annotate which variables should be backtracked to, so that our C implementation has the necessary information.

3.5.1 Sized Generation. Since Bennet makes random choices and generators can be recursive, sometimes generators can recurse to the point of causing a stack overflow. As a result, we have a stack depth limit of 255, where if it is exceeded, we throw an exception. However, this introduces the question of which variables to regenerate to avoid the recursive call that resulted in the stack depth being exceeded. As such, we introduce a new type of exception called a depth exception, which is near identical to the assertion exception. The only difference is that when a call to a generator fails with a depth exception, all variables that appeared in if-conditions are blamed.

While this stack depth limit prevents overflows, for data structures like trees, which branch, there's an exponential blowup of recursive calls, leading to a low chance of termination. QuickCheck [24] solves this by restricting the size of the inputs generated.

We choose a size uniformly from 1 to the configured max input size, for each generation attempt. We limit the depth of recursive calls by adding a `size` argument. If recursive calls exceed a stack depth of `size`, we throw a depth exception. Recursive calls here is meant to be calls to generators that can return to the current generator. So a recursive generator `f` can call another recursive generator `g` while resetting the `size` argument, if there is no path in the call graph from `g` to `f`.

For each recursive function, we consider each path (through **`pick`** and **`if-else`**). At the end of each path (at the first non-branching term), we count the number of calls that could lead back to the current generator. If there is one recursive call, we call it with `size - 1`. Otherwise, if there are `n` recursive calls, we call them with `size / n`.

In line with QuickCheck's approach to sized generation, we restrict **`uniform`**`<cty>()` for unsigned integers to be uniformly across an interval $[0, \mathtt{size} - 1]$, for a given size `size`. If the desired type is signed, we sample from the interval $[-\mathtt{size} + 1, \mathtt{size} - 1]$.

3.5.2 Backtracking Annotations. As given below, we make the possible paths of backtracking explicit via two sets of annotations. You can also see the additional `size` variable.

$$\begin{aligned} \mathtt{g}_{\mathtt{elab}} ::= {} & \mathbf{pick}([(\mathtt{n}_1,\ \mathtt{g}_1);\ \ldots;\ (\mathtt{n}_m,\ \mathtt{g}_m)]) \uparrow \mathtt{x} \mid \mathtt{f}(\mathtt{x}_1 : \mathtt{e}_1,\ \ldots,\ \mathtt{x}_n : \mathtt{e}_n,\ \mathtt{size}) \Uparrow \mathtt{y}_1,\ \ldots,\ \mathtt{y}_m \uparrow \mathtt{x} \\ & \mid \mathtt{cty\ e\ :=\ e;\ g} \uparrow \mathtt{x} \mid \mathbf{let}(\mathtt{n})\text{*}\ \mathtt{x\ =\ g;\ g} \uparrow \mathtt{x} \mid \mathbf{assert}(\mathtt{lc});\ \mathtt{g} \uparrow \mathtt{x} \mid \ldots \end{aligned}$$

The first annotation is adding a "$\uparrow$ `x`" tag to all the constructs that can cause backtracking, signifying that `x` was the previous variable to be generated. If it is the first **`let*`** in a generator, we use a special marker variable $\perp$, which backtracks to the caller.

The second annotation is on calls to other generators. It records the variables ($\Uparrow \mathtt{y}_1, \ldots, \mathtt{y}_m$) that appeared on if-conditions on the path to this call. This one is used when the stack depth limit or size is exceeded, to know which variables if regenerated could avoid a recursive call.

4 Implementation

We implement a C runtime for the elaborated form of our DSL, where each backtracking point corresponds to a label and each exception thrown is a "goto" to the previous label. This includes a runtime library to manage backtracking information, e.g., which variables were relevant to a failing logical constraint. For our PRNG, we use 64-bit Mersenne twister [56, 63]. To implement `uniform` we use Lemire [49]'s method for sized integer generation and our built-in ranged generators.

To simplify memory management, all allocation and metadata for Bennet is stored in stacks, so that, when backtracking to some variable `x`, we can reclaim memory simply by resetting the stack pointers to their state before `x` was generated.

To allow failing counterexamples to be presented to the user, we compile preconditions into C functions that, given the arguments to a function, synthesize a C program that produces the failing input; this program is displayed to the user to help them understand the failure. This process turns out to be surprisingly challenging: due to the non-determinism of the heap, identifying which invariants of the memory to preserve is non-trivial.

