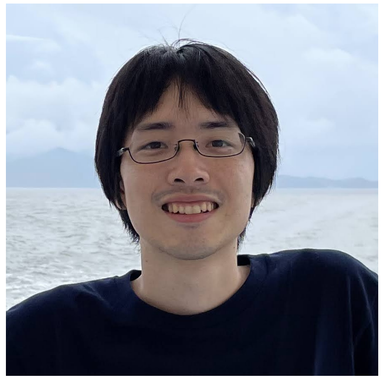
Gradual Assurance for Systems Software

A Separation-Logic Approach

Rini Banerjee

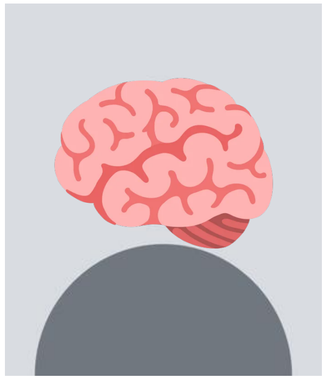
University of Cambridge

Iris Workshop ’26
ISTA, Austria
8 June 2026



Hiroyuki Katsura

David Kaloper-Meršinjak



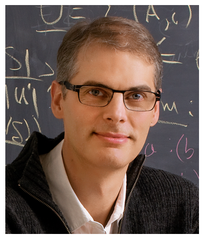
Zain Aamer

Dimitrios Economou

Dhruv Makwana

Kayvan Memarian

Christopher Pulte



Benjamin Pierce

Neel Krishnaswami

Peter Sewell

UNIVERSITY OF CAMBRIDGE

UNIVERSITY OF OXFORD

Penn
University of Pennsylvania

Fulminate: Testing CN Separation-Logic Specifications in C

RINI BANERJEE, KAYVAN MEMARIAN, DHRUV MAKWANA, CHRISTOPHER PULTE, NEEL KRISHNASWAMI, and PETER SEWELL, University of Cambridge, UK

Separation logic has become an important tool for formally capturing and reasoning about the ownership patterns of imperative programs, originally for paper proof, and now the foundation for industrial static analyses and multiple proof tools. However, there has been very little work on program *testing* of separation-logic specifications in concrete execution. At first sight, separation-logic formulas are hard to evaluate in reasonable time, with their implicit quantification over heap splittings, and other explicit existentials.

In this paper we observe that a restricted fragment of separation logic, adopted in the CN proof tool to enable predictable proof automation, also has a natural and readable *computational* interpretation, that makes it practically usable in runtime testing. We discuss various design issues and develop this as a C+CN source to C source translation, Fulminate. This adds checks – including ownership checks and ownership transfer – for C code annotated with CN pre- and post-conditions; we demonstrate this on nontrivial examples, including the allocator from a production hypervisor. We formalise our runtime ownership testing scheme, showing (and proving) how its reified ghost state correctly captures ownership passing, in a semantics for a small C-like language.

CCS Concepts: • **Theory of computation → Separation logic**; **Type theory**; **Program reasoning**.

Code-Specify-Test-Debug-Prove: Flexibly Integrating Separation Logic Specification into Conventional Workflows

ZAIN K AAMER*, University of Pennsylvania, USA
RINI BANERJEE*, University of Cambridge, UK
HIROYUKI KATSURA*, University of Cambridge, UK
DAVID KALOPER-MERŠINJAK*, University of Cambridge, UK
DIMITRIOS J. ECONOMOU, University of Cambridge, UK
KAYVAN MEMARIAN, University of Cambridge, UK
DHRUV MAKWANA, University of Cambridge, UK
NEEL KRISHNASWAMI, University of Cambridge, UK
BENJAMIN C. PIERCE, University of Pennsylvania, USA
CHRISTOPHER PULTE, University of Oxford, UK
PETER SEWELL, University of Cambridge, UK

We seek to enable more flexible use of rich specifications, in a variety of ways that smoothly extend conventional software development practice: we show how a single specification language based on separation logic, that can capture the subtle ownership disciplines of systems code, can be used for runtime assertion checking, for property-based testing, and for formal machine-checked proof—with each of these complementing and supporting the others. We demonstrate this on a challenging example: a component of a production hypervisor, running both stand-alone at user level and *in situ* in the hypervisor.

CCS Concepts: • **Software and its engineering → Formal software verification**; **Empirical software validation**; • **Theory of computation → Program reasoning**; **Automated reasoning**; **Separation logic**.

Additional Key Words and Phrases: C, specification, runtime testing, property-based testing, separation logic, refinement types, verification, pKVM, Android

Motivation

We want bug-free software!

How can we achieve that?

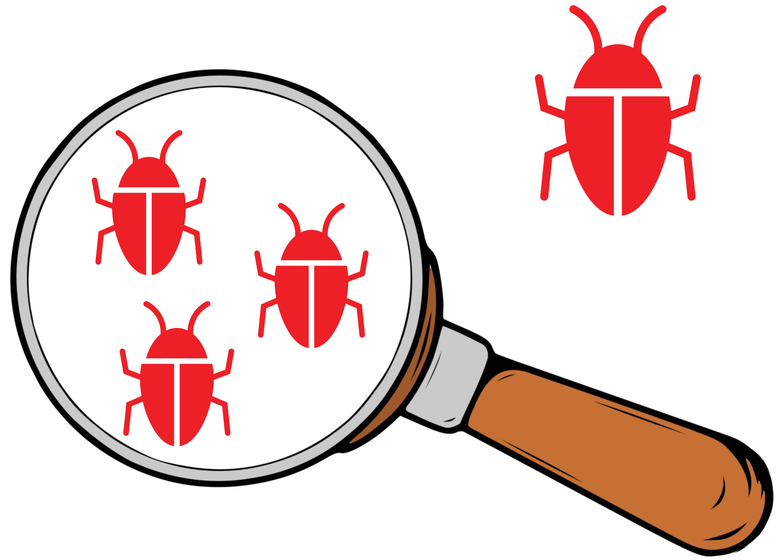
Specifications are implicit, but more generic

- Push-button static and dynamic analysis tools (e.g. Infer, Valgrind)

```
static size_t chunk_reclaimable(struct chunk_hdr *chunk,
                                struct hyp_allocator *allocator)
{
        unsigned long start, end = chunk_unmapped_region(chunk);

        /*
         * This should not happen, chunks are installed at a minimum distance
         * from the page start
         */
        WARN_ON(!PAGE_ALIGNED(end) &&
                (end - PAGE_ALIGN_DOWN(end) < chunk_size(0UL)));

        if (chunk_destroyable(chunk, allocator))
                start = (unsigned long)chunk;
        else
                start = PAGE_ALIGN((unsigned long)chunk + chunk_size(chunk-
>alloc_size));
```

*source code, **as-is***

run analysis tool



Motivation

We want bug-free software!

Specifications are explicit, but capture richer properties

How can we achieve that?

- Traditional formal verification with explicit specifications
  - code-first, or spec-first, or code and spec together

```
static size_t chunk_reclaimable(struct chunk_hdr *chunk,
                                struct hyp_allocator *allocator)
{
        unsigned long start, end = chunk_unmapped_region(chunk);

        /*
         * This should not happen, chunks are installed at a minimum
         * from the page start
         */
        WARN_ON(!PAGE_ALIGNED(end) &&
                (end - PAGE_ALIGN_DOWN(end) < chunk_size(0UL)));

        if (chunk_destroyable(chunk, allocator))
                start = (unsigned long)chunk;
        else
                start = PAGE_ALIGN((unsigned long)chunk + chunk_size
>alloc_size));
```

source code

time, ££, effort →

```
static size_t chunk_reclaimable(struct chunk_hdr *chunk,
                                struct hyp_allocator *allocator)
/*@
requires
        take HA = Cn_hyp_allocator_focusing_on(allocator, chunk);
ensures
        take HA_post = Cn_hyp_allocator_focusing_on(allocator, chunk);
        let C = HA.lseg.chunk;
        HA == HA_post;
        let start = (Cn_chunk_destroyable(HA_post.lseg) ?
                (u64)chunk : PAGE_ALIGN((u64)chunk + Cn_chunk_size((u64)C.allo
        let end = PAGE_ALIGN_DOWN(C.header_address + (u64)C.mapped_size);
        (start <= end ? end - start : 0u64) == return;
@*/
{
        unsigned long start, end = chunk_unmapped_region(chunk);

        /*
         * This should not happen, chunks are installed at a minimum distance
         * from the page start
         */
        WARN_ON(!PAGE_ALIGNED(end) &&
                (end - PAGE_ALIGN_DOWN(end) < chunk_size(0UL)));

        if (chunk_destroyable(chunk, allocator))
```

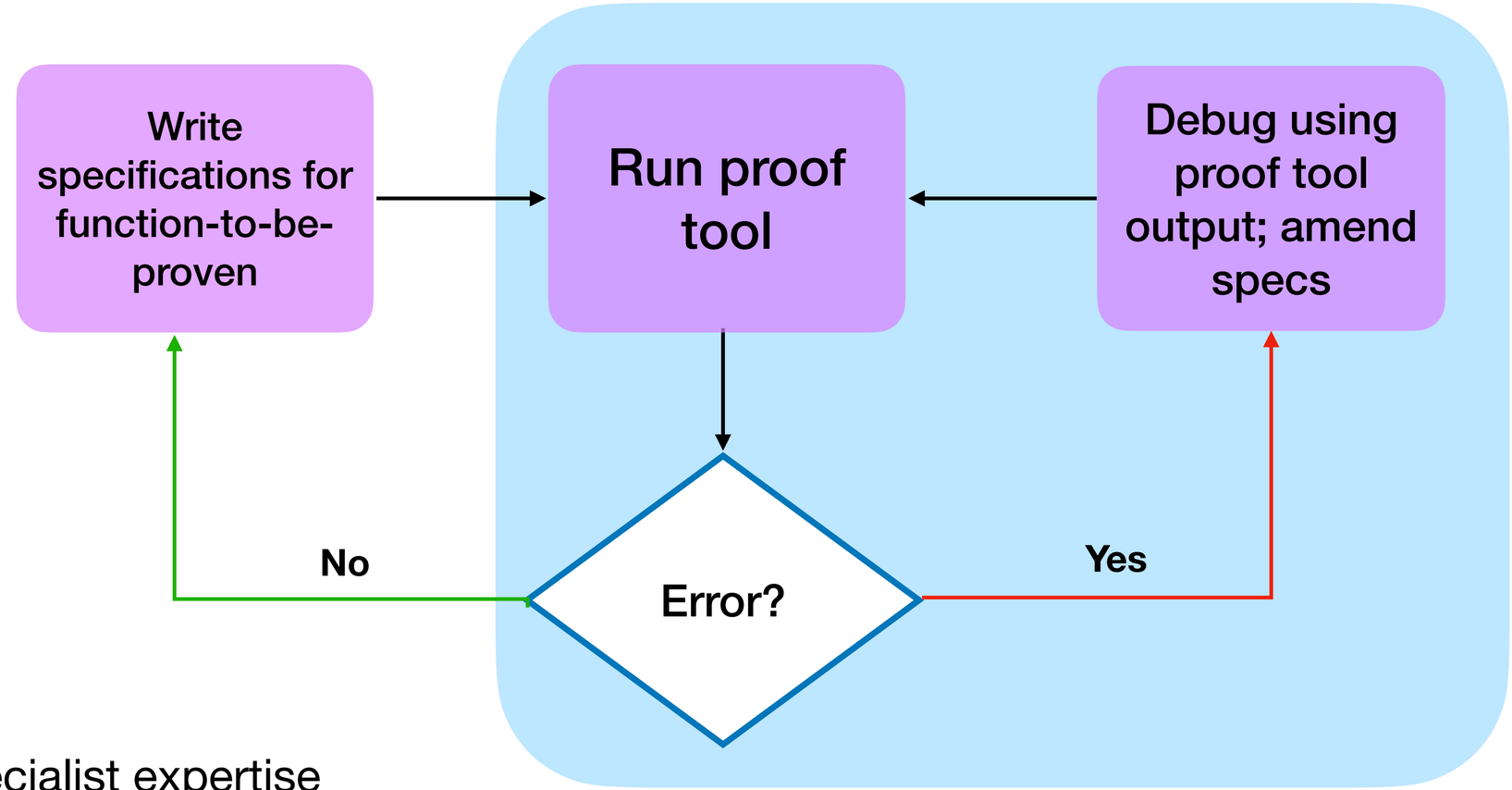
*source code, **annotated with specs***

Where to start?!

run proof tool →

Full functional correctness

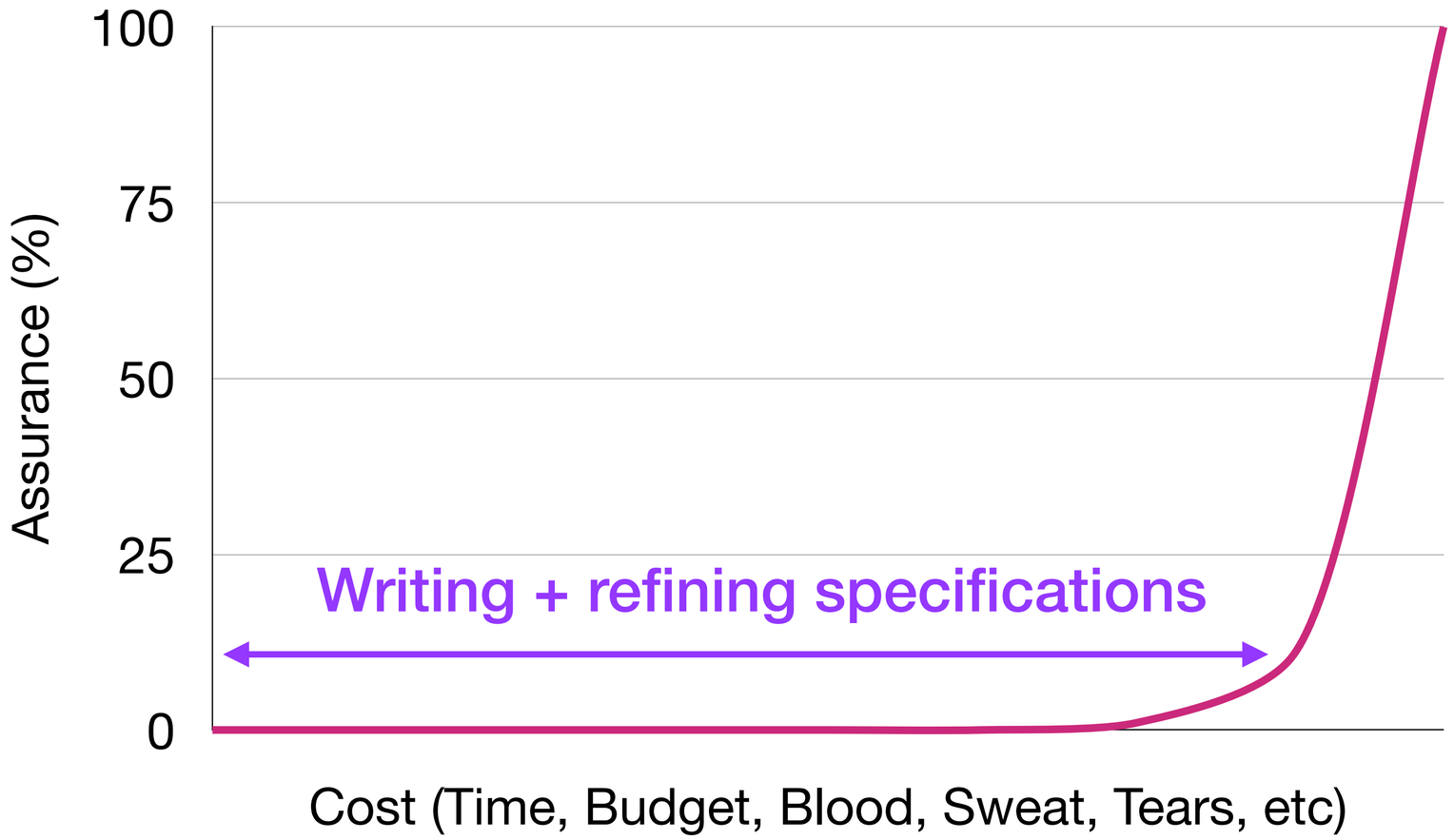
Problem 1 Writing specifications is **hard in itself!**

