Alpha is for Address!

The Essence of Memory Safety

Abstract

Languages and systems that guarantee memory safety are fundamental tools for building robust software, protecting programs from numerous security vulnerabilities and bugs. But what, exactly, distinguishes such tools from "memory-unsafe" ones? Traditionally, memory safety has been described in terms of bad behaviors (buffer overflow, use-after-free, etc.) that are guaranteed not to occur, ignoring the question of what *good* things are guaranteed by memory safety—specifically, what new reasoning principles are available in a memory-safe setting.

We study the implications of memory safety for a simple imperative language, articulating a powerful local reasoning principle that extends conventional separation logic. To justify this principle, we give the language a semantics based on nominal sets, using the name restriction operation for characterizing the scope in which some pointer's concrete value is relevant, and giving a convenient means for composing parts of a heap while preserving isolation guarantees. The key idea is that outside of its scope, a pointer should be treated modulo alpha-equivalence. This semantics gives rise to a natural equational theory for heap-manipulating programs.

1. Introduction

Good programmers want their code to be memory safe. Those who work in low-level languages such as C—either because efficiency is a primary concern or because they simply want to live free—use careful reasoning and methodical coding idioms to make sure that their programs never access arrays out of bounds or free pointers twice. To be completely certain, they may go a step further and write rigorous correctness proofs, for example in separation logic [20]. This approach can work, but it has a significant drawback: reasoning about memory safety—whether informal or formal—must encompass every aspect of the program, including all plugins, libraries, etc. If a memory-safety violation occurs *anywhere* in the program, all bets are off *everywhere*. Formally, this all-or-nothing character can be seen in the separation-logic rule for sequential composition, which requires a Hoare triple to be proved about each of the two sub-commands; supplying these proofs involves, in particular, proving that both sub-commands are memory safe, which may demand arbitrarily detailed reasoning about their internals.

Programmers who prefer an easier life rely on a variety of automatic techniques for preventing or detecting memory-safety violations, including languages with garbage collection, runtime array-bounds checking, and a range of static and dynamic analyses. Besides the obvious benefit of reducing programmer effort, this approach has two big advantages. First, it safeguards not only the programmer's own code but also potentially buggy libraries and malicious plugins. Second, it supports robust *local reasoning* by providing a strong guarantee, when looking at a particular piece of code, about what other code elsewhere in the system *cannot* do.

To illustrate this point, suppose we are building a web browser in some hypothetical memory-safe language. The browser might include a mechanism for executing a plugin downloaded from the Internet:

$$\mathsf{plugin.run}(x, y, z)$$

Another part of the program might manipulate an object p that stores user passwords:

$$p \leftarrow \mathsf{Password.new}(\texttt{"passwords.txt"})$$

If we happen to know that p is not reachable (directly or transitively) through pointers present in the arguments x, y, and z passed to the plugin or in global variables that it can access, we can infer that the plugin can neither disclose passwords stored in p nor change their values.

This raises the question of what kind of formal treatment would justify this informal reasoning. Previous formal treatments of memory safety [4, 11, 14, etc.], have commonly been phrased in terms of specifying *bad* low-level behaviors (e.g., out-of-bounds array access) that cannot arise in a memory-safe setting. Our aim in this paper is to go a step further, articulating formally the *good* high-level reasoning principles that are available in a memory-safe setting. Our main contributions are:

- We identify key informal reasoning principles in a small memory-safe language (Sections 2 and 3) and argue that these principles capture the essence of memory safety.
- Using the theory of nominal sets [17] (reviewed in Section 4), we recast the semantics of this language in a more abstract form (Section 5), where program states can be viewed as a union of structured components, each consisting of public objects that can be referred to from other components and local objects that cannot be named directly. The new semantics is a natural setting for formalizing the above reasoning principles, yielding a rich equational theory of program states.
- To illustrate the expressive power of the semantics and the usefulness of the reasoning principles it supports, we compare them to conventional separation logic (Section 6), discussing how it extends the logic's local reasoning features and showing how it can support simple program verification. (We leave formulating a full-blown extended separation logic for future work.)
- We discuss how our results are weakened in systems that allow limited violations of memory safety (Section 7) and the implications of such violations for program reasoning.