5 Evaluation

We designed an evaluation to answer the following research questions (RQs):

- **RQ1:** Can Bennet generate tests satisfying preconditions in existing CN-annotated C code?
- **RQ2:** How does Bennet's bug-finding effectiveness compare to AFL++ on complex recursive data structures and critical code idioms?
- **RQ3:** How important are our two major optimization passes?

To address RQ1, we evaluated Bennet on examples from the CN tutorial [75] (5.1). To address RQ2, we measured its performance on complex data structures and critical code examples in a bug-finding study (5.2). To address RQ3, we performed an ablation study, disabling one or the other of the two major optimization passes—lifting constraints to our DSL and reordering when values are generated for variables. The results show that a significant number of tasks that are easily solved by Bennet are no longer solved if we disable one of these optimizations (5.3).

We performed all our benchmarks on a MacBook Pro with an M3 processor and 36GB of RAM. We used 20 trials for all our studies to account for variability. We used a Mann-Whitney U test—a non-parametric test [52] recommended for evaluating randomized algorithms [6]. The results show that, with $\alpha = 0.01$, Bennet performs (statistically) significantly better across most bug-finding tasks that both tools were able to solve, both in terms of number of inputs and time taken to find a bug. There were no tasks that AFL++ could solve that Bennet could not, nor did AFL++ perform statistically significantly better on any.

5.1 Examples from the CN Tutorial

Our first experiment asks whether Bennet can generate initial memory states satisfying the preconditions of examples from the CN tutorial. The tutorial includes 66 small examples exercising many of CN's features, such as pointers, arrays and recursive predicates, plus four larger case studies—an imperative queue, singly and doubly linked lists, and a simple airport runway scheduler. We reserved the runway scheduler for our bug-finding study and consider the other three here.

For the 66 small examples and the three remaining case studies, we found that Bennet can successfully generate 100 passing inputs for (almost) every correct example and find failing cases for every incorrect example, in fact finding two bugs in examples previously believed to be correct. (The first was introduced in unverified code added for the Fulminate evaluation, where there was an incorrect base case of `n == 0`, in a recursive function, when `n` was signed, instead of `n <= 0`. The second bug was in a lemma, a trusted fact used to describe something that CN cannot prove, used to prove correctness of pushing into the imperative queue case study.) A passing input is one that satisfies the precondition and whose result satisfies the postcondition.

Bennet can discard (give up on) generation attempts if it backtracks to the top level or exceeds a set time limit. By default, 1000 failed attempts are allowed. For the non-buggy functions that were able to be tested, none had more than five discards.

The lone function that could not be tested by Bennet was one whose precondition demanded that the list given as input was a specific list. Since lists are recursively defined, Bennet does not inline or unfold them at all, meaning that the generator was doing rejection sampling looking for one particular input in this case. Bennet tried 1000 generation attempts and gave up on all of them.

With this one exception, all of the examples from the CN tutorial were successfully generated and we were able to discover new bugs in some examples.

5.2 Bug-finding Case Studies

We selected six case studies to assess the bug-finding ability of Bennet. One of these comes from the CN tutorial, as mentioned above. One comes from a demonstration of Fulminate [39] and another from an open-source verification effort [29]. We wrote the last three case studies ourselves, to test Bennet's performance on more complex data structures.

To assess Bennet's bug-finding ability, we used Etna [83], a benchmarking tool for evaluating PBT frameworks. Etna allows us to automatically inject (manually described) faults into our case studies, run multiple trials, and summarize the results. We extended Etna to support testing C programs via Bennet and AFL++.

We adopted Etna's terminology, which focuses on various *tasks*, each a pair of a mutant (a wrong version of the code under test) and an executable specification embodying a correctness claim that does *not* hold for the mutant code. The tasks are grouped into *workloads*, which are our nine case studies. Each testing method is called a *strategy*; in our setting these are Bennet and AFL++.

For each task and strategy, we ran 20 trials with a timeout of 60 seconds, measuring the wall-clock time it takes to find the bug. If a given strategy was able to find the bug in all 20 trials, we say it *solves* the task. If it is able to find the bug in some of the trials, but not all of them, we say the strategy *partially solves* the task. We display the results of this experiment in Table 1.

We used Fulminate to instrument the function under test for both Bennet and AFL++. Bennet calls Fulminate and compiles the test code alongside synthesizing the appropriate generators. The real time it takes to use the tool includes all these steps, so we included all of them in our measurements. In the spirit of fairness, we also included the time to instrument and compile the function under test in the reported times for AFL++. For context, instrumenting, building and running the program `int main() {return 0;}` took 0.09 seconds on average across 20 runs.

