Memory safety

Program analysis, type systems, and separation logic

CMPUT 664 · Formal Verification · November 23, 2026

What is memory safety?

On **every permitted execution**, memory operations must use live storage within the permitted region, with the required initialization and access rights.

Failure	What goes wrong
Out-of-bounds access	The address lies outside the allocated object
Use after free	The object’s lifetime has ended
Double free	Storage is released after it has already been released
Uninitialized read	The accessed storage does not yet contain a valid initialized value

Memory safety depends on the language

A language defines objects, their lifetimes, and permitted accesses. A memory-safety claim is **relative to those rules**.

For an existing array or slice, consider a read with a runtime index outside its bounds. A defined exception or panic can prevent the invalid memory access:

Operation	**Language-level outcome**
C array element read `a[i]`	Undefined behavior
Java array read `a[i]`	`ArrayIndexOutOfBoundsException`
Safe Rust array/slice indexing `a[i]`	Panic

Aliasing connects distant operations

In a language with explicit deallocation, `free(p)` can invalidate a later read through `q` when both pointers refer to the same cell.

Such pointers **alias**. Here `[q]` reads the cell at the address held in `q`.

```
1  p := alloc(10);       // allocate one initialized cell
2  q := p;               // copy its address
3  free(p);              // release the cell shared by p and q
4  x := [q];             // use after free
```

After	Pointer variables	Allocated cells
Allocation	p = a	a contains 10
Address copy	p = a, q = a	a contains 10
Free	p = a, q = a	none

Establishing memory safety

Approach	Where the guarantee comes from
Runtime enforcement	Bounds checks prevent invalid accesses; garbage collection manages reclamation
Static verification	Establish before execution that every modeled access is valid

Traditional program analyses

One access may require several kinds of facts:

Analysis	**Example fact**	**Safety question it helps answer**
Numerical / range	$0 \leq i < n$	Is `a[i]` within the array?
Nullness	$p \neq$ null	Can this access dereference null?
Pointer / points-to	`p` may point to either of two objects	Which objects might this access affect?
Allocation lifetime	An object is allocated or released	Is the target still live?

Propagating facts through control flow

A dataflow analysis assigns an **abstract state**, summarizing possible concrete states, to each program point.

For soundness, the summary must cover **all reachable states**. It proves an access safe when all represented states permit that access; extra states can cause false alarms.

1. Update the facts after each statement.
2. At a branch join, include possibilities from both paths.
3. At a loop, establish facts preserved by every iteration.

Example: array index bounds

Track an interval containing every possible value of `i`. Array `a` is allocated with length 3.

```
1 if flag then i := 0;
2 else         i := 2;
3 while i < 3 do
4     a[i] := 0;
5     i := i + 1;
```

Program point	**Interval for `i`**
After the branches join	$[0, 2]$, covering both assignments
Loop head, before the guard	$[0, 2] \rightarrow [0, 3] \rightarrow [0, 3]$ across analysis rounds
At `a[i]`, with `i < 3`	$[0, 2]$: the index is within bounds
After `i := i + 1`	$[1, 3]$, propagated back to the loop head

Pointer analysis

A **may-points-to analysis** computes a set $\mathrm{Pt}(p)$ of abstract objects that p may reference.

An **allocation site** is a place in the source code that allocates storage. We label sites **A**, **B**, …

One site may execute repeatedly, so its label can represent **many concrete allocations**.

Result	What it tells us
$\mathrm{Pt}(p) \cap \mathrm{Pt}(q) = \varnothing$	The pointers cannot target the same modeled object
$\mathrm{Pt}(p) \cap \mathrm{Pt}(q) \neq \varnothing$	Aliasing remains possible in the abstraction; this is not evidence of an actual alias

Example: possible pointer targets

Let `flag` be an arbitrary Boolean input.

```
1 a := alloc(0);       // allocation site A
2 b := alloc(0);       // allocation site B
3 q := a;
4 if flag then p := a;
5 else         p := b;
```

Fact at the branch join	Result	Interpretation
$\mathrm{Pt}(p)$	$\{\mathrm{A}, \mathrm{B}\}$	Include targets from both branches
$\mathrm{Pt}(q)$	$\{\mathrm{A}\}$	`q` copied `a`
$\mathrm{Pt}(b)$	$\{\mathrm{B}\}$	`b` received the allocation at B
$\mathrm{Pt}(p) \cap \mathrm{Pt}(q)$	$\{\mathrm{A}\}$	May alias: `flag = true` is a witness
$\mathrm{Pt}(q) \cap \mathrm{Pt}(b)$	$\varnothing$	Cannot alias

Inclusion constraints for pointer analysis

A **flow-insensitive** analysis computes one target set per variable for the whole program, rather than a different set at each program point.