Section 8 discusses previous work on memory safety, while Section 9 concludes and points to future work. Our main results, including all of Section 5 and some of Sections 6 and 7, have been formalized with the Coq proof assistant [23].

2. Preliminaries

The setting in which we work is a simple imperative language with dynamic allocation and manual memory management. The syntax, completely standard, is presented in Figure 1.

$$
\begin{aligned}
\mathbb{B} &::= \mathsf{true} \mid \mathsf{false} \\
\langle \mathrm{op} \rangle &::= + \mid - \mid \times \mid \mathsf{and} \mid \mathsf{or} \mid = \mid \cdots \\
\langle \mathrm{const} \rangle &::= n \in \mathbb{Z} \mid b \in \mathbb{B} \mid \mathsf{nil} \\
\langle \mathrm{exp} \rangle &::= \langle \mathrm{var} \rangle \mid \langle \mathrm{const} \rangle \mid \langle \mathrm{exp} \rangle \ \langle \mathrm{op} \rangle \ \langle \mathrm{exp} \rangle \\
\langle \mathrm{comm} \rangle &::= \langle \mathrm{var} \rangle \leftarrow \langle \mathrm{exp} \rangle \mid \langle \mathrm{var} \rangle \leftarrow [\langle \mathrm{exp} \rangle] \mid [\langle \mathrm{exp} \rangle] \leftarrow \langle \mathrm{exp} \rangle \\
&\quad \mid \langle \mathrm{var} \rangle \leftarrow \mathsf{alloc}(\langle \mathrm{exp} \rangle) \mid \mathsf{free}(\langle \mathrm{exp} \rangle) \mid \mathsf{skip} \\
&\quad \mid \mathsf{if} \ \langle \mathrm{exp} \rangle \ \mathsf{then} \ \langle \mathrm{comm} \rangle \ \mathsf{else} \ \langle \mathrm{comm} \rangle \\
&\quad \mid \mathsf{while} \ \langle \mathrm{exp} \rangle \ \mathsf{do} \ \langle \mathrm{comm} \rangle \ \mathsf{end} \mid \langle \mathrm{comm} \rangle ; \langle \mathrm{comm} \rangle
\end{aligned}
$$

Figure 1. Syntax

This language has essentially the same features as the ones often used to discuss separation logic [20]; in particular, it makes a clear distinction between commands that manipulate the memory and ones that don't. The command $x \leftarrow e$ is used to assign the value of expression e to local variable x; the command $x \leftarrow [e]$ loads a value from the memory location denoted by e; and the command $[e] \leftarrow e'$ stores the value of e' *to* the location denoted by e. (We explain the semantics of some of these commands in detail below.) The important difference when compared to more standard settings is that we adopt a *structured* memory model: The memory is partitioned into independent *blocks*, which can be thought of as arrays of atomic values. Blocks are accessed through *pointers*, and each pointer can reference at most one block. Pointer arithmetic is allowed, for accessing different elements of a block; however, as explained below, it is not possible to make a pointer reference a different block by manipulating it.

The *values* $\mathcal{V}$ of the language are:

$$\mathcal{V} ::= n \in \mathbb{Z} \mid b \in \mathbb{B} \mid (a, n) \in \mathbb{A} \times \mathbb{Z} \mid \mathsf{nil}$$

Pointers are pairs $(a, n) \in \mathbb{A} \times \mathbb{Z}$, where $\mathbb{A}$ is some countably infinite set of atoms, which will be used as *block identifiers*. These identifiers distinguish between pointers that point to distinct memory blocks. The integer component of a pointer represents an *offset* into the block corresponding to that pointer. The value nil is an atomic value that can be used to represent an invalid pointer.

As usual [20], programs of this language operate on a state that is partitioned into two components: a *local store* $l \in \mathcal{L} = \langle \mathrm{var} \rangle \rightharpoonup_{\mathrm{fin}} \mathcal{V}$ and a *heap* $m \in \mathcal{M} = (\mathbb{A} \times \mathbb{Z}) \rightharpoonup_{\mathrm{fin}} \mathcal{V}$, where $\rightharpoonup_{\mathrm{fin}}$ denotes a partial function with finite domain. We write $\mathcal{S} = \mathcal{L} \times \mathcal{M}$ for the set of program states.