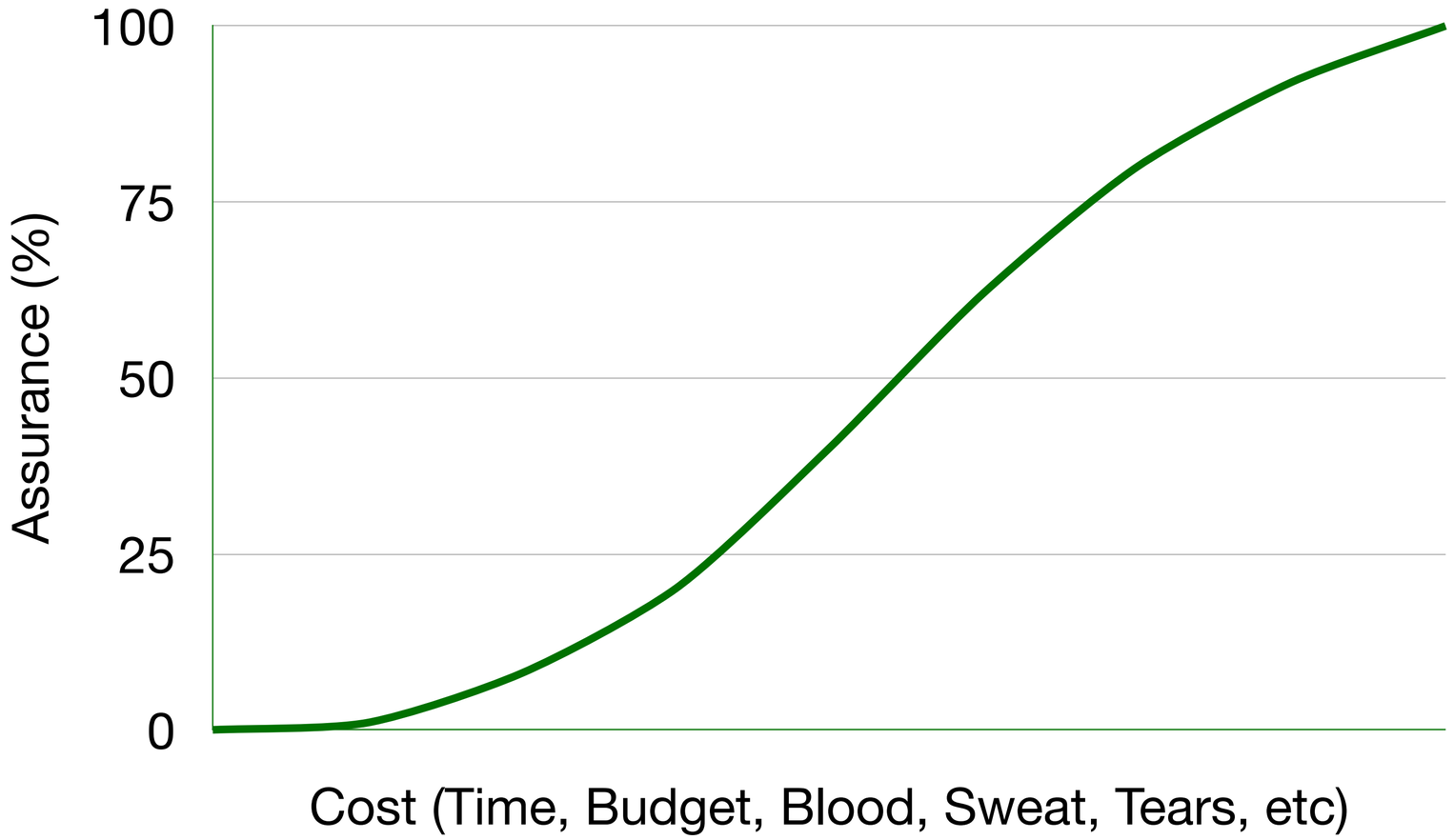


- Requires specialist expertise
- Requires <u>complete</u> specifications
- Proof attempts themselves can be time-consuming to run

$\Longrightarrow$ **slow feedback loop**

Problem 2 High assurance is totally **all-or-nothing**




100
75
50
25
0
Assurance (%)
Cost (Time, Budget, Blood, Sweat, Tears, etc)

Idea: *test* specifications

#1 Helps to guide the specification-writing process

Partial specifications can be checked at runtime ‘as you go’, flagging mistakes quickly

#2 **On-ramp** to full formal verification

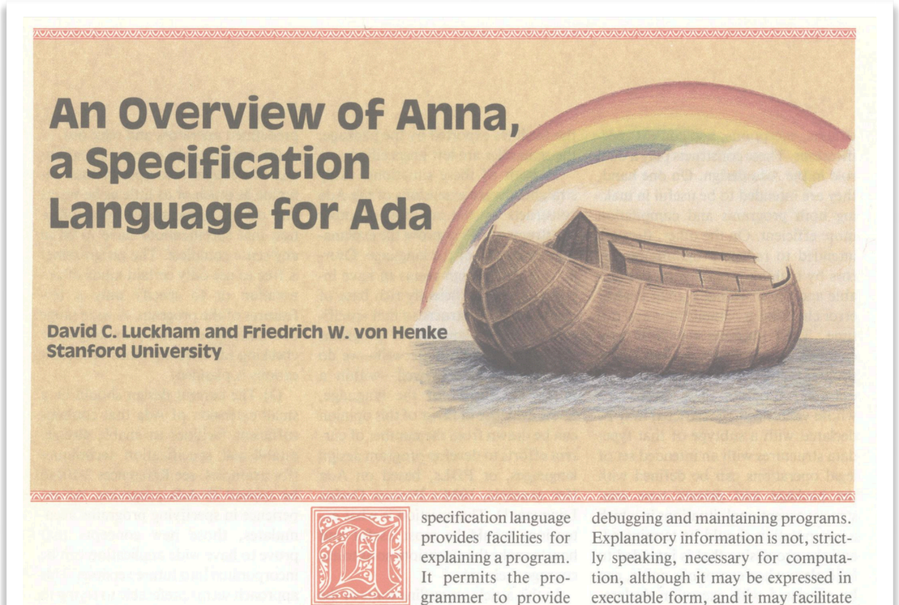
Gradual assurance, rather than “all-or-nothing”

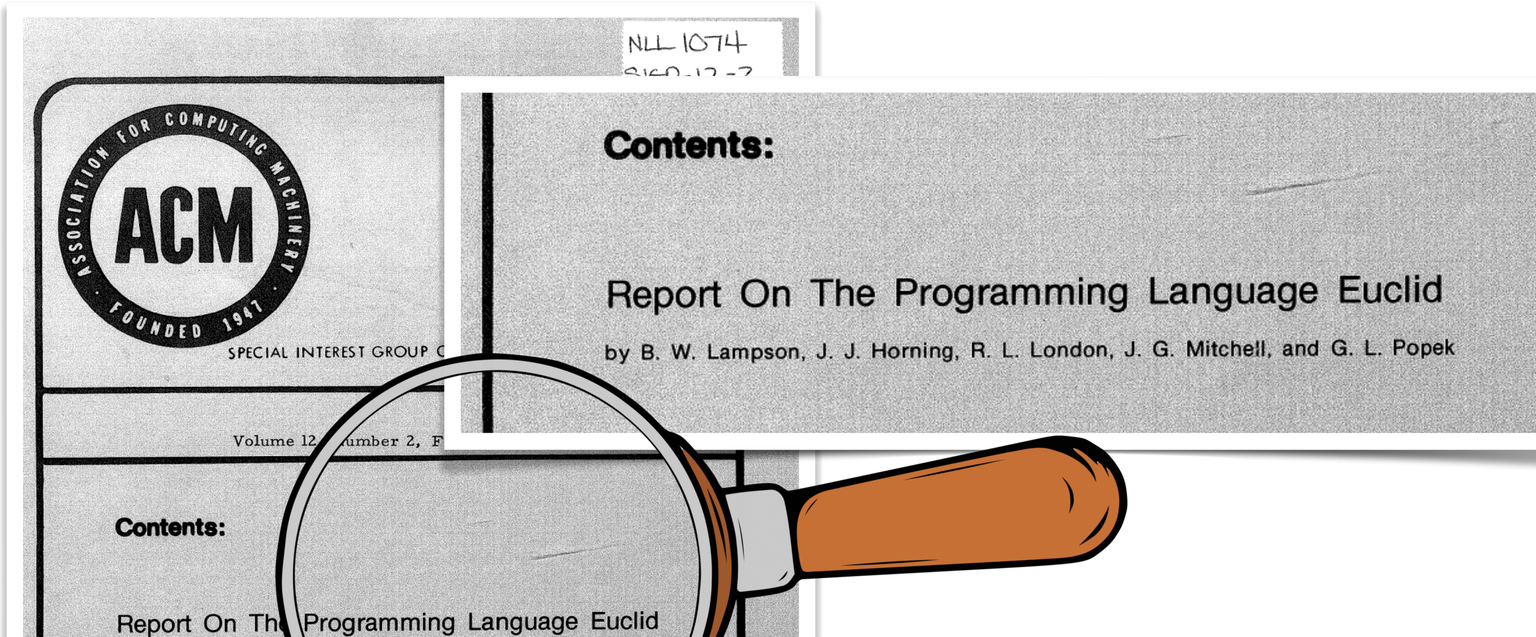
#3 Fits **test-debug cycle** engineers are familiar with

More familiarity => more likely to be adopted

Testing specifications

- The idea of testing specifications dates back to at least the 1970s
  - Seems underappreciated in the field of verification as a whole :-(
  - Historically, testing and verification communities have been largely disconnected
- Testing specifications to *make writing them easier* doesn't seem commonplace





We think it should be!

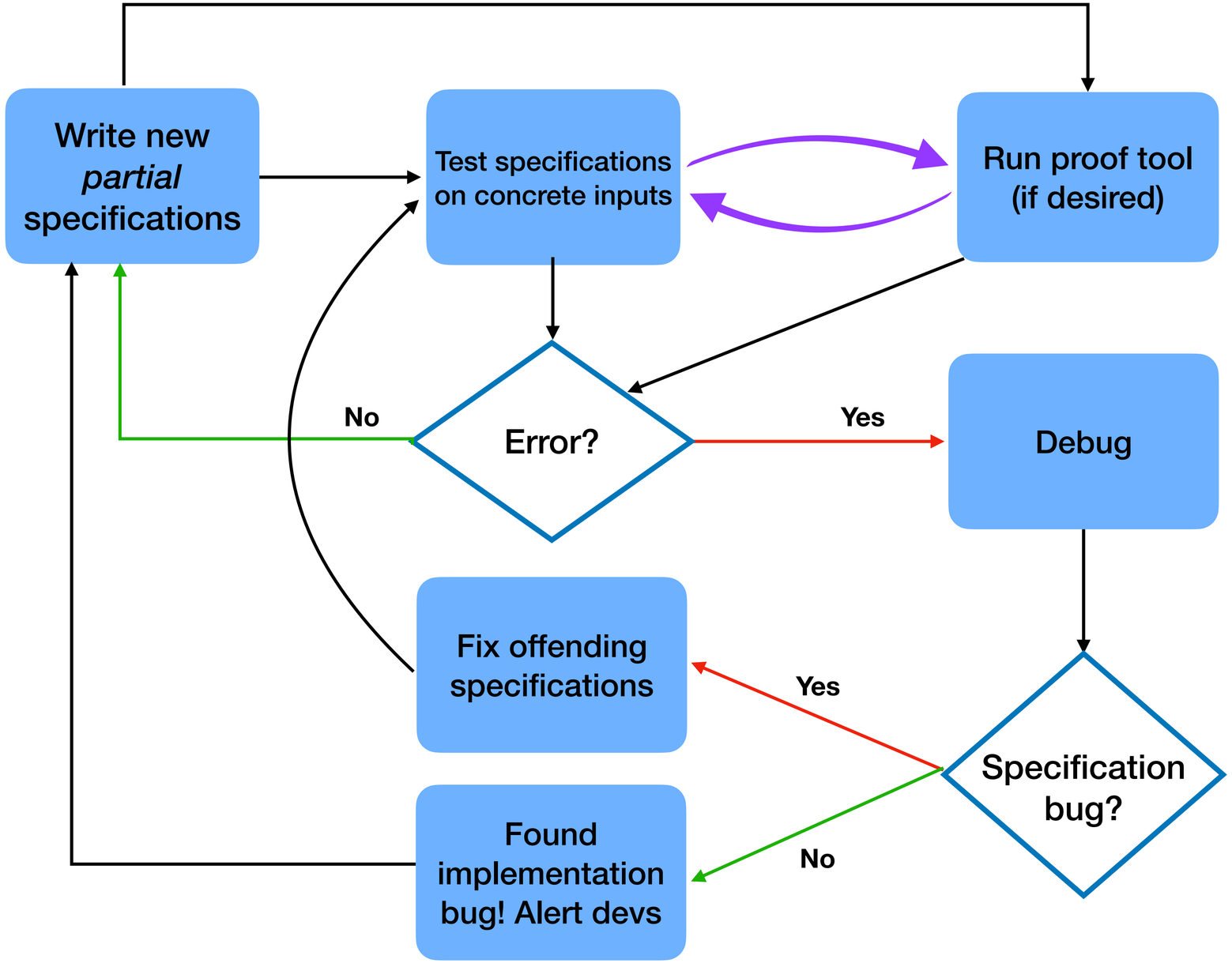
We propose an ***incremental*** workflow that:

✓ builds upon the standard **code-test-debug** cycle

✓ puts **specification-testing and -debugging** at its core

✓ combines **testing and proof** under a **single framework**

Code-Specify-Test-Debug-Prove

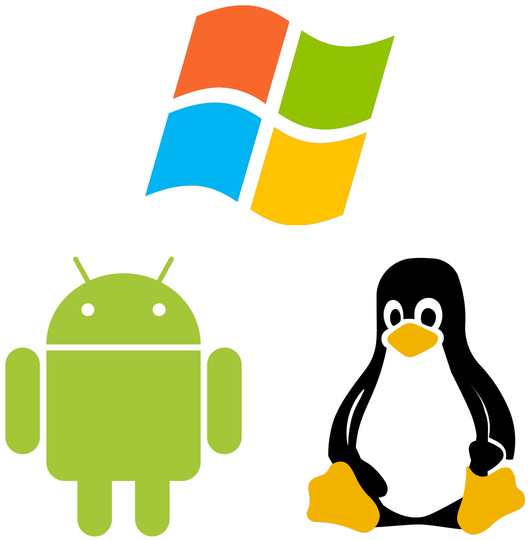


Code-Specify-Test-Debug-Prove

Code-Specify-Test-Debug-Prove

for **systems software written in C**

using **separation logic** to capture its complex ownership patterns



$$p \mapsto v$$

$$A * B$$

CN separation-logic testing and verification toolchain

The CN toolchain

Fulminate, Bennet, Darcy and the CN proof tool

The CN specification language

$$
\begin{aligned}
&\texttt{IntList}(p, xs) = \\
&\quad (p = \texttt{NULL} \wedge \\
&\quad\ xs = \texttt{nil}) \\
&\vee \\
&\quad (\exists hd, q, tl\,. \\
&\quad\ p \mapsto (hd, q) * \\
&\quad\ \texttt{IntList}(q, tl) \wedge \\
&\quad\ xs = \texttt{cons}(\texttt{hd}, \texttt{tl}))
\end{aligned}
$$

The CN specification language

$$\texttt{IntList}(p, xs) =$$
$$(p = \texttt{NULL} \wedge$$
$$xs = \texttt{nil})$$
$$\vee$$
$$(\exists hd, q, tl\,.$$
$$p \mapsto (hd, q) *$$
$$\texttt{IntList}(q, tl) \wedge$$
$$xs = \texttt{cons}(\texttt{hd}, \texttt{tl}))$$

```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p)
{
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

This looks executable!

Fulminate

- Translates CN specifications into C runtime checks, instrumenting the source
  - Implemented as a source-to-source translation in OCaml

```
signed int increment(signed int i)
/*@ requires i < power(2, 31) - 1 @*/
/*@ ensures return == i + 1 @*/
{
  return i + 1;
}
```

C file + CN annotations

Source-to-source translation

```
signed int increment(signed int i)
{
  check_increment_precondition(i);
  signed int ret_value = i + 1;
  check_increment_postcondition(i, ret_value);
  return ret_value;
}
```

C file instrumented with checks

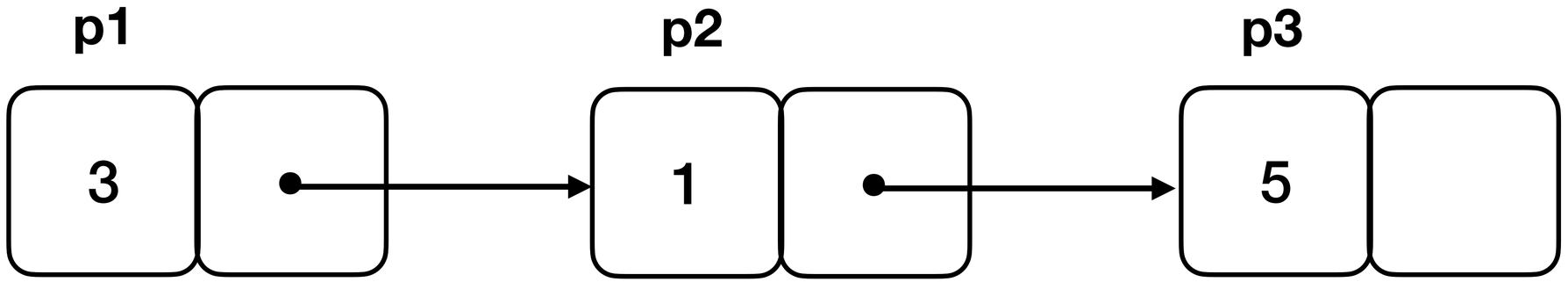
My PhD work :-)

Fulminate ownership checking

- Makes use of CN's restrictive syntax: $p \mapsto v$ becomes **`take v = Owned(p)`**
  - Makes executing separation logic specs computationally feasible by disallowing arbitrary quantification over addresses
  - Fulminate checks ownership of p and, if successful, dereferences p to get v
- Maintains a **global ownership ghost map**, tracking ownership of stack and heap addresses by mapping and comparing them to the call stack depth
- Performs ownership checks within dynamic calls to **`Owned`** and C memory accesses

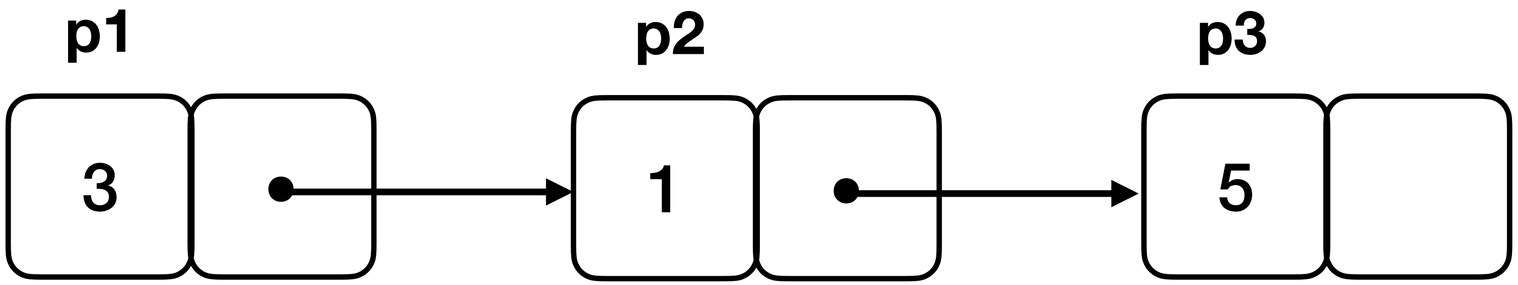
$$\texttt{IntList}(p, xs) =$$
$$(p = \texttt{NULL} \wedge$$
$$xs = \texttt{nil})$$
$$\vee$$
$$(\exists hd, q, tl\,.$$
$$p \mapsto (hd, q) *$$
$$\texttt{IntList}(q, tl) \wedge$$
$$xs = \texttt{cons}(\texttt{hd}, \texttt{tl}))$$

```
struct node
  {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```



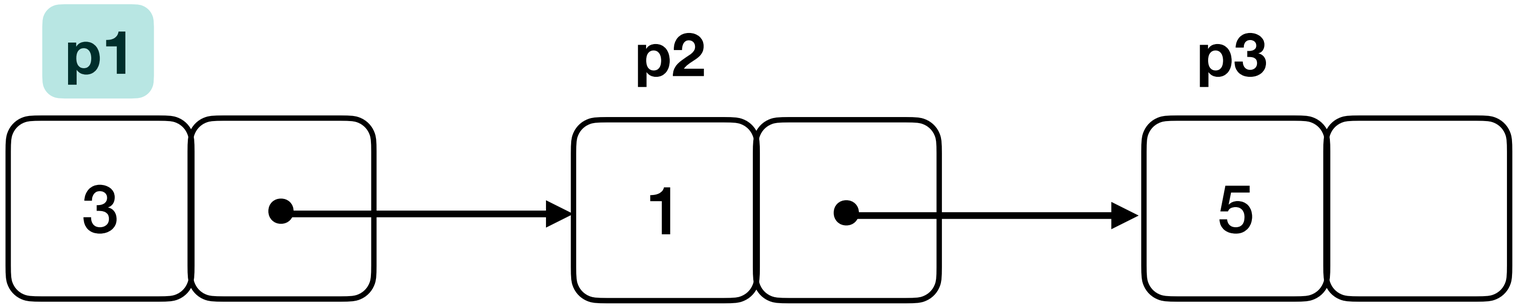
```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

```
/*@ requires take L1 = IntList(p1); @*/
```



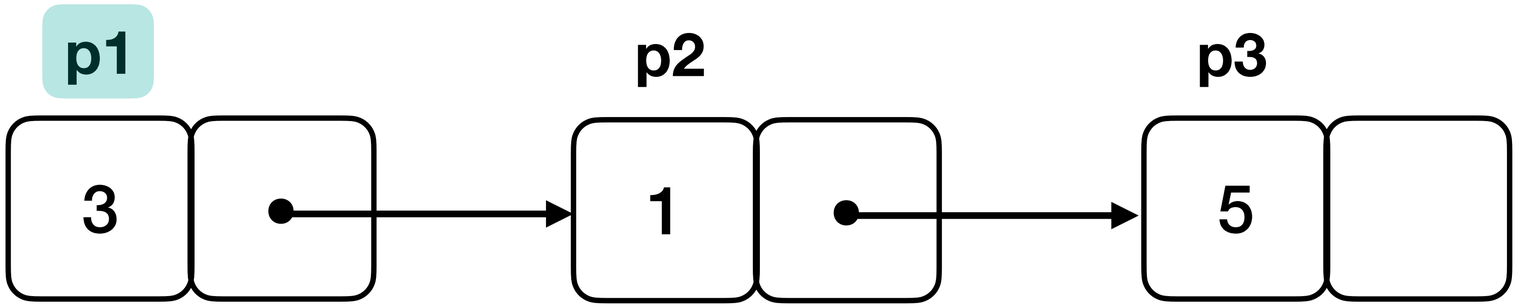
```
struct node
  {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

```
/*@ requires take L1 = IntList(p1); @*/
```



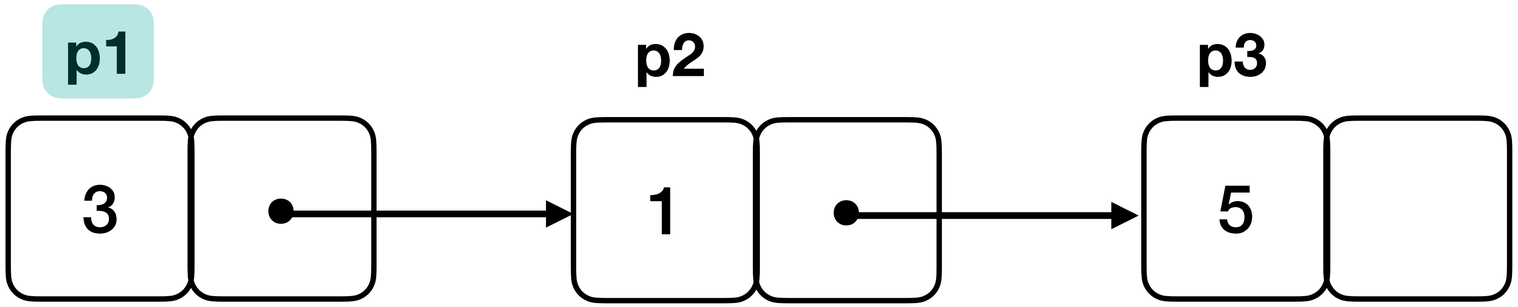
```
/*@ requires take L1 = IntList(p1); @*/
```

```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```



```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

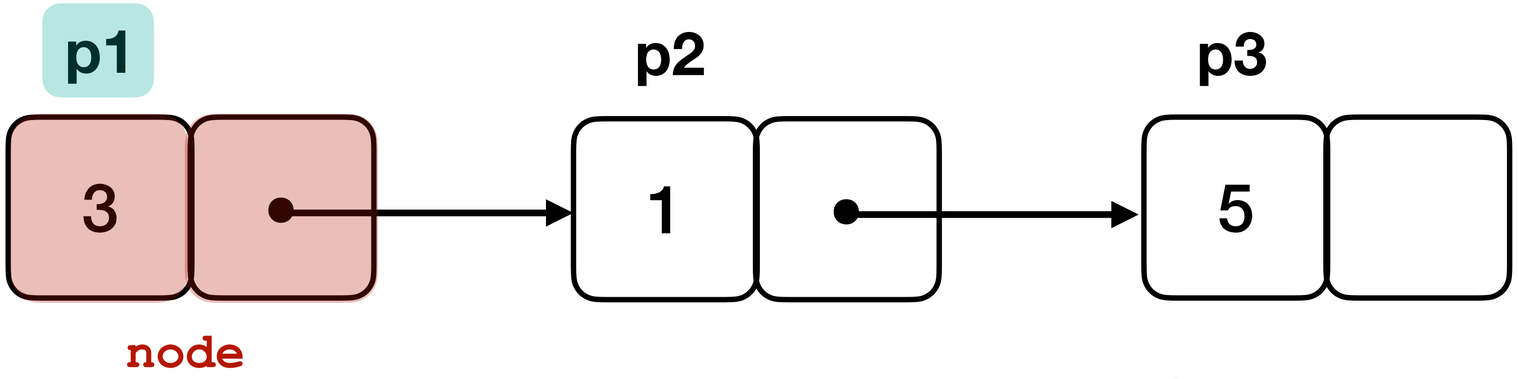
```
/*@ requires take L1 = IntList(p1); @*/
```