Consider `q := p`. It copies an address. If `p` might point to A, then `q` must also be allowed to point to A.

We express this requirement as an **inclusion constraint**:

$$\mathrm{Pt}(p) \subseteq \mathrm{Pt}(q).$$

Read it as **"include every target of `p` among the targets of `q`."** Another assignment may give `q` additional targets, so this does not require equality.

Statement	Required information
`p := alloc(0)` at site A	Add A to $\mathrm{Pt}(p)$
`q := p`	Add every member of $\mathrm{Pt}(p)$ to $\mathrm{Pt}(q)$

Example: solving the inclusion constraints

Each statement adds a requirement to the same target sets:

```
1  p := alloc(0);       // site A: add A to Pt(p)
2  q := p;              // include Pt(p) in Pt(q)
3  q := alloc(0);       // site B: add B to Pt(q)
```

Start with empty sets. Add the allocation labels, then propagate targets through the copy constraint. Repeat until no set grows: this is the **least fixed point**.

Solver stage	$\mathrm{Pt}(p)$	$\mathrm{Pt}(q)$
Allocation facts from both sites	$\{\mathrm{A}\}$	$\{\mathrm{B}\}$
Include all of $\mathrm{Pt}(p)$ in $\mathrm{Pt}(q)$	$\{\mathrm{A}\}$	$\{\mathrm{A}, \mathrm{B}\}$
Check again: no additions needed	$\{\mathrm{A}\}$	$\{\mathrm{A}, \mathrm{B}\}$

Flow sensitivity preserves statement order

Now change the last assignment to overwrite p instead of q:

```
1 p := alloc(0);       // site A
2 q := p;
3 p := alloc(0);       // site B
4 // inspect p and q here
```

Analysis	Targets of p	Targets of q	Reason
Flow-insensitive	{A, B}	{A, B}	$\mathrm{Pt}(p) \subseteq \mathrm{Pt}(q)$ propagates both labels to q
Flow-sensitive, at the final point	{B}	{A}	q copied p before the second allocation

Pointers stored inside heap cells

Cells can also hold pointers. $\mathrm{Pt}(\mathrm{box})$ describes where `box` points; $\mathrm{Pt}(\mathrm{B})$ describes the pointer targets stored **inside object B**.

```
1  p := alloc(0);          // site A
2  box := alloc(null);     // site B
3  [box] := p;             // B now contains a pointer to A
4  r := [box];             // r receives that pointer to A
```

Here $\mathrm{Pt}(\mathrm{box}) = \{\mathrm{B}\}$: `box` leads to B, whose contents lead to A.

For each object $o \in \mathrm{Pt}(\mathrm{box})$, the store and load generate these constraints:

Operation	Constraint	In this example, $o = \mathrm{B}$
`[box] := p`	$\mathrm{Pt}(p) \subseteq \mathrm{Pt}(o)$	Add A to $\mathrm{Pt}(\mathrm{B})$
`r := [box]`	$\mathrm{Pt}(o) \subseteq \mathrm{Pt}(r)$	Add A to $\mathrm{Pt}(r)$

Pointer targets and allocation lifetime

Knowing **where** a pointer may point does not establish whether that storage is still allocated. Checking a read needs both target and **lifetime** information.

```
1  p := alloc(10);       // site A
2  q := p;
3  free(p);
4  x := [q];
```

Program point	$\mathrm{Pt}(q)$	**Cell from A still live?**	**Is `[q]` safe?**
Before `free(p)`	$\{\mathrm{A}\}$	Yes	Yes
At `x := [q]`	$\{\mathrm{A}\}$	No	No: use after free

Type systems as a safety discipline

We can also put rules for access and deallocation into the **type system**. The type checker controls how those rights are transferred or shared.