Since testing with AFL++ requires a fuzzing harness, each task in this evaluation is accompanied by a hand-written harness. These harnesses convert bytes from the standard input into a valid memory shape, check the precondition of the function (exiting if it fails), and call the function under test. AFL++ is a mutation-based fuzzer that takes "seed" inputs and mutates them into new inputs. We give AFL++ an empty seed to start.

Our results show that Bennet outperforms AFL++ across the board, finding more bugs faster. We performed a Mann-Whitney U test with $\alpha = 0.01$, finding that Bennet's average number of inputs and average time to solve a task were (statistically) significantly better (lower) for 74 and 66 of the 74 tasks that AFL++ solves, respectively. AFL++ did not outperform Bennet on any of the tasks.

Fuzzers like AFL++ are typically meant to test full applications for hours, being evaluated for 24 hours or more [41], so these results are not entirely surprising. We chose AFL++ due to its use of random input generation.

Critical Code. The first three workloads consider small pieces of critical code—the sort of code for which testing is especially important, due to their the high cost of failure. One is a ring queue whose

Table 1. Bug-finding study results, with lines of code (LoC) for each workload. For the workloads with multiple versions of the predicate (Sorted List, BST, and AVL), the average is given. For tasks that were completely solved, the mean and standard error of the number of inputs and time taken to find the bugs are provided.

Workload	Tasks	LoC	Bennet			AFL++		
			Solved (Partial)	Inputs	Time (s)	Solved (Partial)	Inputs	Time (s)
Ring Queue	4	203	4 (0)	2.81 (± 0.52)	0.10 (± 0.00)	3 (1)	5253 (± 764)	15.37 (± 2.89)
Runway	6	293	6 (0)	20.11 (± 4.05)	0.50 (± 0.01)	6 (0)	18615 (± 2404)	1.20 (± 0.13)
MKM	2	871	2 (0)	31.38 (± 8.10)	0.37 (± 0.00)	2 (0)	7181 (± 808)	0.89 (± 0.11)
Sorted List	6	119	6 (0)	7.83 (± 1.70)	0.10 (± 0.01)	6 (0)	2968 (± 56)	0.32 (± 0.00)
BST	32	602	32 (0)	7.72 (± 1.51)	0.20 (± 0.00)	32 (0)	8555 (± 1156)	1.00 (± 0.12)
AVL	26	800.5	26 (0)	7.99 (± 1.46)	0.60 (± 0.10)	25 (1)	12757 (± 3031)	1.48 (± 0.35)

mutants come from a demonstration of Fulminate [39] and whose predicate involves constraints on valid indices into the underlying buffer. The second is an airport runway management simulation, with many logical constraints on whether departures or arrivals are being allowed at a given time, how many planes are waiting, etc. The mutants here involve altering a validity predicate used to determine valid states, removing bounds or assertions of disjointness. The third is a real-world mission key management system (MKM), which centralizes the management of secret keys, with actions required to receive a key specified against a finite state machine. The mutants are alternate implementations that cause invalid state machine transitions. The correct versions of all three workloads have been formally verified with CN.

These workloads are simpler in their implementation than the data-structure examples described above, but they involve complex logical constraints on their arguments. AFL++ was able to solve all of the associated tasks, except one. The only task it failed to find, it failed in 3 instances, which required a full queue to be generated. In contrast, Bennet was able to solve all 12. The difference in number of inputs was massive, with Bennet using 1896x (Ring Queue), 1484x (Runway), and 358x (MKM) fewer inputs. Bennet was also 165x, 2x, and 2x faster than AFL++ across these workloads.

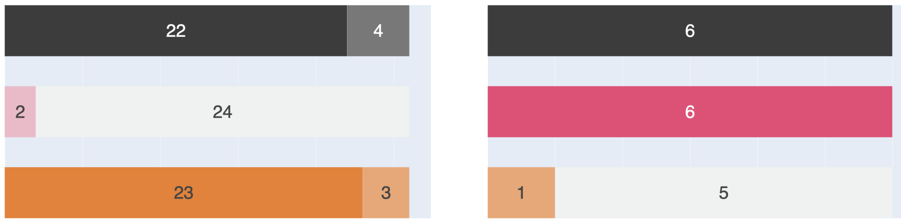
Recursive Data Structures. The final three tasks consider recursive data structures with logical constraints. None of these examples have been formally verified with CN. Instead, we write multiple candidate predicates for each, reflecting multiple ways an author might express their intentions. We explain the differences in detail for each case study below.

For the sorted list, there are three versions of the predicate. One is the standard list predicate, except that, after each recursive call, it asserts that the current node's value is less than or equal to the head of the tail, provided it exists. Another version passes the previous value as an argument to the predicate, wrapping it in an option. The final version passes the previous value as an integer.