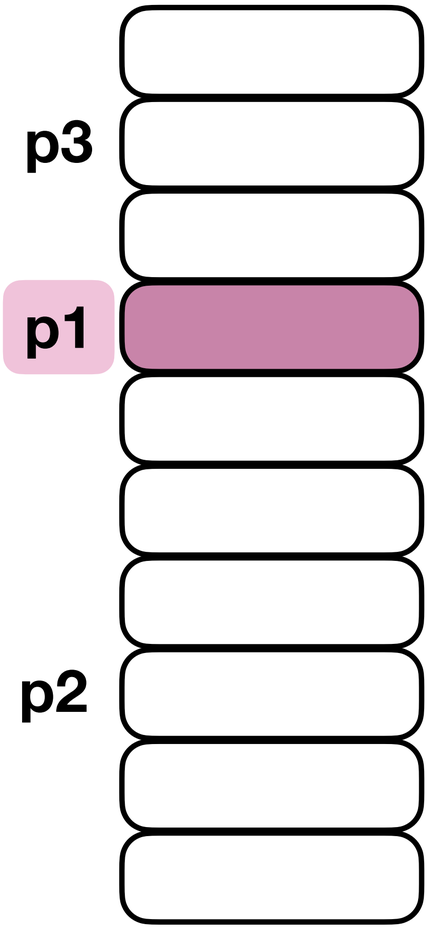


```
struct node
  {int x; struct node *next;};
```

```
/*@ requires take L1 = IntList(p1); @*/
```

```
predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

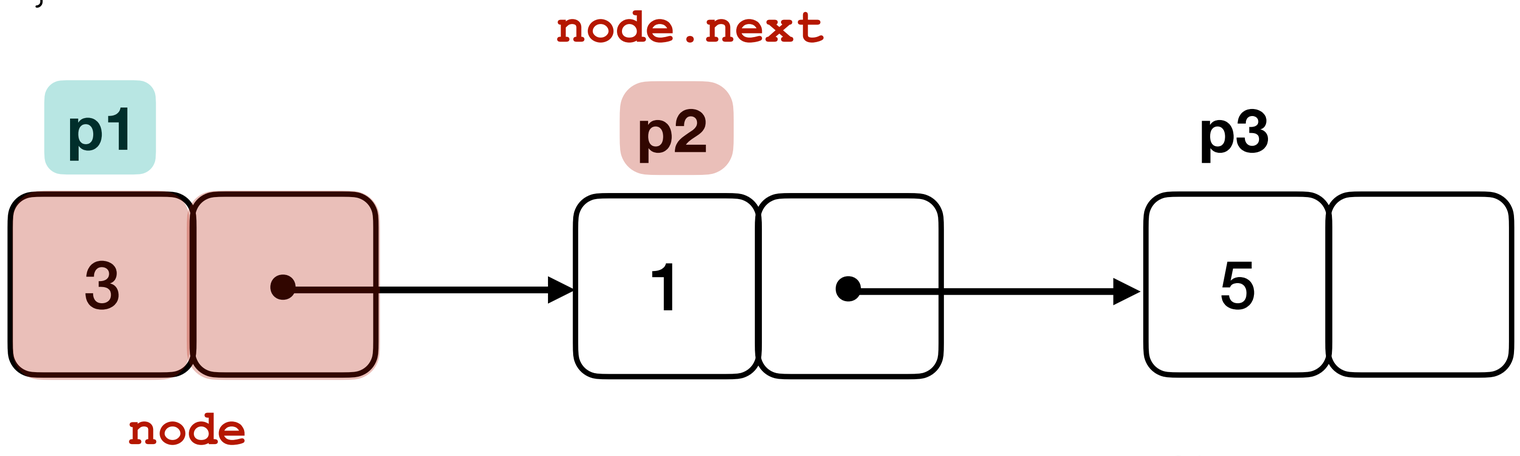


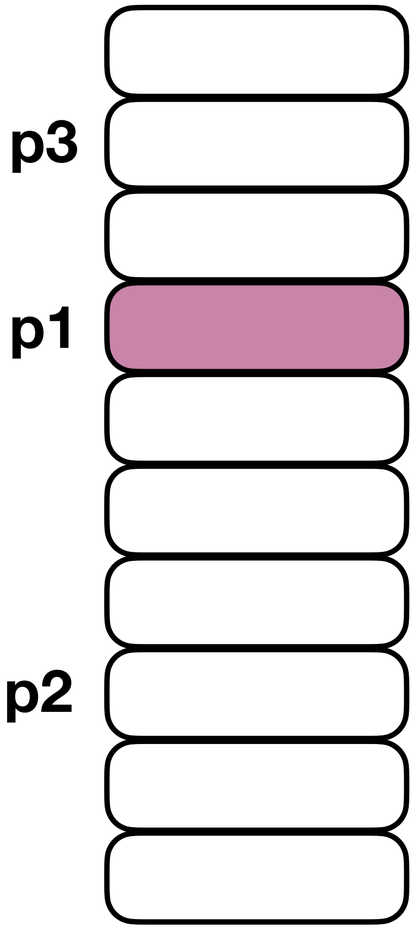


Global ownership ghost state

```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

```
/*@ requires take L1 = IntList(p1); @*/
```

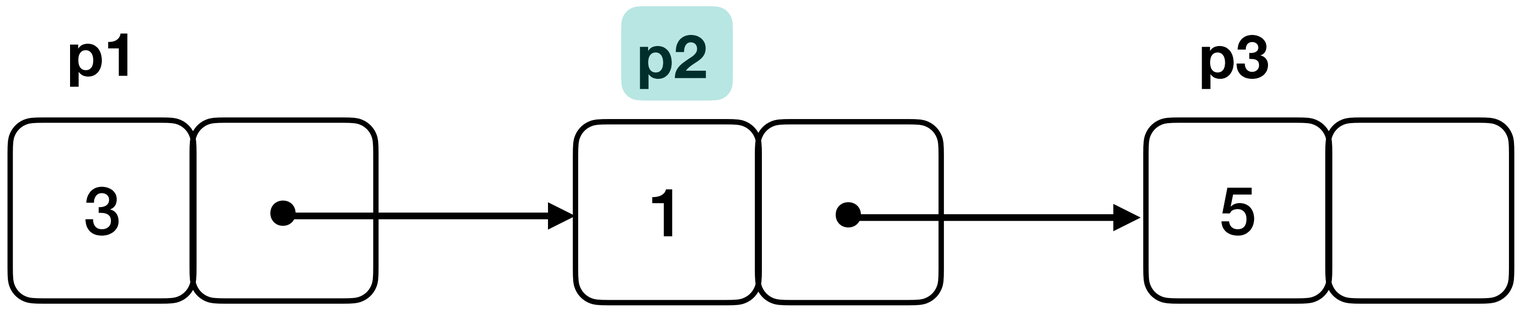


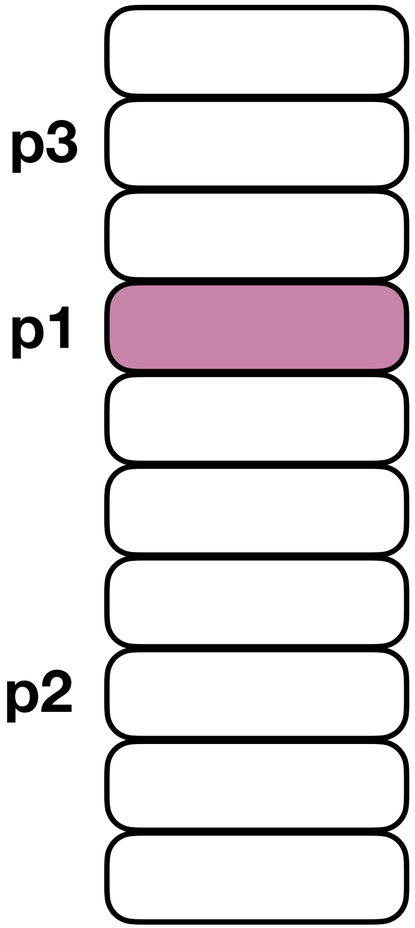


Global ownership ghost state

```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

```
/*@ requires take L1 = IntList(p1); @*/
```

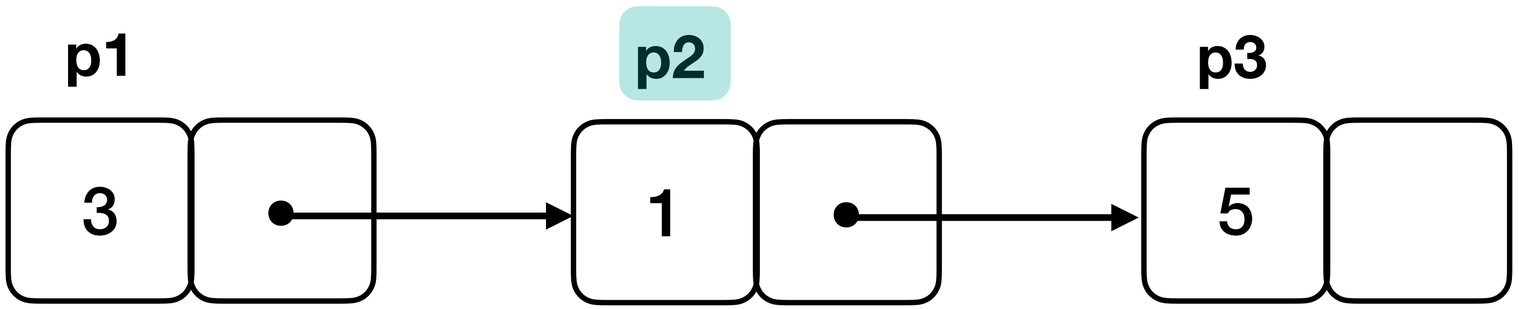


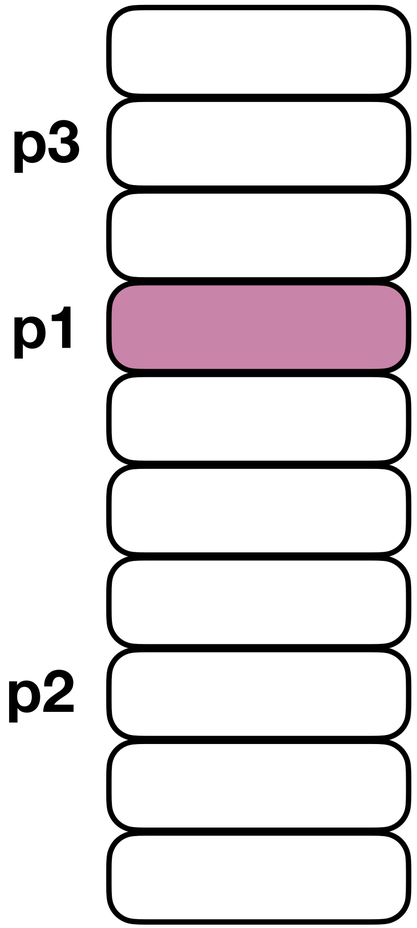


Global ownership ghost state

```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

```
/*@ requires take L1 = IntList(p1); @*/
```

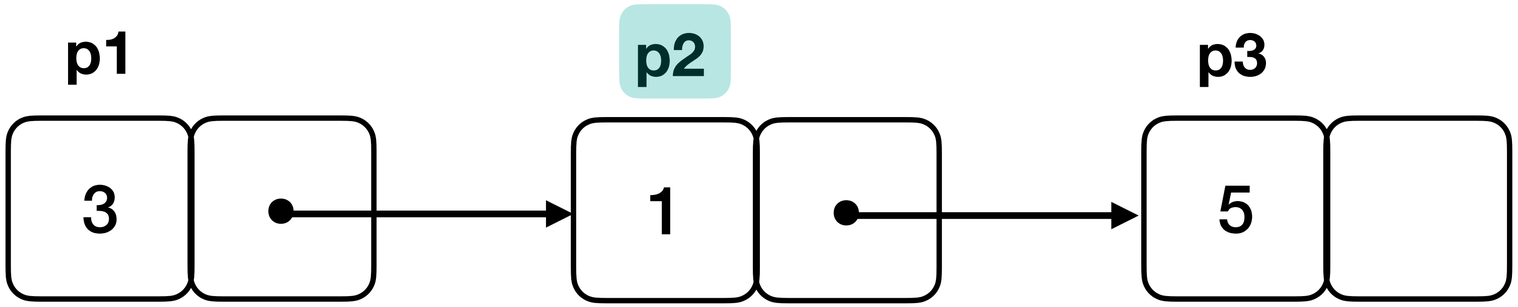


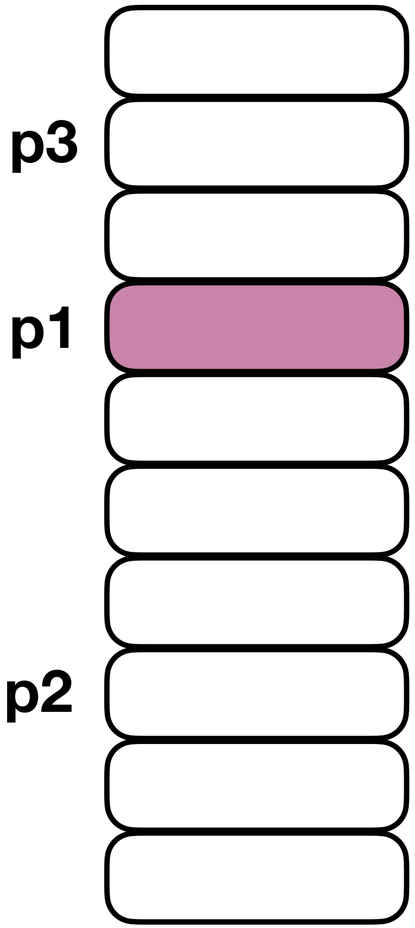


Global ownership ghost state

```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

```
/*@ requires take L1 = IntList(p1); @*/
```

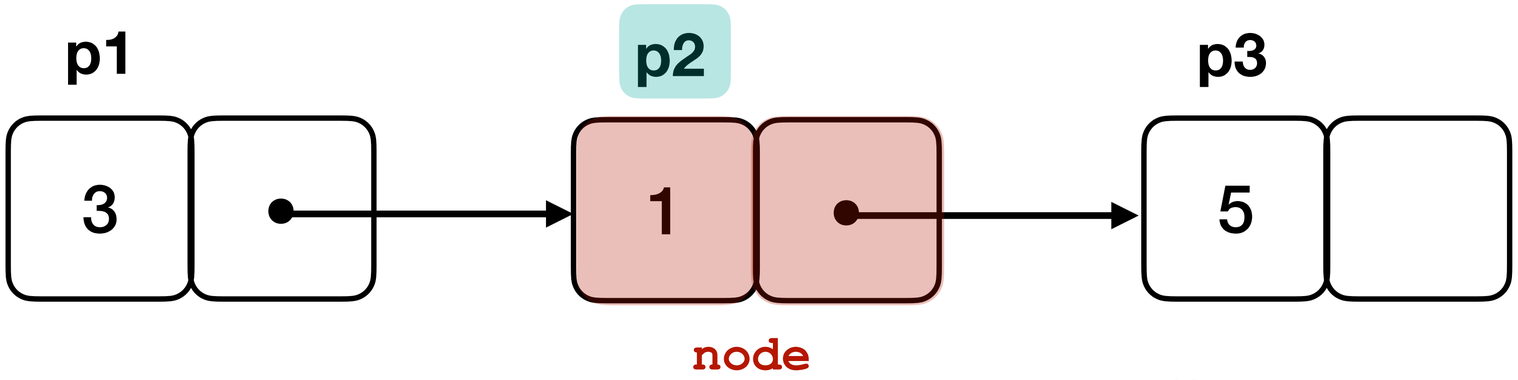


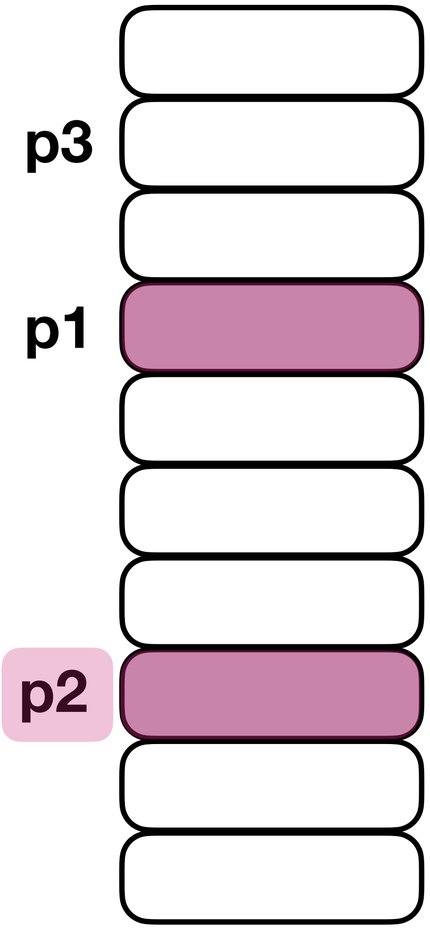


Global ownership ghost state