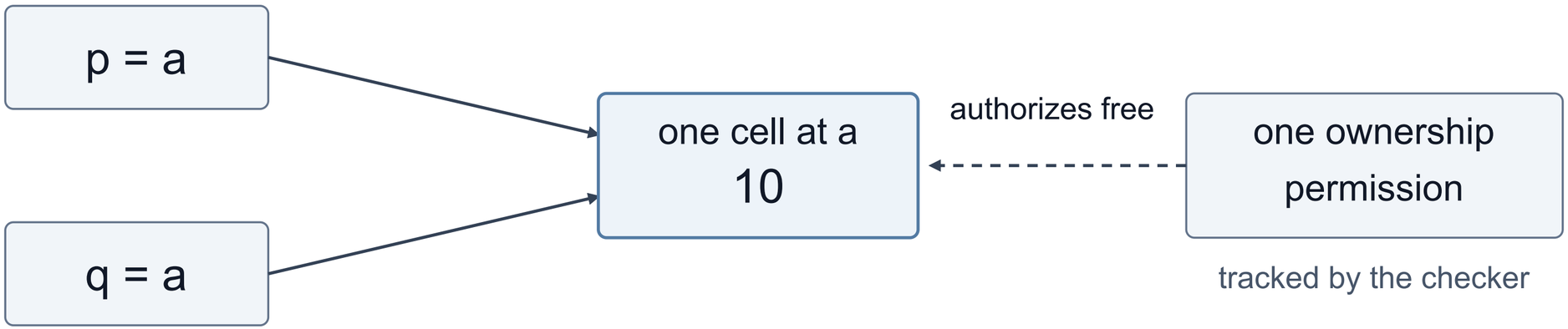
Type information	What it tracks
An ordinary pointer type such as `int*`	The kind of value expected at the target
An ownership type	A resource: the right to access and release the allocation

Copying an address must not duplicate ownership

Both `p` and `q` can contain the address of one allocation. If both names independently authorized deallocation, `free(p); free(q)` would release the same cell twice.

A pointer type such as `int*` describes the target's values, but does not track who may still release the allocation.

This motivates **linear types**, which add resource accounting to type checking. The checker tracks which name carries ownership and when that ownership is transferred or consumed:



Two pointer names reach one cell. One ownership permission authorizes releasing it.

Linear typing accounts for every ownership input

Let `Cell` be the type of an owning handle for one allocated integer cell. A binding `p : Cell` records an **available ownership resource**.

A **linear** input must be accounted for exactly once: return it, pass it onward, or consume it. The linear arrow `⊸` expresses this input-use requirement. For example, `free : Cell ⊸ Unit` consumes ownership and releases the cell; `Unit` means no useful return value.

Each body below receives one `p : Cell`, with no automatic cleanup:

Body	What happens to ownership?	Linear checking
`return p`	Passed back to the caller	Accept
`free(p); return 0`	Consumed by `free`	Accept
`return 0`	Left unaccounted for	Reject
`free(p); free(p); return 0`	Claimed again after it was consumed	Reject

Ownership can change hands

Define `transfer` to return its owning input. Returning the handle counts as its one use and passes ownership back to the caller:

```
1 transfer : Cell ⊸ Cell
2 transfer(x) = x
```

Statement	What happens to ownership?	What happens to the cell?
`p := alloc(10)`	Issued to `p`	Allocated, containing `10`
`q := transfer(p)`	Passed from `p` to `q`; `p` cannot use it again	Still the same cell
`free(q)`	Consumed; neither name owns the cell	Released

Classical and linear logic use assumptions differently

In **classical logic**, assumptions represent facts. A fact such as $n \geq 0$ can support several proof steps or remain unused.

Linear logic forbids these two freedoms for its linear resource assumptions. Linear types apply the same restrictions to available ownership inputs such as `p : Cell`:

Structural rule	Classical logic	Linear resource assumptions
Contraction: reuse an assumption	Allowed: a fact may justify several uses	Forbidden: one resource cannot supply two independent uses (**at most once**)
Weakening: leave an assumption unused	Allowed: a proof need not use every fact	Forbidden: each resource must be accounted for (**at least once**)

A linear interface for a cell

For our owning type Cell, each operation must account for its ownership input and output.