An expression e denotes a function $[\![e]\!] : \mathcal{L} \to \mathcal{V}$ from local stores to values. Its definition is mostly standard, with a few small differences. Arithmetic operations are overloaded to allow manipulating the offset component of a pointer:

$$[\![e_1 + e_2]\!](l) = \begin{cases} n_1 + n_2 & \text{if } [\![e_i]\!](l) = n_i \\ (a, n_1 + n_2) & \text{if } [\![e_1]\!](l) = (a, n_1) \\ & \text{and } [\![e_2]\!](l) = n_2 \\ \mathsf{nil} & \text{if } [\![e_1]\!](l) = \mathsf{nil} \\ \cdots & \end{cases}$$

To simplify the definition, we assume that nonsensical combinations (e.g., trying to add two pointers together, or adding nil to anything) simply result in nil, as does trying to evaluate an undefined variable:

$$[\![x]\!](l) = \begin{cases} l(x) & \text{if } x \in \mathrm{dom}(l) \\ \mathsf{nil} & \text{otherwise} \end{cases}$$

For the moment, we maintain a complete separation between pointers and other values: we can offset pointers, test them for equality, and use them to interact with memory, but nothing else. Later (Section 7.1), we discuss a variation where we add an operator to cast pointers to integers.

The evaluation relation, written $(l, m, c) \Downarrow o$, is again standard: it specifies the outcome o of evaluating a program c on a state (l, m); this outcome can be either a final state (l', m') or the special value abort, indicating that a memory error occurred during execution. We sometimes write this relation as $(s, c) \Downarrow o$, considering the entire initial state as a single entity.

Here is the rule for assigning the result of an expression to a local variable:

$$\frac{[\![e]\!](l) = v \qquad l[x \to v] = l'}{(l, m, x \leftarrow e) \Downarrow (l', m)}$$

The notation $l[x \to v]$ denotes a map that agrees with l on all variables except for x, where its value is v.

The rule for loading from memory is similar:

$$\frac{[\![e]\!](l) = (a, n) \qquad m(a, n) = v \qquad l[x \to v] = l'}{(l, m, x \leftarrow [e]) \Downarrow (l', m)}$$

Notice that we require that the memory be defined at the location denoted by e; if we try to load from a nonexistent memory location (with an out-of-bounds or dangling pointer), we get a memory error instead:

$$\frac{[\![e]\!](l) = (a, n) \qquad m(a, n) \text{ is undefined}}{(l, m, x \leftarrow [e]) \Downarrow \mathsf{abort}}$$

We elide the two similar rules for storing into memory (one for a successful store; the other for storing to an undefined location, resulting in abort).

Programs can allocate new memory using alloc:

$$\frac{[\![e]\!](l) = n \qquad n > 0 \qquad a = \mathsf{fresh}(l, m) \qquad l[x \to (a, 0)] = l'}{(l, m, x \leftarrow \mathsf{alloc}(e)) \Downarrow (l', m \cup \{a \to 0^n\})}$$

Here, $\{a \to 0^n\}$ denotes a heap whose domain is $\{a\} \times \{0, \ldots, n-1\}$ and where all the defined values are 0. The premise $a = \mathsf{fresh}(l, m)$ states that the atom a is chosen in a way that guarantees that it is *fresh* relative to the current state. By fresh, we mean that a should not occur *anywhere* in the program state; formally, for all $n \in \mathbb{Z}$, $x \in \langle \mathrm{var} \rangle$ and $p \in \mathbb{A} \times \mathbb{Z}$:

$$l(x) \neq (a, n) \qquad m(p) \neq (a, n) \qquad m(a, n) \text{ is undefined}$$

In particular, notice that the new block cannot be pointed to by any pointer that way already present in the program state. For simplicity, we can assume that a is chosen according to a deterministic function, but we do not require anything beyond the above properties. (We could also formulate the rule for allocation non-deterministically; this wouldn't change much in what follows, since, in Section 5, we are going to recast the semantics into a deterministic form that is quotiented over the exact choice of pointer values.)