The BST and AVL trees both have two versions of their predicates. There is one where the ordering is checked after the recursive call and one where the valid range of keys is passed in as an argument, (with minimum and maximum integers at the top level). Each of these versions vary in difficulty for Bennet, with the main distinction being between the constraints depending on an argument or the return value of a recursive generator.

The BST mutants are a combination of adapting mutants from *How to Specify It!* [38], alongside some bugs that we discovered "organically" (by writing them by mistake) while writing the workloads. Bennet was 25x faster than AFL++ on 3 bugs related to deletion, which required a specific shape of tree. For example, one bug could only be detected when deleting a key that existed in the tree and had a leaf node as its right subtree.

The AVL tree's implementation is based on an example from VCDryad [69]. The AVL mutants are the BST mutants, plus some "organic" bugs. Four AVL tasks, that came from real bugs, have a



(a) AVL tree (b) Airport runway simulation

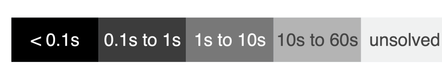
Fig. 7. Effectiveness of Bennet after disabling each of the major passes.
= Default (top), = No Reordering (middle), = No Lifting (bottom)

discrepancy between the specification and implementation on which rotation to perform when both could apply. This bug can only be seen when rebalancing or deleting, which constitute the four tasks—two for each version of the predicate. Generating a failing input for the rebalance function is not that difficult, requiring an unbalanced tree with a child whose children are the same height. Generating a failing input for deletion is more difficult, as you need a balanced AVL tree and a key whose removal results in a tree as above. AFL++ failed to detect this bug in 2 out of 20 trials in the version of the predicate that checks invariants after each call. However, it succeeded in every trial for the other version of the predicate. The more complicated control flow of checking invariants after recursive calls could explain the reduced effectiveness of coverage-guided fuzzing.

5.3 Ablation Study

We utilized Etna [83] once more to run the tests and generate the figures for our ablation study. Etna uses task bucket charts for its result, which we adopt in this section.

Task buckets group tasks according to how quickly a task is solved. If it is solved in less than 0.1 seconds, it goes into the leftmost bucket; tasks that aren't solved go in the rightmost bucket.



For the ablation study, we focused on the AVL tree and the runway simulation case studies. We chose the AVL tree since it is our most complicated recursive data structure, and we chose the runway simulation because it has the most complicated logical constraints. We evaluated all tasks with the reordering pass disabled and with the lifting pass disabled. The results appear in Figure 7

Looking at the AVL tree, we can see that 24 out of the 26 tasks could not be solved when the reordering pass was disabled. This is because the AVL tree involves costly recursive calls for its subtrees, which the reordering pass moves before the generation of the value at the current node. Another important factor is that AVL trees must be both sorted and balanced. The reordering pass moves logical constraints so they are evaluated as soon as possible. For the AVL tree, this means checking if the two subtrees are balanced before generating a value for the current node. This means the generator first finds two AVL trees whose height is within 2 of each other, before generating a value in between them. In contrast, if reordering is disabled, Bennet will generate the subtrees after the current value. After generating both of the subtrees, it will check if the new tree is sorted and balanced. If the left subtree's maximum is greater than the value at the current node, it will backtrack and regenerate the *right* subtree before backtracking further to the left subtree. Since generating an AVL tree is costly, it is better to regenerate the value at the current node instead.

It's interesting to note that the only effect of lifting on the AVL tree is splitting the balanced constraint and the sortedness constraint. Then, when generating a tree, it will check whether the tree is balanced before generating the integer value at the current node. Regenerating an integer is not costly, so avoiding this provides limited benefit. However, since lifting changed the generator, all the basic simplification passes get rerun. This overhead results in a net increase in the time taken, pushing one task into a slower bucket.

For runway simulation, 5 out of the 6 tasks could not be solved without the lifting pass. Lifting logical constraints, specifically disjunctions, means optimizing a dedicated generator for each case. The runway simulation describes valid states with fairly complex boolean formulas; lifting breaks apart their cases and builds optimized generators for each case. In fact, lifting logical constraints allows the resulting generator to be backtrack-free.

6 Related Work

Bennet applies PBT techniques to testing C programs. We discuss related work from both areas.

Coverage-guided Fuzzing. Popularized by tools like AFL [88] and libFuzzer [2], coverage-guided fuzzing (CGF) starts with a set of interesting example inputs, called seeds, which get mutated into new inputs. Any inputs that increase coverage are added to the set of interesting inputs. CGF is an active area with continuous advances from incorporating smarter mutation strategies [10, 48, 51, 73], improving seed prioritization [13, 14, 37, 76], and using better feedback [8, 87].