Read and write return ownership for further operations; free consumes it without returning it.

Operation	Consumes	Produces
Allocate	An integer	A cell
Read	A cell	The cell and its integer value
Write	A cell and a new integer	The updated cell
Free	A cell	Unit

Splitting the typing context

A **linear typing context** lists the resource assumptions available to a program. $\Gamma \vdash e : A$ means that e has type A under those assumptions.

To form a linear pair $A \otimes B$, split the resources between its two expressions:

$$\frac{\Gamma \vdash e_1 : A \qquad \Delta \vdash e_2 : B}{\Gamma, \Delta \vdash (e_1, e_2) : A \otimes B}.$$

Γ and Δ contain **disjoint resource bindings**.

- With resources `p : Cell` and `q : Cell`, we can form `(p, q)`.
- With only `p : Cell`, we cannot form `(p, p)`.

Moves transfer ownership

Rust's `Box<i32>` owns a heap-allocated integer.

```
let p = Box::new(10);
let q = p;                    // move ownership to q
// println!("{}", *p);        // rejected: p was moved
println!("{}", *q);           // accepted: prints 10
drop(q);                     // consume q and release the cell
```

Rust actually supports Affine Type (allow weakening)

The previous example shows that Rust rejects using an owner after a move. Rust also accepts an owner that the program never explicitly uses.

This is **affine typing**: ownership may be used at most once. It forbids contraction but allows weakening. Rust normally drops unused owners at scope exit:

```
1 fn unused_owner() {
2     let _p = Box::new(10);    // never explicitly read, moved, or dropped
3 }                             // Rust drops the box and releases its cell
```

Borrowing permits temporary access

Borrowing creates a reference to an existing value without taking ownership of it. Here `p` owns the allocation; `r : &mut i32` gives temporary exclusive read/write access to the **same integer** inside it.

During the borrow, Rust prevents access to this integer through `p` and prevents dropping the box.

```
1 let mut p = Box::new(10);
2 {
3     let r: &mut i32 = &mut *p; // reference to the same integer
4     *r += 1;                   // change it from 10 to 11
5 }                              // r is out of scope
6 assert_eq!(*p, 11);             // p sees the updated value
```

Syntax	Meaning in this example
`*p`	The integer stored inside the box
`&mut *p`	Create an exclusive reference to that integer
`*r += 1`	Update that same integer through `r`

The owner must outlive the borrow

A shared reference `&i32` gives read access to an existing integer. Its allocation must remain live until the last use of the reference.

```
let p = Box::new(10);
let r = &*p;
drop(p);                    // rejected: p is still borrowed
println!("{}", r);          // r would access the freed cell
```

What does the type discipline establish?

A **sound type system** establishes that well-typed programs avoid the modeled memory faults, assuming correct primitive operations.

This guarantee permits memory leaks.

- Unique ownership controls transfer and deallocation.
- Borrowing controls access while references remain usable.
- Array bounds still need proofs, restricted interfaces, or runtime checks.

Separation logic: states and heaps

Ordinary Hoare logic already tracks the values of variables. The heap alone does not tell us which address `p` holds, so we keep that variable information and add the allocated cells.

The two maps answer different questions:

Map	What it records	Example
Store σ	The value held in each program variable	$\sigma(p) = a$: `p` holds address a
Heap h	The value stored at each allocated address	$h(a) = 10$: the cell at a contains 10

Reading `[p]` needs both lookups: first find the address in `p`, then read the cell there. In this example, $h(\sigma(p)) = 10$. If that address is not allocated, the read faults.

We use initialized, single-cell allocations. A state is the pair (σ, h), with a finite heap defined only at allocated addresses:

$$\sigma : \mathrm{Var} \to \mathrm{Val}, \qquad h : \mathrm{Loc} \rightharpoonup_{\mathrm{fin}} \mathrm{Val}.$$

Variable assignment and heap update are different

Follow one allocation through these statements. Let a be the address returned by `alloc`.