```
/*@ requires take L1 = IntList(p1); @*/
```

```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

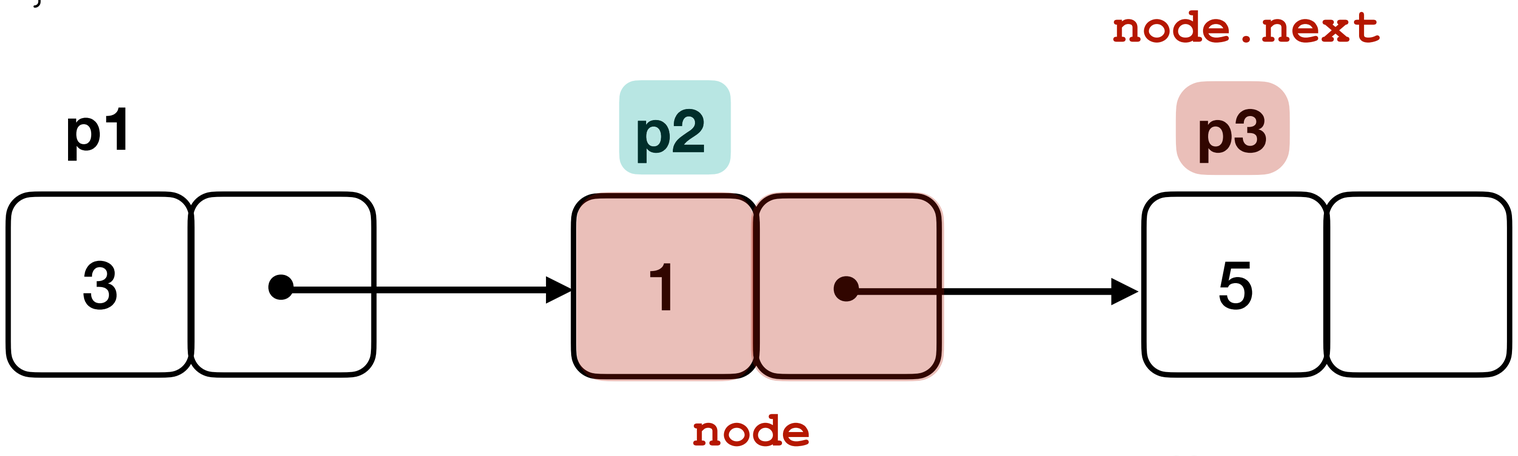


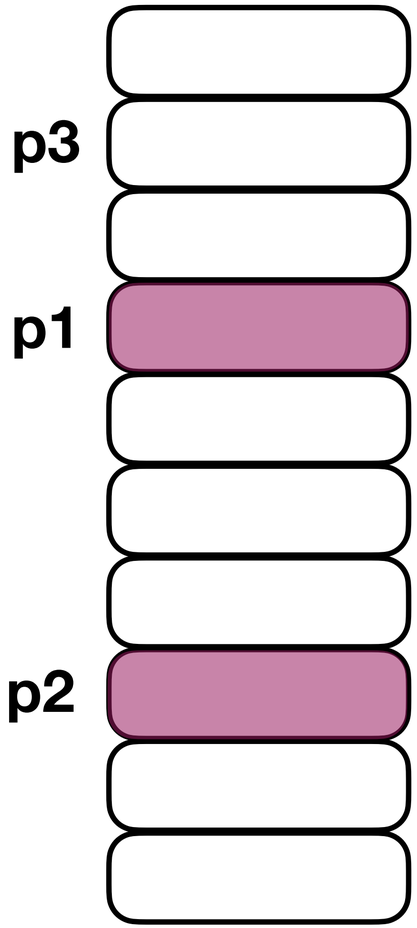


Global ownership ghost state

```
/*@ requires take L1 = IntList(p1); @*/
```

```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

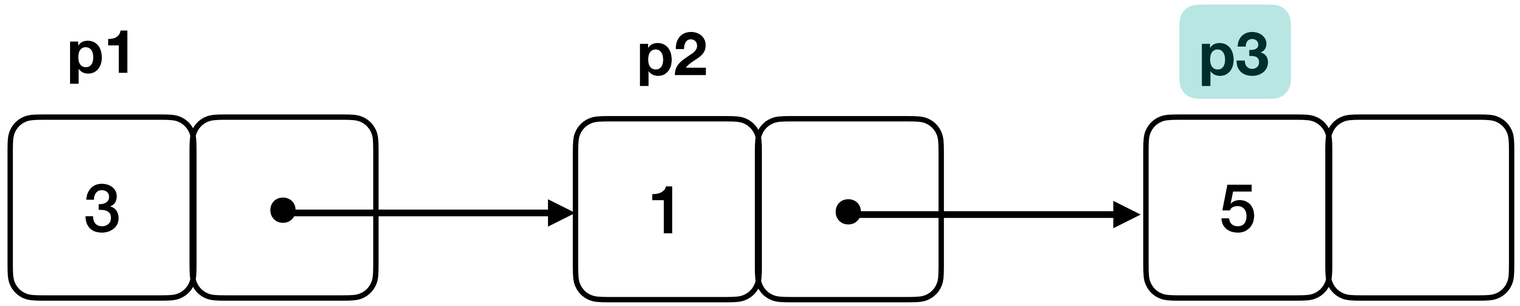


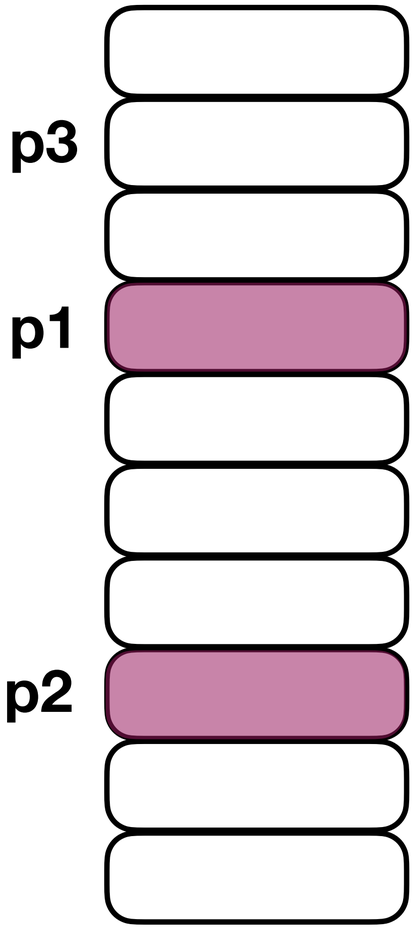


Global ownership ghost state

```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

```
/*@ requires take L1 = IntList(p1); @*/
```

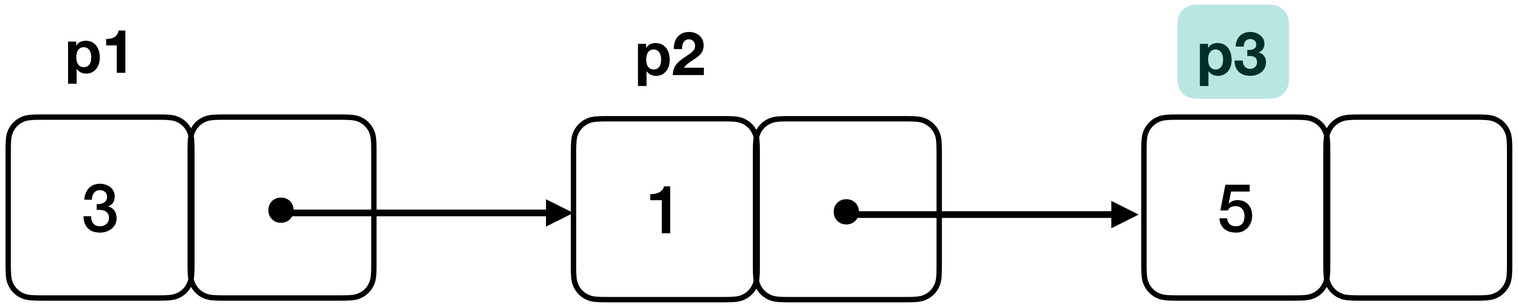


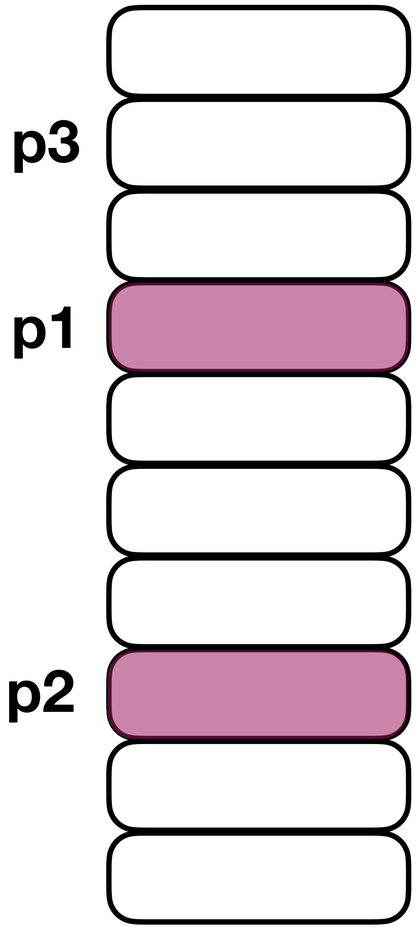


Global ownership ghost state

```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

```
/*@ requires take L1 = IntList(p1); @*/
```

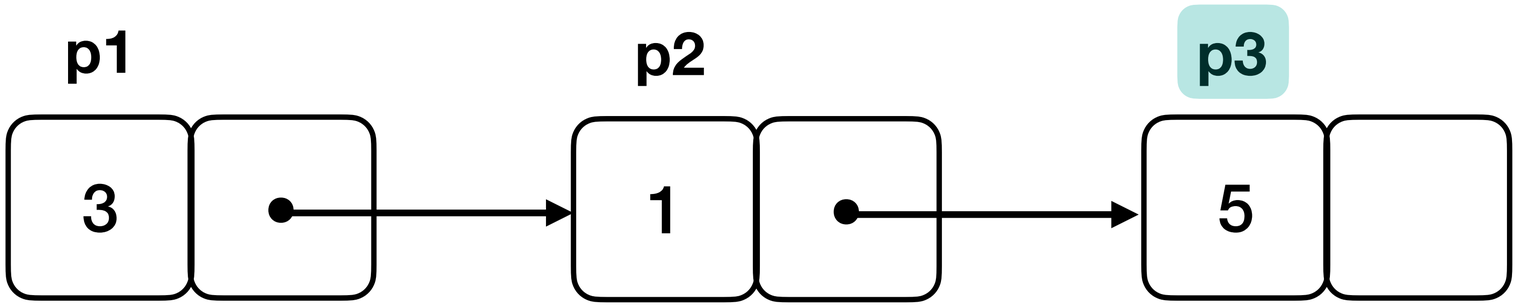


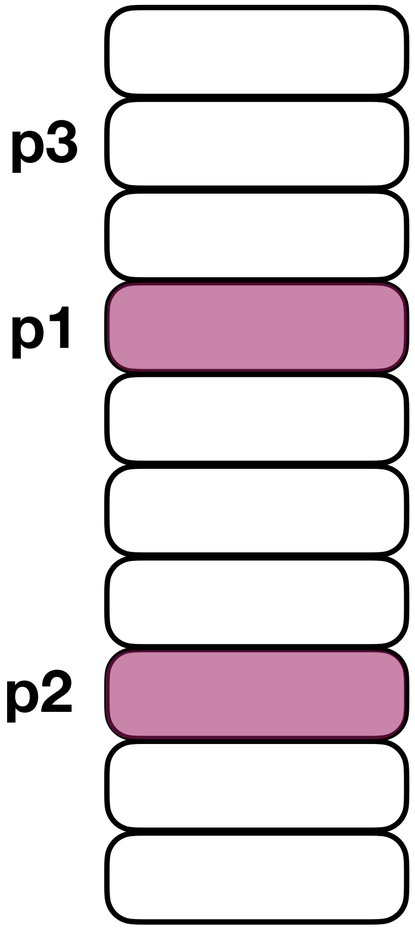


Global ownership ghost state

```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

```
/*@ requires take L1 = IntList(p1); @*/
```