It can require significant domain knowledge to write fuzzing harnesses, which convert fuzzer-provided inputs into library calls. There has been significant work on synthesizing fuzz harnesses, including GraphFuzz [36], Hopper [20] and OGHarn [82]. A recent tool, AFLLive [28], converts fuzzer-provided inputs into structured inputs in a similar manner to JQF [65], per function.

JQF (specifically the Zest [66] algorithm) and Crowbar [84] take inputs generated by fuzzers and use them as the source of randomness for PBT-style generators. This guarantees test inputs satisfy validity properties or preconditions, while allowing the fuzzer to use coverage feedback to mutate the randomness. Since, internally, our implementation builds a list of the numbers sampled from our PRNG, which it can then treat as its source of randomness, it should be simple to set up our tool to use input from a fuzzer instead. This should be a useful addition, as it would enable feedback from the program under test, while preserving the precondition-satisfying guarantee of our generators.

Constraint Solving. Prior work has developed numerous constraint-solver-based testing tools for C, for example Euclide [35] and KLEE [15]. Many of our workloads utilize dynamic memory allocation and recursion, both of which Euclide forbids. As a result, Euclide is restricted to testing a much smaller class of programs. KLEE [15] and other concolic testing tools [31, 32, 81] are limited by their ability to symbolically model more complex programs. Using KLEE with Fulminate poses a dilemma: without instrumenting Fulminate's runtime library, KLEE is able to explore very little, but with instrumentation, it quickly runs out of memory. KLEE also requires harnesses to be written which establish which variables to symbolically model and what preconditions exist. If an mistake is made in the precondition assumptions in the harness, it will either lead to false positives (harness isn't strict enough) or false negatives (harness is too strict).

In terms of property-based testing, there are FocalTest [18, 19] and Target [80], which translate their preconditions into a constraint logic program and an SMT query, respectively. Prior tools on using separation logic for input generation, Java StarFinder [71, 72] and Concolic StarFinder [70], are also based on translating their preconditions into SMT queries. These tools target Java, a high-level language, and are limited to a decidable fragment of separation logic [47], using an SMT solver to generate precondition-satisfying inputs. This translation does not allow user control over the distribution nor give any guarantee of how the distribution will be shaped. While Bennet does not give the user control over individual choices, it would not be difficult to add support for annotating weights on the branches of if-statements. They also use a more restricted fragment of separation logic, supporting fewer specifications.

Narrowing-based Generators. The idea of *narrowing* [4, 50, 85] is that values are lazily instantiated when they are first used. It has been successfully applied in multiple projects for generating constrained test data [21, 25, 30, 44].

Claessen et al. [22, 23] took advantage of the laziness of Haskell to generate near-uniform distributions of constrained data, by applying their predicates to partially defined values. *Luck* [42] is a language which combines constraint solving and narrowing, defaulting to gathering constraints and allowing the user to annotate locations at which to generate a concrete value.

Our approach does not lazily instantiate values, nor gathers constraints symbolically, but rather chooses a specific location to generate a concrete value, prior to their first reference. While a bit similar to narrowing, it is fundamentally different, as we generate a value *before* calling a predicate, while narrowing would delay its instantiation until it is first used *within* the predicate. One of the main reasons for being explicit, about where values are instantiated, was to simplify control flow for the C implementation. Implementing narrowing would add significant complexity and bookkeeping to Bennet; it is left for future work. Adapting, Luck's constraint gathering abilities would require additional work past what is needed for needed narrowing.

Making Generators Better. We are not the first to consider how to take an existing generator and automatically improve it. There has been much work focused on improving generators by dynamically shaping their distribution during testing. Choice Gradient Sampling [34] targets wherever a choice is made between branches and skews its distribution, such that generated inputs pass later constraints more often. Unlike JQF [65] and Crowbar [84], which reuse existing fuzzers, FuzzChick [43] adapted the core loop of coverage-guided fuzzing to property-based testing, inspiring other efforts [57, 59]. RLCheck [77] used reinforcement learning to guide the generation towards valid inputs. All of these approaches provide ways to improve an existing generator, and all could be adapted to our setting. However, they act at test time, which is orthogonal to our objective of synthesizing more efficient generators to begin with.

In contrast, DRAGEN [58, 60] uses compile-time heuristics to predict and optimize the distribution of outputs towards a user-specified distribution. Instead of optimizing the distribution of our generators, we instead focused on producing valid inputs quickly. However, the branching processes technique of DRAGEN could potentially be adapted to Bennet to improve our generators' distributions.