Copying a pointer changes a variable's value. Updating or freeing its cell changes the heap. Freeing the cell does not erase the addresses held in the variables:

After the statement	Store: variable values	Heap: allocated cells
`p := alloc(10)`	$p = a$	$a \mapsto 10$
`q := p`	$p = a,\ q = a$	Still $a \mapsto 10$
`[p] := 11`	Still $p = a,\ q = a$	$a \mapsto 11$
`free(p)`	Still $p = a,\ q = a$	Empty: a is no longer allocated

Empty heap and points-to

Heap assertions describe **exact fragments**: emp describes an empty heap; $p \mapsto 10$ describes one allocated cell at `p` containing `10`.

Points-to supplies the resource needed to read, update, or free the cell. Pure assertions such as $p = q$ constrain the store, independently of the heap.

eval_σ evaluates a value expression using the store:

$$(\sigma, h) \vDash \mathsf{emp} \iff h = \varnothing,$$

$$(\sigma, h) \vDash p \mapsto v \iff h = \{\sigma(p) \mapsto \mathrm{eval}_\sigma(v)\}.$$

Separating conjunction

$P * Q$ describes a heap that can be split into **disjoint parts**:

Write $h_1 \uplus h_2$ for the union of heaps with disjoint address domains.

Both assertions use the **same store** σ. We split the heap, not the variables.

$$(\sigma, h) \vDash P * Q \iff \exists h_1, h_2 . \begin{cases} h = h_1 \uplus h_2, \\ (\sigma, h_1) \vDash P, \\ (\sigma, h_2) \vDash Q. \end{cases}$$

A concrete heap split

Suppose $\sigma(p) = a$ and $\sigma(q) = b$, with $a \neq b$.

These heap fragments form a simple **resource algebra**: composition is disjoint union, with the empty heap as identity. Overlapping fragments cannot be combined by $\uplus$.

Heap	Allocated cells	Assertion
h_1	$a \mapsto 10$	$p \mapsto 10$
h_2	$b \mapsto 20$	$q \mapsto 20$
$h_1 \uplus h_2$	$a \mapsto 10,\ b \mapsto 20$	$p \mapsto 10 * q \mapsto 20$

Heap triples include memory safety

We write $\{P\}\ C\ \{Q\}$ to require, from every state satisfying P:

1. Every memory operation performed by C is **valid**.
2. If C **finishes normally**, the resulting state satisfies Q.

For `free(p)`, the initial heap must contain an allocated cell at the address in `p`. Recall that emp means **no cells are allocated**:

Proposed triple	**Starting heap**	**What happens?**
$\{p \mapsto 10\}\ \text{free}(p)\ \{\text{emp}\}$	One allocated cell at `p`	**Valid:** the cell is released, leaving an empty heap
$\{\text{emp}\}\ \text{free}(p)\ \{\text{emp}\}$	No allocated cells	**Invalid:** there is no cell to release, so the operation faults

Ordinary conjunction and separation differ

Assertion	What it requires
$p \mapsto 0 \land q \mapsto 0$	Both describe the **same** singleton heap; here $p = q$
$p \mapsto 0 * q \mapsto 0$	Two **disjoint** singleton heaps; therefore $p \neq q$

With $p = q$, updating through p changes the cell described by $q \mapsto 0$, so this triple is **invalid**:

$$\{p \mapsto 0 \land q \mapsto 0\}\ [p] := 1\ \{p \mapsto 1 \land q \mapsto 0\}.$$

With disjoint cells, the cell at q retains its contents:

$$\{p \mapsto 0 * q \mapsto 0\}\ [p] := 1\ \{p \mapsto 1 * q \mapsto 0\}.$$

Local rules for update and free

Each rule describes the single cell required by the command.

Update preserves ownership and changes the contents. Free consumes the cell. Neither command changes the value of variable p.

$$\{p \mapsto v\}\ [p] := w\ \{p \mapsto w\}$$

$$\{p \mapsto v\}\ \mathsf{free}(p)\ \{\mathsf{emp}\}$$

Allocation creates a resource

Allocation chooses an address absent from the **whole current heap**, initializes the cell, and stores the address in p. Existing cells are unchanged.