The $\cup$ operator denotes the (left-biased) union of two partial maps:

$$(m_1 \cup m_2)(p) = \begin{cases} m_1(p) & \text{if } p \in \mathrm{dom}(m_1) \\ m_2(p) & \text{otherwise} \end{cases}$$

Hence, $m \cup \{a \to 0^n\}$ denotes the result of adding a new block of size n to m pointed to by identifier a.

The allocation rule above can never fail due to out-of-memory errors, implying that memory can grow indefinitely during execution. Even so, we include a memory deallocation command to model the manual memory management features present in many

languages:

$$\frac{[\![e]\!](l) = (a, 0) \qquad m[a \to \bot] = m'}{(l, m, \mathsf{free}(e)) \Downarrow (l, m')}$$

We ensure that e denotes some pointer value under store l, which must be of the form $(a, 0)$ for some block identifier a. We remove the block associated with a from m, written $m[a \to \bot]$. Once again, this notation implicitly assumes that a currently has some block associated with it in m; thus, freeing a pointer twice causes a memory error.

$$\frac{[\![e]\!](l) = (a, 0) \quad \forall n.\, m(a, n) \text{ is undefined}}{(l, m, \mathsf{free}(e)) \Downarrow \mathsf{abort}}$$

Finally, it may occur that a program executing in some state does not lead to *any* final outcome. Usually, in big-step semantics, this could represent an execution that does not terminate or that ends in an error. Here, however, all errors are captured by the special value abort, and a pair (s, c) that is not related to anything via $\Downarrow$ represents indeed a non-terminating computation; in that case, therefore, we say that c *loops* on s.

3. Memory-Safe Reasoning, By Example

Informally, it is clear that the language presented above is memory safe: potential memory errors are immediately trapped when they occur. In common jargon, it ensures both *temporal* (preventing uses after free and guaranteeing that new blocks are initialized) and *spatial* (preventing out-of-bounds accesses) memory safety. We now explore some consequences of this observation.

Here's a small program written in our language, which could represent the example discussed in Section 1. Variable p is first initialized with a fresh object that we may want to protect, but right after that control is transferred to a (potentially unknown) command plugin:

$$p \leftarrow \mathsf{alloc}(1); [p] \leftarrow 42; \mathsf{plugin}$$

If we know that plugin does not mention variable p and the program terminates, we can assert that the object referenced by p still contains 42, regardless of what plugin did. This crucially relies on the fact that our language is memory safe and that p points to a *fresh* object, that couldn't have been referred elsewhere in the program state. Slightly more formally:

Principle 3.1 (Integrity)**.** *Let $s_1, s_2, s' \in \mathcal{S}$ be program states and $c \in \langle \mathrm{comm} \rangle$ a command. Suppose that $(s_1 * s_2, c) \Downarrow s'$, where $*$ denotes the union of two states, defined as*

$$(l_1, m_1) * (l_2, m_2) = (l_1 \cup l_2, m_1 \cup m_2).$$

Furthermore, suppose that all local variables used by c are defined in s_1, that the domains of the memories of s_1 and s_2 are disjoint, and that no pointers in s_1 point to objects in s_2. In this case, we can write s' as $s'_1 * s_2$ for some s'_1, where the memories of s'_1 and s_2 are disjoint. In other words, s_2 is not affected by the execution of c.

This principle is quite strong, allowing us to place a bound on the effect of a piece of code on its environment. But we can go further: Since our language does not allow users to cast pointers to integers, we know that the behavior of plugin cannot depend on that portion of the heap in any way—for example, plugin cannot distinguish between the program above and this similar one:

$$p \leftarrow \mathsf{alloc}(2); [p] \leftarrow 1729; [p + 1] \leftarrow 561; \mathsf{plugin}$$

Indeed, p could be initialized to contain *any* fresh object or collection of fresh objects, with an arbitrary topology, and plugin would not be able to tell the difference (as usual, modulo side channels such as timing channels that fall outside the scope of the model). More formally:

Principle 3.2 (Secrecy)**.** *Let $s_1, s_{21}, s_{22} \in \mathcal{S}$ be program states and c a command. Suppose that all local variables used by c are defined in s_1 and that no pointers in s_1 point to objects in s_{21} or s_{22}. If running c on $s_1 * s_{21}$ results in an error, then running it on $s_1 * s_{22}$ also results in an error. If c loops on $s_1 * s_{21}$, then it also loops on $s_1 * s_{22}$. Finally, if c terminates successfully on $s_1 * s_{21}$, it will do the same on $s_1 * s_{22}$, and the part of the program state affected by the execution of c (which does not include s_{21} or s_{22}) will be the same after both executions.*

Principles 3.1 and 3.2 articulate the well-known intuition that pointers in a memory-safe language can effectively be viewed as *capabilities* to the regions they point to. Knowing which capabilities a piece of code has (and doesn't have) allows us to determine more easily what it can and cannot do, without the need for understanding its behavior in great detail.