7 Limitations and Future Work

We have shown how Bennet can efficiently generate random C memory states satisfying separation logic preconditions, allowing for efficient, effective random testing. However, Bennet has some limitations as it stands.

All of our optimizations after inlining non-recursive generators are intraprocedural, so the introduction of recursion can disable many of these optimizations. For example, an equality constraint that constrains the result of a recursive generator. In such cases, specialization will not be effective, and the likelihood of generating a satisfying input may be very low. In case of BSTs, suppose one wanted to add a lower bound for subtrees, by adding an assertion on the result of each recursive call. If the lower bound was higher than the max input size, Bennet would not be able to satisfy it, because sized generation limits the range of integers. Also, since specialization uses simple pattern matching, more complicated (in)equalities are not used to improve our generators.

Bennet's runtime uses a simple pointer generation method where newly generated pointers point to successive, adjacent memory cells. While this works well for many data structures, it can fail to find certain bugs related to aliasing. To support better generation, the runtime needs to be able to perform more bookkeeping to undo assignments if they later turn out to be to a wrongly aliased pointer. The memory management scheme, a simple bump allocator, has its own restrictions. If a pointer `p` is generated after the pointer `q`, `p` will always be greater than `q` (or `NULL`). Implementing a randomized allocator could help achieve a more balanced distribution of possible heap arrangements. For reproducibility, it should also be deterministic, similar to KDAlloc [79].

When the user of Bennet encounters an error, it will synthesize a C program that rebuilds the failing input, allowing the user to inspect it and potentially add it to a regression test suite. The current version of Bennet can synthesize counterexamples for all the inputs our generators create. However, since memory allocation is non-deterministic, the counterexample synthesizer must consider which invariants to preserve, such as alignment, ordering and the distance between pointers. For example, a function may have stricter requirements on pointer alignment than its type indicates. As we improve our implementation, e.g., with a randomized allocator, this synthesizer will need to be expanded significantly.

The fundamental loop of property-based testing is that inputs are generated, tested, and—if they fail—minimized, or "shrunk," as much as possible. Bennet does not currently support shrinking. We conjecture that failing inputs that have shorter paths through the precondition are generally "smaller" inputs. Prior work on minimizing the choices made during generation, in order to minimize test cases [33, 53], could possibly be applied.

CN also supports a form of *iterated resources*, which are used to represent arrays. Bennet currently only provides basic support for these—in particular, it is not able to efficiently satisfy constraints between different elements of an array. Such constraints are important in systems software like the pKVM buddy allocator, a case study from the original CN paper [74].

Bennet doesn't depend on CN's specification language, but rather the fragment of separation logic it desugars to. Many tools use basically the same fragment, where the use of existential quantifiers is restricted, so Bennet could be adapted to compiling their specifications into generators.

Bennet could also be adapted to other programming languages that have seen use in applying separation logic, such as Rust or Java. In both cases, there is more information about aliasing and the amount of memory that can be allocated, allowing for **`alloc`** to be simplified. However, our DSL and optimizations would still be useful for satisfying logical constraints. Moreover, even in memory-safe languages like Rust, unsafe code is still used, and this would require all of Bennet's allocation and assignment logic, as in C.

Acknowledgments

We thank our reviewers for helpful feedback, Iavor Diatchki for helping specify the BST case study, Harrison Goldstein for early discussions, and Peter Sewell, Rini Banerjee, Dhruv Makwana, and other members of the VERSE project for discussions and technical support. Our work was supported by the NSF under grant *SHF: Medium: Usable Property-Based Testing, NSF #2402449* and by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agencies (DARPA) under Contract No. FA8750-24-C-B044. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFRL and DARPA.

Data Availability Statement

There are three parts of the available artifact at Zenodo [3] for Bennet. First is the tool itself, whose up-to-date version is publicly available as part of the CN Github [1]. Second is the CN tutorial from the Fulminate evaluation, adapted to use Bennet. Third is a fork of Etna [83], which we added Bennet and AFL++ support to. It also includes the case studies and the scripts for calculating all of the numbers in Table 1 and Figure 7.

References

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

[2] 2025. libFuzzer. http://llvm.org/docs/LibFuzzer.html

[3] Zain K Aamer and Benjamin C. Pierce. 2025. *Artifact for Bennet: Randomized Specification Testing for Heap-Manipulating Programs.* Zenodo. doi:10.5281/zenodo.16937824

[4] Sergio Antoy, Rachid Echahed, and Michael Hanus. 2000. A Needed Narrowing Strategy. *Journal of the ACM (JACM)* 47, 4 (2000), 776–822.