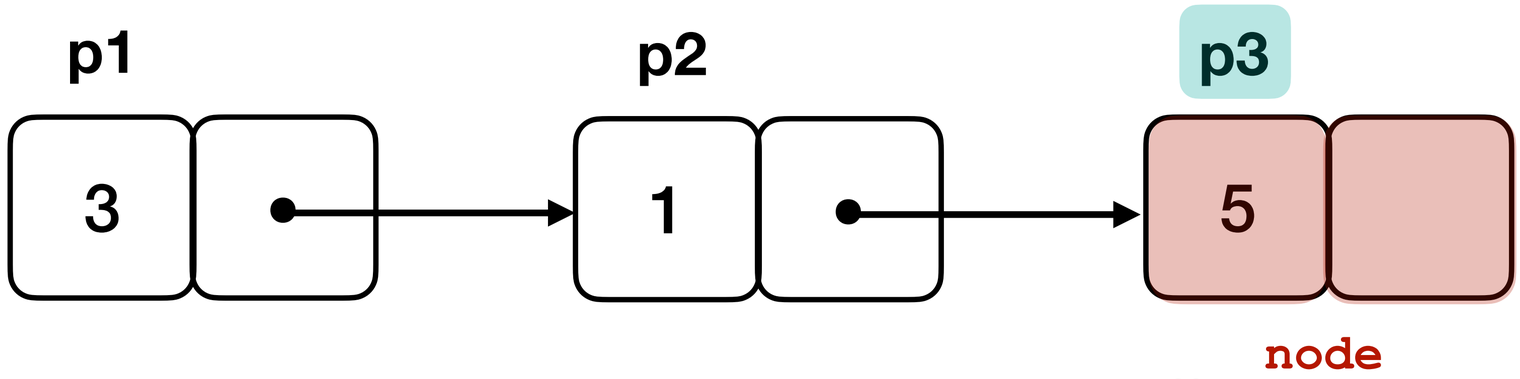


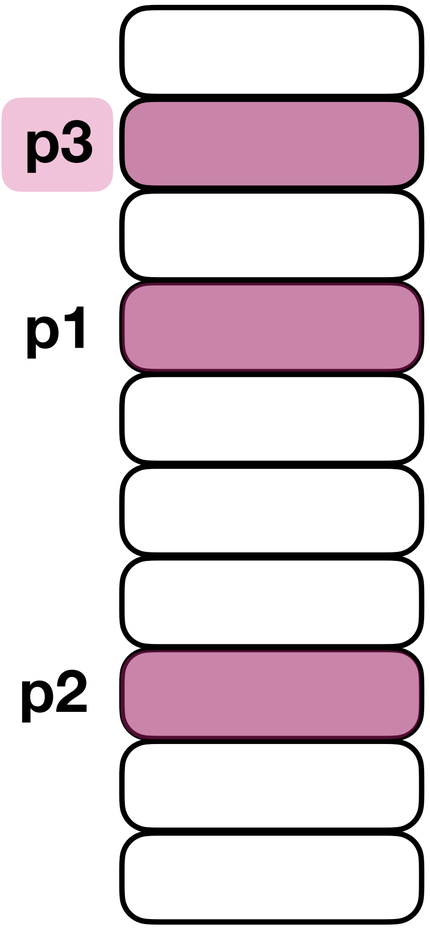


Global ownership ghost state

```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

```
/*@ requires take L1 = IntList(p1); @*/
```

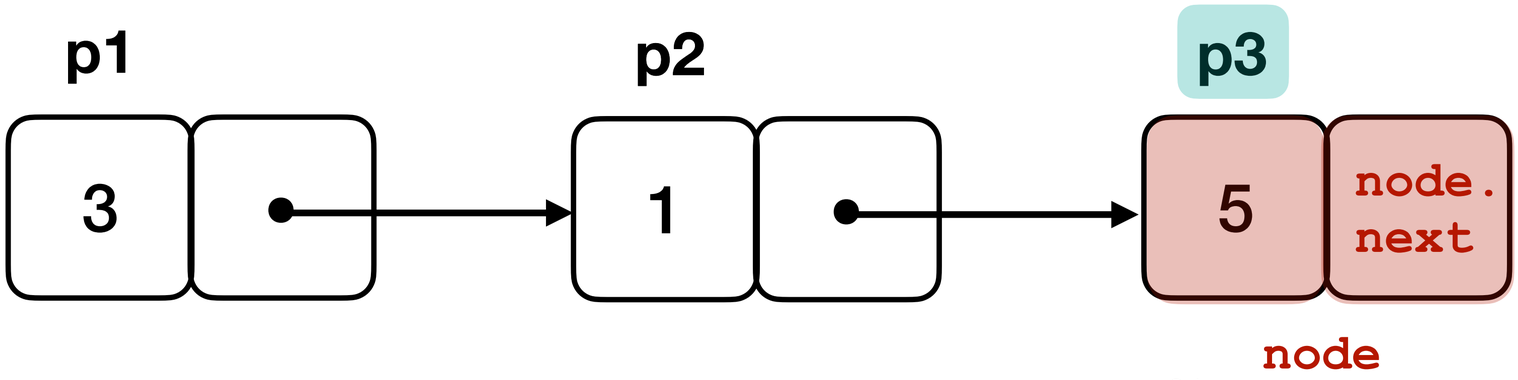


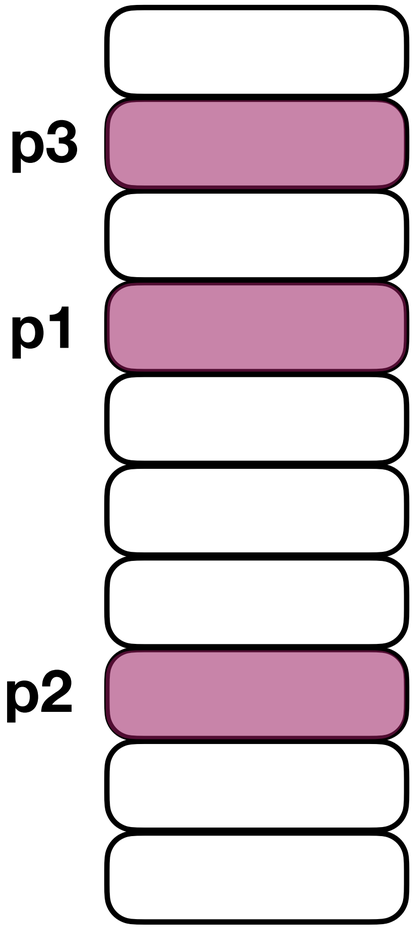


Global ownership ghost state

```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

```
/*@ requires take L1 = IntList(p1); @*/
```

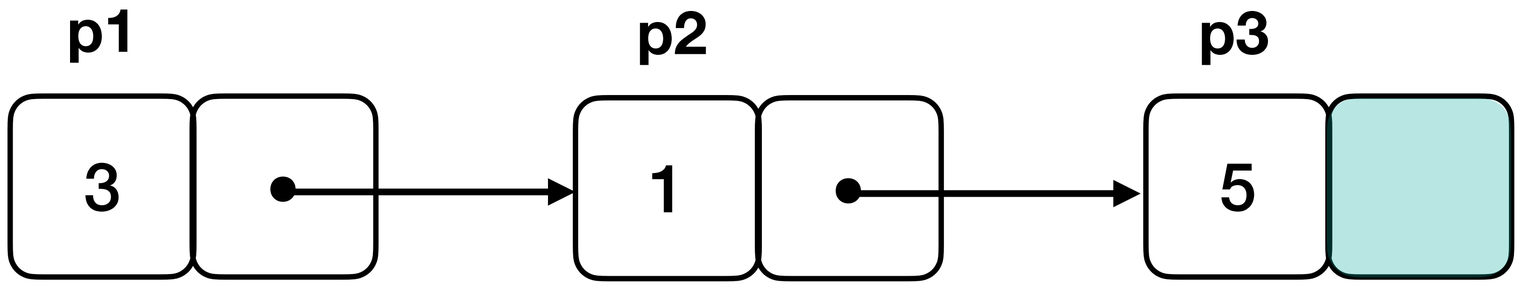


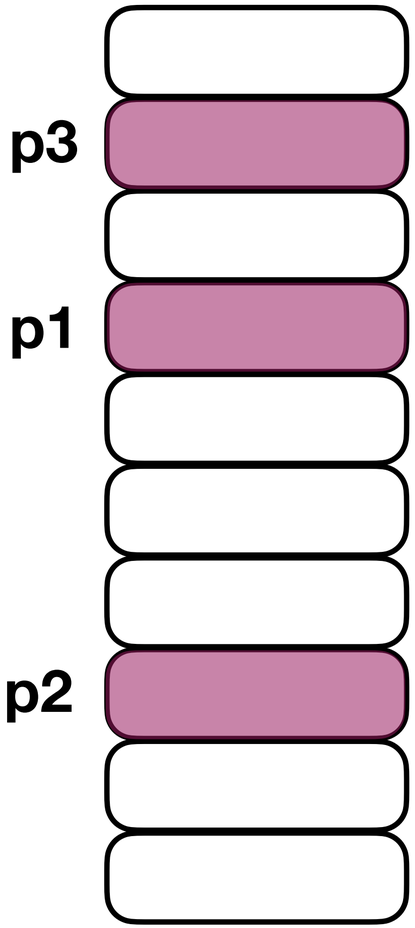


Global ownership ghost state

```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

```
/*@ requires take L1 = IntList(p1); @*/
```

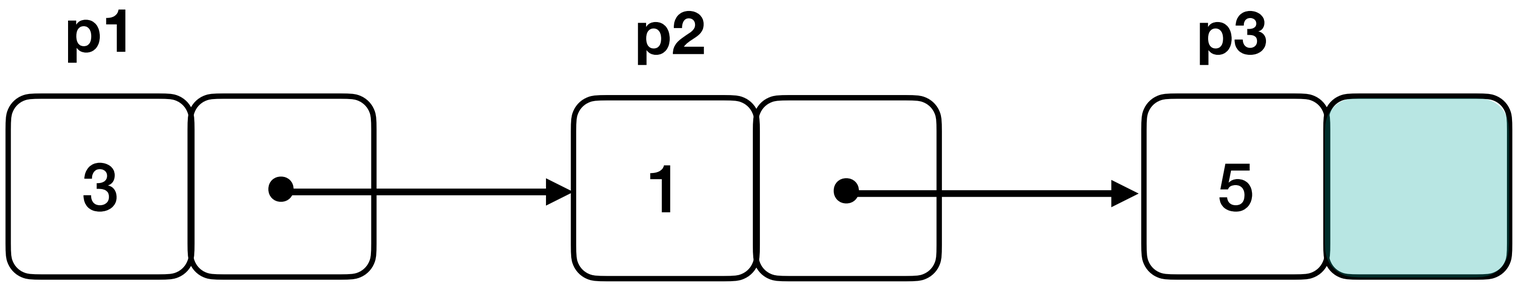


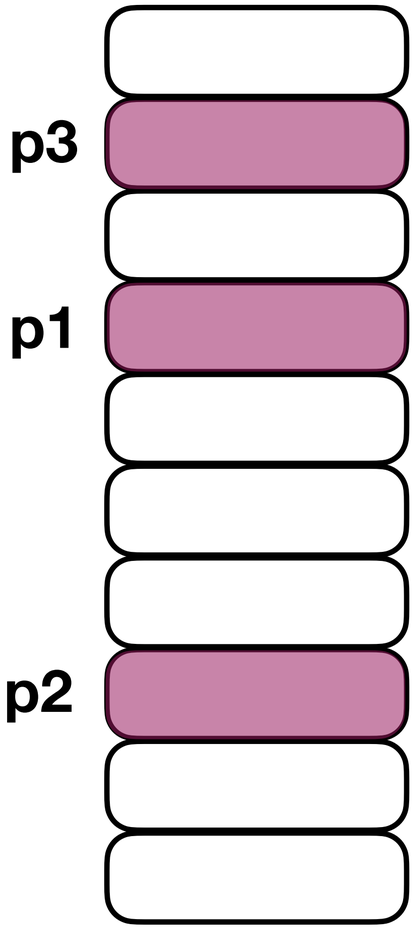


Global ownership ghost state

```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

```
/*@ requires take L1 = IntList(p1); @*/
```