We assume allocation succeeds and there are always fresh addresses.

$$\{\mathsf{emp}\}\ p := \mathsf{alloc}(0)\ \{p \mapsto 0\}.$$

Reusing a proof in a larger heap

The local update rule already proves what happens to the cell at `p`:

$$\{p \mapsto 10\}\ [p] := 11\ \{p \mapsto 11\}.$$

Now suppose the heap also contains a separate cell at `q`, holding `20`. The same command updates only the cell at `p`, so the cell at `q` remains allocated and still holds `20`.

We can reuse the first proof, carrying $q \mapsto 20$ unchanged through it. This additional assertion is called the **frame**:

$$\{p \mapsto 10 * q \mapsto 20\}\ [p] := 11\ \{p \mapsto 11 * q \mapsto 20\}.$$

The frame rule

Suppose we have proved $\{P\}\ C\ \{Q\}$. A larger heap may also contain a **disjoint fragment** described by R.

The frame rule lets us reuse that proof and keep R in the postcondition. It requires that C does not reassign any variable whose value is used by R:

$$\frac{\{P\}\ C\ \{Q\} \qquad \mathrm{Mod}(C) \cap \mathrm{FV}(R) = \varnothing}{\{P\ *\ R\}\ C\ \{Q\ *\ R\}}.$$

Symbol	In the preceding example
P	$p \mapsto 10$: the cell before the update
C	`[p] := 11`: the command
Q	$p \mapsto 11$: the cell after the update
R	$q \mapsto 20$: the additional cell, unchanged

Why the variable side condition matters

The frame $R = q \mapsto 20$ uses the address stored in `q`. Even if its cell stays unchanged, reassigning `q` can make this assertion false.

Start each row with $p \mapsto 10 * q \mapsto 20$:

Command	Variable assignments	Is the frame $q \mapsto 20$ preserved?
`[p] := 11`	None: `p` still holds the same address	Yes: `q` still names the untouched cell
`q := p`	`q` receives a different address	No: `q` now names the cell containing `10`

$\mathrm{Mod}(C)$ lists variables reassigned by the command. $\mathrm{FV}(R)$ lists variables whose values the assertion uses; here it is $\{q\}$.

The condition requires these sets to have no variable in common:

$$\mathrm{Mod}(C) \cap \mathrm{FV}(R) = \varnothing.$$

A proof with an unchanged frame

Use $q \mapsto 20$ as the frame: it does not mention p, and none of these commands assigns q.

The endpoints use emp $* R \equiv R$, since an empty fragment contributes no cells.

```
1 { q ↦ 20 }
2 p := alloc(0);          // fresh address, so p != q
3 { p ↦ 0 * q ↦ 20 }
4 [p] := 1;
5 { p ↦ 1 * q ↦ 20 }
6 free(p);
7 { q ↦ 20 }
```

Address equality does not duplicate ownership

Two names can describe **one owned cell**. Equality does not produce the separating resource $p \mapsto 10 * q \mapsto 10$:

```
1 { p ↦ 10 }
2 q := p;
3 { p ↦ 10 ∧ q = p }     // one cell, also described by q ↦ 10
4 free(p);
5 { emp ∧ q = p }         // address equality remains; the cell is gone
```

Which preconditions make this safe?

```
1  free(p);
2  [q] := 0;
```

For each unsafe case, identify the missing resource at the second command.

Proposed precondition	Safe from every satisfying state?
$p \mapsto 7 * q \mapsto 9$	?
$p \mapsto 7 \wedge q = p$	?
$p \mapsto 7 \wedge q \neq p$	?

Ownership connects the two approaches

Both approaches use resource-sensitive reasoning: our cell types track ownership use, while points-to assertions also record cell contents.

Linear / affine typing	**Separation-logic proof**
Move an owned cell	Transfer its points-to resource between specifications
Split resources between expressions	Split the heap with $*$
Consume ownership when freeing	$p \mapsto v$ becomes emp
Require an ownership resource before use	Require a points-to resource before an access

Beyond one cell: an owned list

Extend the heap so one allocated **node** stores a pair: its value and next pointer. null is never allocated.

Build the list inductively from the empty case. Each added head is separate from its tail, giving a finite acyclic list:

$$\begin{aligned}\mathsf{list}(p) \;\equiv\; & (p = \mathsf{null} \wedge \mathsf{emp}) \\ & \vee\; \exists v, n.\;\; p \mapsto (v, n) \ast \mathsf{list}(n).\end{aligned}$$