[5] 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

[6] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250. doi:10.1002/stvr.1486

[7] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing telecoms software with quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang (ERLANG '06)*. Association for Computing Machinery, New York, NY, USA, 2–10. doi:10.1145/1159789.1159792

[8] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings 2019 Network and Distributed System Security Symposium*, Vol. 19. Internet Society, San Diego, CA, 1–15. doi:10.14722/ndss.2019.23371

[9] 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

[10] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *Proceedings of the 28th USENIX Security Symposium.* 1985–2002. https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko

[11] James Brotherston, Nikos Gorogiannis, Max Kanovich, and Reuben Rowe. 2016. Model checking for symbolic-heap separation logic with inductive predicates. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 84–96. doi:10.1145/2837614.2837621

[12] 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

[13] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 678–689. doi:10.1145/3368089.3409748

[14] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. doi:10.1145/2976749.2978428

[15] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, USA, 209–224.

[16] 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

[17] 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

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

[19] 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

[20] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. 2023. Hopper: Interpretative Fuzzing for Libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1600–1614. doi:10.1145/3576915.3616610

[21] Jan Christiansen and Sebastian Fischer. 2008. EasyCheck — Test Data for Free. In *Functional and Logic Programming*, Jacques Garrigue and Manuel V. Hermenegildo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 322–336.

[22] 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.

[23] 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

[24] 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

[25] Burke Fetscher, Koen Claessen, Michał Pałka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 383–405. doi:10.1007/978-3-662-46669-8_16

[26] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. https://www.usenix.org/conference/woot20/presentation/fioraldi

[27] 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

[28] Octavio Galland and Marcel Böhme. 2025. Invivo Fuzzing by Amplifying Actual Executions . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 709–709. doi:10.1109/ICSE55347.2025.00172

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

[30] 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

[31] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. doi:10.1145/1065010.1065036

[32] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (March 2012), 40–44. doi:10.1145/2093548.2093564

[33] Harrison Goldstein, Samantha Frohlich, Meng Wang, and Benjamin C. Pierce. 2023. Reflecting on Random Generation. *Proceedings of the ACM on Programming Languages* 7, ICFP (Aug. 2023), 322–355. doi:10.1145/3607842

[34] Harrison Goldstein and Benjamin C. Pierce. 2022. Parsing Randomness. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (Oct. 2022), 89–113. doi:10.1145/3563291

[35] 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

[36] Harrison Green and Thanassis Avgerinos. 2022. GraphFuzz: library API fuzzing with lifetime-aware dataflow graphs. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1070–1081. doi:10.1145/3510003.3510228

[37] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 230–243. doi:10.1145/3460319.3464795

[38] John Hughes. 2020. How to Specify It!. In *Trends in Functional Programming*, William J. Bowman and Ronald Garcia (Eds.). Springer International Publishing, Cham, 58–83.

[39] 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.

[40] 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

[41] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. doi:10.1145/3243734.3243804

[42] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: A Language for Property-Based Generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on*

Principles of Programming Languages (POPL '17). Association for Computing Machinery, New York, NY, USA, 114–129. doi:10.1145/3009837.3009868

[43] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage Guided, Property Based Testing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 181:1–181:29. doi:10.1145/3360607

[44] 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

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

[46] Leonidas Lampropoulos, Antal Spector-Zabusky, and Kenneth Foner. 2017. Ode on a Random Urn (Functional Pearl). In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, Oxford UK, 26–37. doi:10.1145/3122955.3122959

[47] Quang Loc Le, Makoto Tatsuta, Jun Sun, and Wei-Ngan Chin. 2017. A Decidable Fragment in Separation Logic with Inductive Predicates and Arithmetic. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 495–517. doi:10.1007/978-3-319-63390-9_26

[48] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 475–485. doi:10.1145/3238147.3238176

[49] Daniel Lemire. 2019. Fast Random Integer Generation in an Interval. *ACM Transactions on Modeling and Computer Simulation* 29, 1 (Jan. 2019), 3:1–3:12. doi:10.1145/3230636

[50] Fredrik Lindblad. 2007. Property Directed Generation of First-Order Test Data. *Trends in Functional Programming* 8 (2007), 105–123.