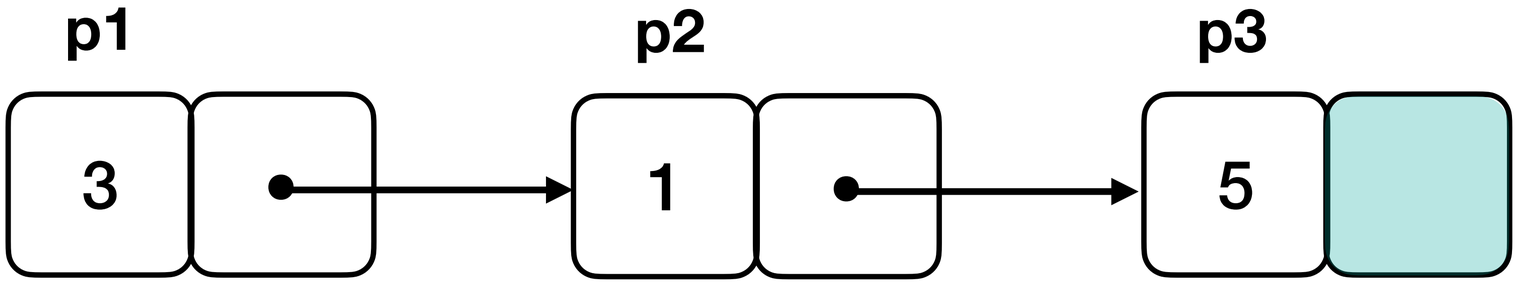


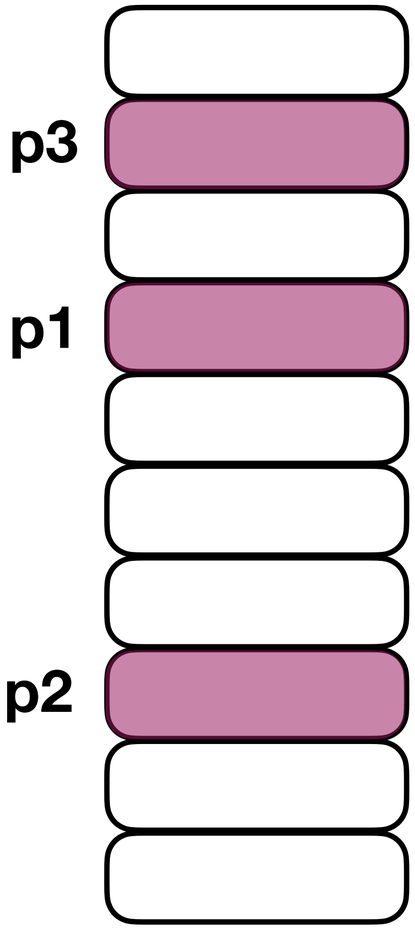


Global ownership ghost state

```
/*@ requires take L1 = IntList(p1); @*/
```

```
struct node
   {int x; struct node *next;};

predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```





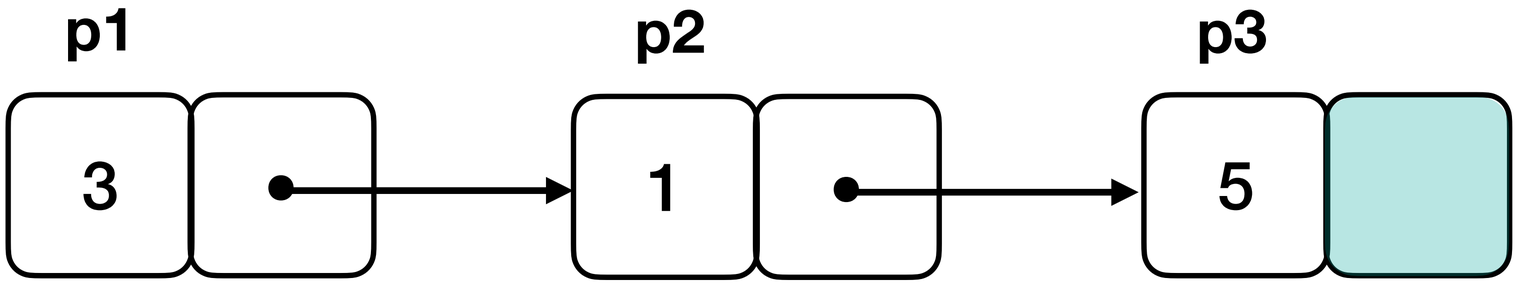
Global ownership ghost state

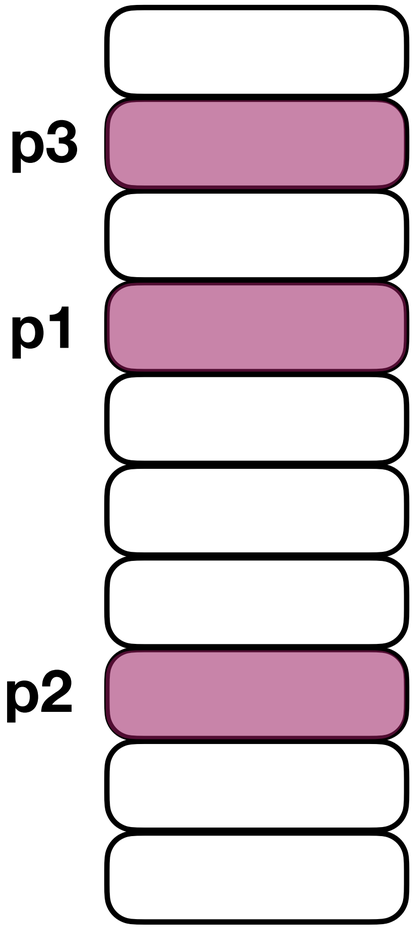
```
struct node
  {int x; struct node *next;};
```

```
/*@ requires take L1 = IntList(p1); @*/
```

```
predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

Cons {hd: **3**, tl: Cons {hd: **1**, tl: Cons {hd: **5**, tl: Nil {} } } }





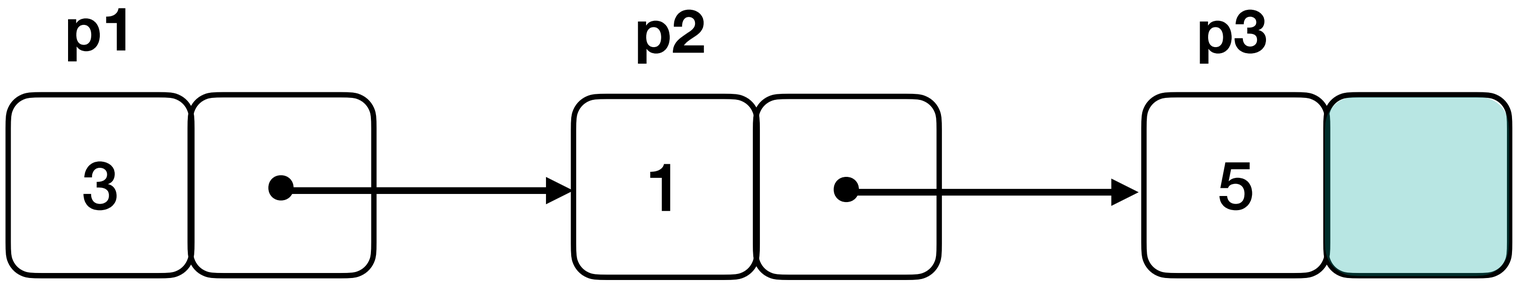
Global ownership ghost state

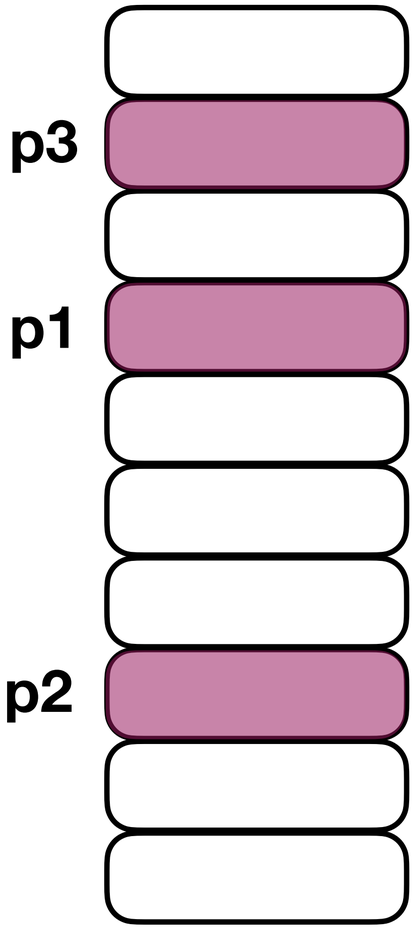
```
struct node
  {int x; struct node *next;};
```

```
/*@ ensures take L2 = IntList(p1); @*/
```

```
predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

Cons {hd: **3**, tl: Cons {hd: **1**, tl: Cons {hd: **5**, tl: Nil {} } } }





Global ownership ghost state

Partial specifications in Fulminate

- Partial intermediate specifications (e.g. loop invariants, lemma checks)
- “Fragmentary” specifications: CN-annotated and unannotated functions in the same call stack
- Wildcard ownership: manually marking a region of memory as owned throughout the program, at any stack depth

Bennet and Darcy

- PBT tools that generate random CN-precondition-satisfying inputs to exercise Fulminate checks
- Bennet uses backtracking to build test cases
- Darcy uses an SMT-solver, collecting constraints and asking Z3 for a satisfying model
  - Supports wider range of preconditions than Bennet

Bennet: Randomized Specification Testing for Heap-Manipulating Programs

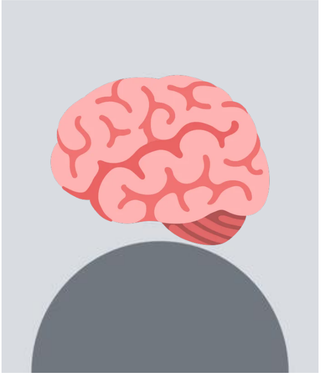
ZAIN K AAMER, University of Pennsylvania, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA

Property-based testing (PBT), widely used in functional languages and interactive theorem provers, w
by randomly generating many inputs to a system under test. While PBT has also seen some use in low-l
languages like C, users in this setting must craft all their own generators by hand, rather than letting
tool synthesize most generators automatically from types or logical specifications. For low-level code
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 separa
logic that is specially tuned to present only "easy" logical problems to an underlying constraint solver. I

7 The Darcy property-based testing tool

To apply property-based testing to the hyp allocator, we need to be able to generate inputs that satisfy the CN preconditions of its constituent functions. The only existing PBT tool for CN, Bennet [1], uses random backtracking search, which performs well when constraints are easy to satisfy and mostly local, but wastes time on values that satisfy local constraints but not constraints encountered later in the search. It cannot handle any of the preconditions involving the allocator's chunk list, which excludes all the interesting functions in this experiment. So we built a new tool, Darcy, that uses an SMT solver to generate random test inputs satisfying CN preconditions, in the form of synthesised standalone C programs that require only Z3 [21] and can be executed by Fulminate. Darcy supports nearly all of the functions in the hyp allocator. It reuses the user interface and internal generator DSL of Bennet.



Zain Aamer

CN proof tool


Christopher Pulte


Dhruv Makwana

Thomas Sewell


Kayvan Memarian

- Separation-logic refinement type system with SMT solver backend
- Aims to provide *predictable* verification
- Built on top of Cerberus, well-validated semantics for large fragment of ISO C
  - Fulminate also reuses Cerberus machinery for translating CN specs to C

CN: Verifying Systems C Code with Separation-Logic Refinement Types

CHRISTOPHER PULTE, University of Cambridge, UK
DHRUV C. MAKWANA, University of Cambridge, UK
THOMAS SEWELL, University of Cambridge, UK
KAYVAN MEMARIAN, University of Cambridge, UK
PETER SEWELL, University of Cambridge, UK
NEEL KRISHNASWAMI, University of Cambridge, UK

Despite significant progress in the verification of hypervisors, operating systems, and compilers, and in verification tooling, there exists a wide gap between the approaches used in verification projects and conventional development of systems software. We see two main challenges in bringing these closer together: verification

Into the Depths of C: Elaborating the De Facto Standards

Kayvan Memarian[1] Justus Matthiesen[1] James Lingard[2] Kyndylan Nienhuis[1]
David Chisnall[1] Robert N.M. Watson[1] Peter Sewell[1]
[1]University of Cambridge, UK [2]University of Cambridge, UK (when this work was done)
[1]first.last@cl.cam.ac.uk [2]james@lingard.com

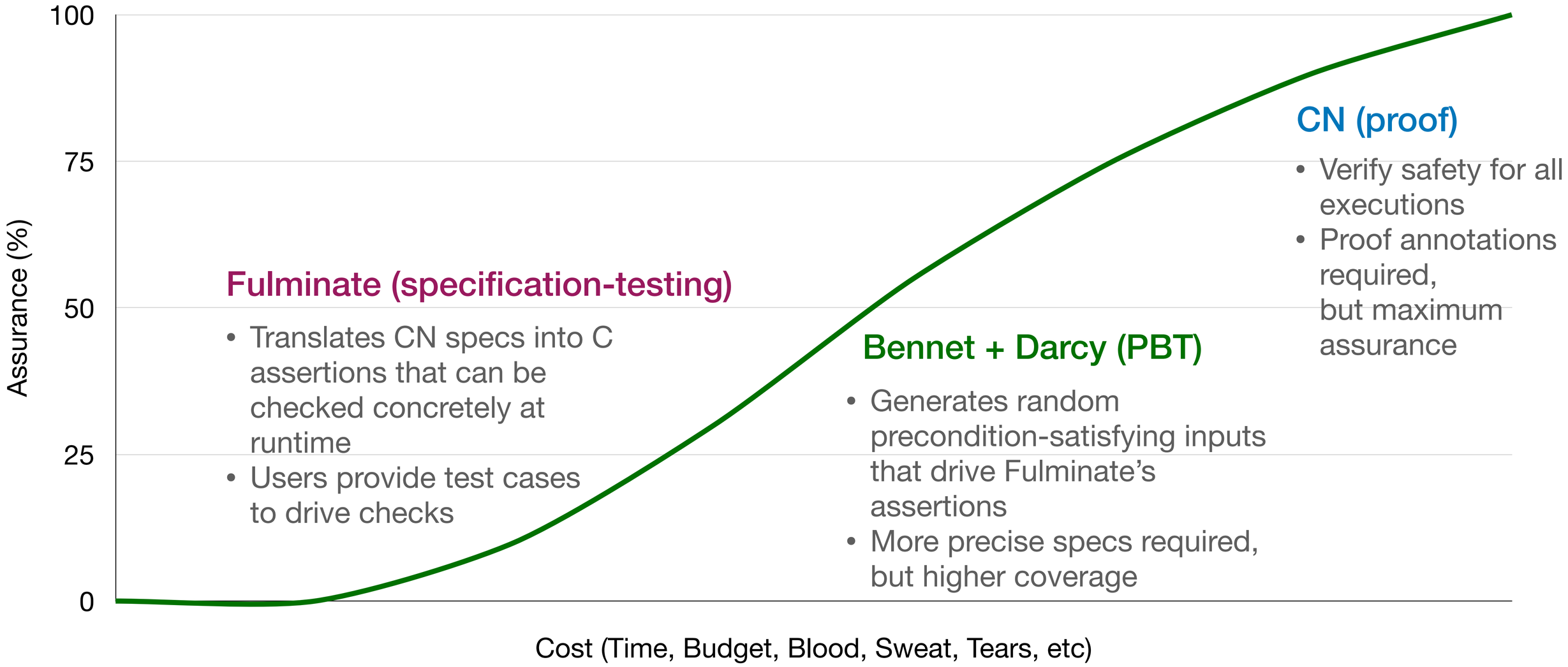
Abstract

C remains central to our computing infrastructure. It is notionally defined by ISO standards, but in reality the properties of C assumed by systems code and those implemented by compilers have diverged, both from the ISO standards and

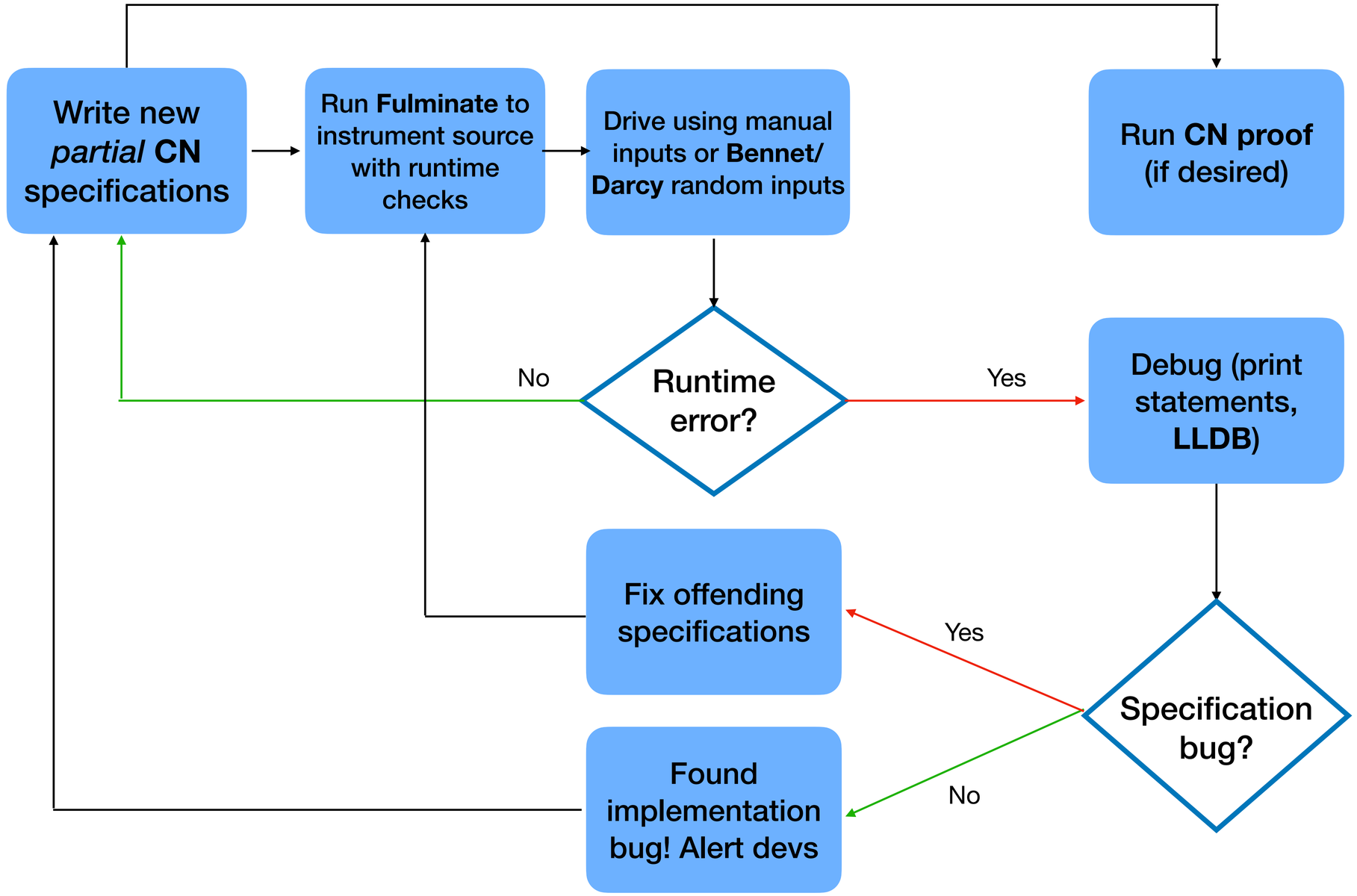
1. Introduction

C, originally developed 40+ years ago, remains one of the central abstractions of our computing infrastructure, widely used for systems programming. It is notionally defined by ANSI/ISO standards, C89/C90, C99 and C11, and a number

Gradual assurance via testing



Code-Specify-Test-Debug-Prove, using Fulminate, Bennet, Darcy and CN



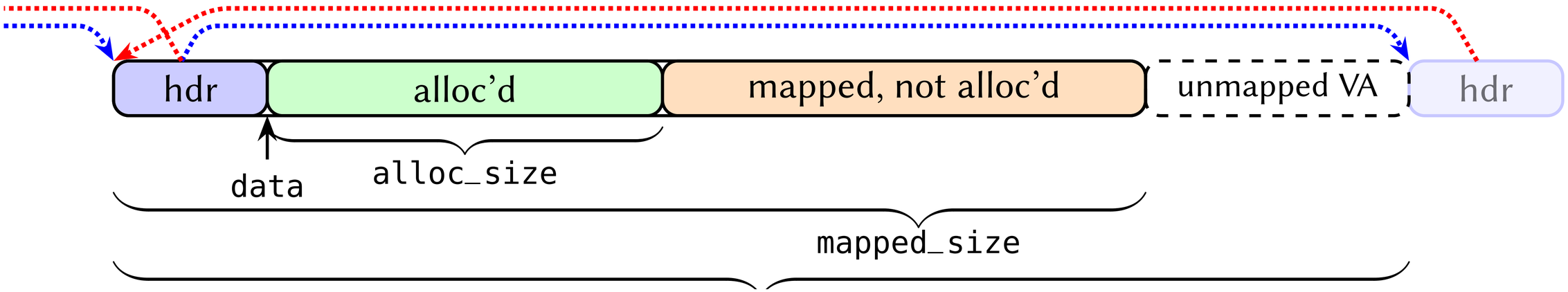
Applying *Code-Specify-Test-Debug-Prove* to real systems code using the CN toolchain

Case study: pKVM hyp allocator

- pKVM: real world hypervisor for Android developed by Google
  - Enforces memory isolation between the Android Linux kernel and guest VMs handling sensitive data
  - Runs at ARM Exception Level 2 (EL2)
  - Mostly written in C
- *hyp* allocator: heap memory allocator for pKVM like malloc/free in libc

Details: pKVM hyp allocator

- Manages memory using a doubly-linked list of **memory chunks**



Candidate spec for `chunk_install` in CN

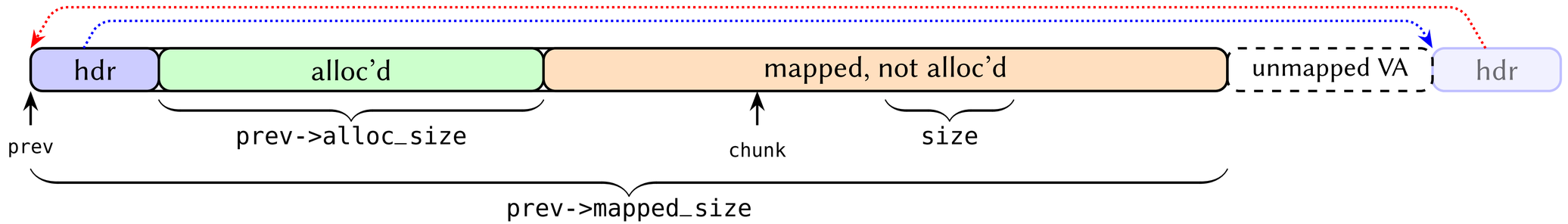
```
static int chunk_install(struct chunk_hdr *chunk, size_t size,
     struct chunk_hdr *prev, struct hyp_allocator *allocator)
/*@
requires
  take ha = OwnAllocator(allocator);
  take P = OwnChunk(prev, ha);