[51] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[52] Thomas W. MacFarland and Jan M. Yates. 2016. Mann–Whitney U Test. In *Introduction to Nonparametric Statistics for the Biological Sciences Using R*, Thomas W. MacFarland and Jan M. Yates (Eds.). Springer International Publishing, Cham, 103–132. doi:10.1007/978-3-319-30634-6_4

[53] David R. MacIver and Alastair F. Donaldson. 2020. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 13:1–13:27. doi:10.4230/LIPIcs.ECOOP.2020.13

[54] David R. MacIver, Zac Hatfield-Dodds, and Many Contributors. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (21 11 2019), 1891. doi:10.21105/joss.01891

[55] 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

[56] 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

[57] Agustín Mista. 2021. MUTAGEN: Faster Mutation-Based Random Testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 120–122. doi:10.1109/ICSE-Companion52605.2021.00053 ISSN: 2574-1926.

[58] Agustín Mista and Alejandro Russo. 2019. Generating Random Structurally Rich Algebraic Data Type Values. In *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*. IEEE, 48–54. doi:10.1109/AST.2019.00013

[59] Agustín Mista and Alejandro Russo. 2023. MUTAGEN: Reliable Coverage-Guided, Property-Based Testing using Exhaustive Mutations. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 176–187. doi:10.1109/ICST57152.2023.00025 ISSN: 2159-4848.

[60] Agustín Mista, Alejandro Russo, and John Hughes. 2018. Branching Processes for QuickCheck Generators. *ACM SIGPLAN Notices* 53, 7 (Dec. 2018), 1–13. doi:10.1145/3299711.3242747

[61] 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

[62] Rickard Nilsson. 2024. ScalaCheck. https://scalacheck.org/.

[63] 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

[64] Sam Owre. 2006. Random Testing in PVS. In *Workshop on Automated Formal Methods (AFM)*, Vol. 10. Seattle.

[65] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: coverage-guided property-based testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 398–401. doi:10.1145/3293882.3339002

[66] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. doi:10.1145/3293882.3330576

[67] Manolis Papadakis and Konstantinos Sagonas. 2011. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang (Erlang '11)*. Association for Computing Machinery, New York, NY, USA, 39–50. doi:10.1145/2034654.2034663

[68] Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. 2022. Computing correctly with inductive relations. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 966–980. doi:10.1145/3519939.3523707

[69] Edgar Pek, Xiaokang Qiu, and P. Madhusudan. 2014. Natural proofs for data structure manipulation in C using separation logic. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 440–451. doi:10.1145/2594291.2594325

[70] 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

[71] 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

[72] 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

[73] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2021. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (Sept. 2021), 1980–1997. doi:10.1109/TSE.2019.2941681 Number: 9 Conference Name: IEEE Transactions on Software Engineering.

[74] 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

[75] 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].

[76] Ruixiang Qian, Quanjun Zhang, Chunrong Fang, Ding Yang, Shun Li, Binyu Li, and Zhenyu Chen. 2024. DiPri: Distance-Based Seed Prioritization for Greybox Fuzzing. *ACM Trans. Softw. Eng. Methodol.* 34, 1 (Dec. 2024), 26:1–26:39. doi:10.1145/3654440

[77] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1410–1421. https://ieeexplore.ieee.org/document/9284117

[78] 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.

[79] Daniel Schemmel, Julian Büning, Frank Busse, Martin Nowack, and Cristian Cadar. 2023. KDAlloc: The KLEE Deterministic Allocator: Deterministic Memory Allocation during Symbolic Execution and Test Case Replay. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1491–1494. doi:10.1145/3597926.3604921

[80] Eric L. Seidel, Niki Vazou, and Ranjit Jhala. 2015. Type Targeted Testing. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer, Berlin, Heidelberg, 812–836. doi:10.1007/978-3-662-46669-8_33

[81] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. *SIGSOFT Software Engineering Notes* 30, 5 (Sept. 2005), 263–272. doi:10.1145/1095430.1081750

[82] Gabriel Sherman and Stefan Nagy. 2025. No Harness, No Problem: Oracle-guided Harnessing for Auto-generating C API Fuzzing Harnesses . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 775–775. doi:10.1109/ICSE55347.2025.00239

[83] 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

[84] 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

[85] 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).

[86] Scott Vokes. 2019. theft: property-based testing for C. https://github.com/silentbicycle/theft

[87] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. 1–15. https://www.usenix.org/conference/raid2019/presentation/wang

[88] Michał Zalewski. 2017. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/

[89] Litao Zhou, Jianxing Qin, Qinshi Wang, Andrew W. Appel, and Qinxiang Cao. 2024. VST-A: A Foundationally Sound Annotation Verifier. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 69:2069–69:2098. doi:10.1145/3632911

Received 2025-03-25; accepted 2025-08-12