  (u64)chunk + hdr_size() + size
       <= (u64)prev + (u64)P.mapped_size;
ensures
  ... omitted...
@*/
```

Precondition of chunk_install

Ownership predicates (~ points-to)

Semicolons = Separating conjunctions

Logical constraints



```
static int cut_out_chunk_install(struct chunk_hdr *chunk, size_t size,
                                 struct chunk_hdr *prev,
                                 struct hyp_allocator *allocator)
/*@
    requires
        take ha_raw = RW<struct hyp_allocator>(allocator);
        let ha = {
                head:  member_shift<struct hyp_allocator>(allocator,
                chunks),
                first: ha_raw.chunks.next,
                start: ha_raw.start,
                size:  ha_raw.size
        };
        take prev_chunk = Cn_chunk_hdr(prev, ha);
        // important: check if chunk and size are appropriate to be a
        new chunk
        let prev_old_mapped_size = (u64)prev + (u64)prev_chunk.Hdr.
        mapped_size;
        let chunk_alloc_end = (u64)chunk + Cn_chunk_hdr_size() + size;

        chunk_alloc_end <= prev_old_mapped_size;
```

CN spec for `cut_out_chunk_install`

(details not important)

```
➜  fulminate-demo git:(cut-out-chunk-install) ✗ make cn-instrument-only
./main.exe
************************ Failed at ************************
function __list_add, file main.pp.exec.c, line 1367
Store failed. …
  ==> 0x1588081c0[0] (0x1588081c0) not owned at expected function call stack depth 4
  ==> (owned at stack depth: 3)
make: *** [cn-instrument-only] Error 5
```

Fulminate ownership error

Addr **`0x1588081c0`** not owned

Fulminate said, addr **`0x1588081c0`** is not owned

```
(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.2
  * frame #0: 0x0000000188483730 libsystem_c.dylib`exit
    frame #1: 0x0000000100029744 main.exe`cn_failure_default(failure_mode=<unavailable>, spec_mode=<unavailable>) at utils.c:86:7 [opt]
    frame #2: 0x000000010002a250 main.exe`c_ownership_check [inlined] cn_failure(failure_mode=CN_FAILURE_CHECK_OWNERSHIP, spec_mode=C_ACCESS) at utils.c:93:3 [opt]
    frame #3: 0x000000010002a240 main.exe`c_ownership_check(access_kind="Store", generic_c_ptr=<unavailable>, size=<unavailable>, expected_stack_depth=4) at utils.c:592:9 [opt]
    frame #4: 0x00000001000225b0 main.exe`__list_add(new=0x0000000158808158, prev=0x0000000158808008, next=0x00000001588081b8) at main.pp.exec.c:1367:2
    frame #5: 0x000000010002101c main.exe`list_add(new=0x0000000158808158, head=0x0000000158808008) at main.pp.exec.c:1429:3
    frame #6: 0x0000000100022b04 main.exe`chunk_list_insert(chunk=0x0000000158808150, prev=0x0000000158808000, allocator=0x0000000100040068) at main.pp.exec.c:4111:3
    frame #7: 0x00000001000219a8 main.exe`cut_out_chunk_install(chunk=0x0000000158808150, size=0, prev=0x0000000158808000, allocator=0x0000000100040068) at main.pp.exec.c:5009:3
```

LLDB backtrace

`next` pointer: **`0x1588081b8`** (8 bytes less)

$\Longrightarrow$ missing ownership specification for **`next`**

Fix up the specification to add ownership assertion for **`next`**. Now Fulminate is happy.

(this requires a non-trivial rewrite of the user-defined predicates)

What about Darcy?

```
cut_out_chunk_install(chunk, size, prev, allocator);

**************************************************************

Testing Summary:
cases: 1, passed: 0, failed: 1, errored: 0, skipped: 0
```

Run LLDB again:

```
(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.5
  * frame #0: 0x000000010007d788 tests.out`c_ownership_check [inlined] cn_failure(failure_mode=CN_FAILURE_CHECK_OW
NERSHIP, spec_mode=C_ACCESS) at utils.c:93:3 [opt]
    frame #1: 0x000000010007d788 tests.out`c_ownership_check(access_kind="Store", generic_c_ptr=<unavailable>, siz
e=<unavailable>, expected_stack_depth=1) at utils.c:592:9 [opt]
    frame #2: 0x000000010003b134 tests.out`cut_out_chunk_install(chunk=0x0000000000000000, size=0, prev=0x00000001
474fdf28, allocator=0x00000001464c9f88) at alloc.exec.c:5594:9
```

Suspicious null pointer being generated and passed to our function.

We forgot to specify the lower bound for it!

```
          chunk_alloc_end <= prev_old_mapped_size; // (iv)
+         prev_alloc_end <= (u64) chunk;
```

Now Darcy is happy too.

For CN proof, we need to add some more proof annotations and write and apply a lemma.

(Fulminate can check the lemma at runtime)

```
 {
         size_t prev_mapped_size;
+#ifdef __CN_VERIFY
+        LemmaCreateNewChunk(chunk, size, prev, allocator);
+#endif
         prev_mapped_size = prev->mapped_size;
         prev->mapped_size = (unsigned long)chunk - (unsigned long)prev;

         chunk->mapped_size = prev_mapped_size - prev->mapped_size;
         chunk->alloc_size = size;

+        /*@
+        split_case(ptr_eq(
+                member_shift<struct chunk_hdr>(prev, node)->next,
+                member_shift<struct hyp_allocator>(allocator, chunks)));
+        unpack Cn_chunk_hdrs(member_shift<struct chunk_hdr>(prev, node)->next,
+                member_shift<struct chunk_hdr>(prev, node), focus_pre.ha.last, Cn_hyp_allocator_core(focus_pre.ha
));
+        @*/
+
         chunk_list_insert(chunk, prev, allocator);
```

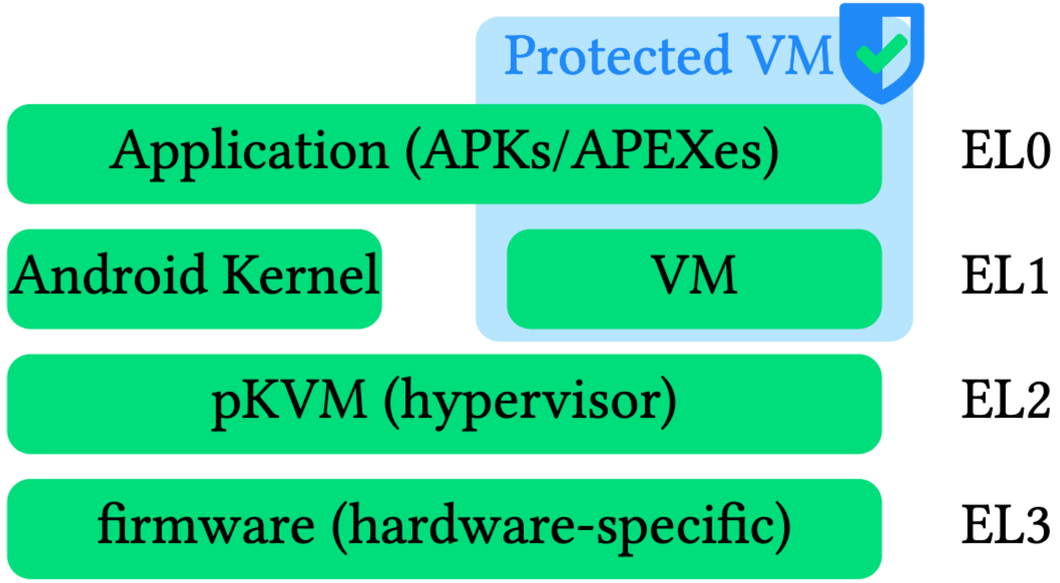
Finally, CN proof is happy with this.

Code-Specify-Test-Debug-Prove for the hyp allocator

- Applied this testing-specs-first workflow to a standalone version of the hyp allocator
  - Found **20** *specification* bugs and **4** *implementation* bugs
- Verified two main APIs: **`hyp_alloc`** and **`hyp_free`**
  - **913** lines of C code
  - **1846** lines of specifications and **735** lines of proof annotations

pKVM runs at EL2!

- Bare-metal environment
  - No C standard library (e.g. malloc/free), no third-party libraries, limited stack size and I/O functionality
- Preprocessed hyp allocator in Linux build environment has **150x** expansion, > **130,000 LoC**
  - Compared to 1.28x expansion for standalone allocator



We ran Fulminate on this, and ran the Fulminate-instrumented hyp allocator *in situ* at EL2 :-)

Figure taken from Ghost in the Android Shell: Pragmatic Test-oracle Specification of a Production Hypervisor. Kayvan Memarian, Ben Simner, David Kaloper-Meršinjak, Thibaut Pérami, Peter Sewell. SOSP 2025.

Conclusion

Our PLDI ’26 paper

- Testing specifications concretely at runtime is ***really useful***
- Writing good specifications is somewhat of an art form
  - Testing partially-written specifications helps to significantly refine that process
  - Crucially, supporting testing and proof in a **single specification language** makes this workflow feasible
- The stepping-stone from toy examples to real-world case studies is **enormous** and requires a huge (and underappreciated) amount of engineering

Questions?