Finally, there are a number of intuitive equations that programmers expect to hold when reasoning about combined states. To describe them, we need a little notation. First, we say (informally) that two program states s_1 and s_2 are equivalent, written $s_1 \equiv s_2$, if they cannot be distinguished by any program executing on them. (We avoid giving a more detailed definition of equivalence here, appealing to intuition only; Section 5 justifies the equations presented here in a formal setting.) Suppose that s is some program state, which possibly mentions an atom a. We write $\nu a.\, s$ to denote the same state s while emphasizing the fact that the name a is *local* to s and cannot be referenced directly from the outside. Since the precise value of that name is only relevant locally, it makes sense to consider two states as being equivalent when they differ only by a local name:

$$\frac{[a/a_1]s_1 = [a/a_2]s_2 \qquad a \text{ does not appear in } s_1 \text{ or } s_2}{\nu a_1.\, s_1 \equiv \nu a_2.\, s_2}$$

Here, $[e/a]s$ denotes the familiar substitution operation, which replaces a by e in s. Further valid identities reflect the fact that marking as local a name that does not appear on a state does not matter, nor does the order in which we localize names:

$$\nu a.\, s \equiv s \text{ if } a \text{ does not occur in } s$$
$$\nu a_1.\, \nu a_2.\, s \equiv \nu a_2.\, \nu a_1.\, s$$

It is also valid to enlarge the scope of a local name to a bigger heap. More formally, if a does not appear in s':

$$(\nu a.\, s) * s' \equiv \nu a.\, (s * s') \qquad s' * (\nu a.\, s) \equiv \nu a.\, (s' * s)$$

Readers familiar with the π-calculus [12] will recognize these identities as describing the behavior of the name restriction operator ν and how it interacts with process composition.

We submit that these principles and equations are the main reason programmers care about memory safety: They guarantee that we can use the structure of the objects in a program to place strong boundaries between software components, limiting the amount of damage an attack or programming error can cause, without having to formally verify the entire system (including attacker-provided components). We return to this point more formally in Section 6. More generally, we believe that formal characterizations of memory safety in terms of specific bad behaviors that are prevented [4, 11, 14, etc.] can be made more useful by establishing an explicit connection between the low-level invariants they maintain and the high-level reasoning principles they enable. This is related to the well-known contrast between syntactic and semantic characterizations of type safety, as discussed for instance by Benton and Zarfaty [7].

To conclude, it is worth mentioning a subtle but important distinction between memory safety and type safety. Memory safety is sometimes regarded as a necessary part of type safety, and indeed type-safe languages are generally also memory safe, but this need not be the case. For instance, we could imagine a typed language with, say, dynamically allocated integer arrays that, for performance reasons, are not automatically initialized to known values (they start out with whatever values happen to be in that region of the memory from whatever it was last used for). This language might well be type safe in the sense that it satisfies sensible progress and preservation theorems; but it would not be memory safe, since the above principles could be violated in practice, as discussed in Section 7. Conversely, memory safety does not necessarily imply type safety: it does imply a clean distinction between pointers and other data, but aside from this the typing rules can be arbitrarily broken without breaking memory safety.

4. A Review of Nominal Sets

To give a more formal account of the reasoning principles discussed above, it is useful to formulate an alternative semantics where they are easier to apply. The problem with the original semantics is that it is too concrete: it explicitly talks about pointer values, even if those values are unimportant for how the program executes. Specifically, if we run a program on two states that are equal modulo a bijective atom renaming, the results of these two computations should also be equal modulo renaming; in particular, the integer values appearing on the results will be the same. Intuitively, this is a consequence of pointers being unforgeable (i.e., we cannot convert a non-pointer value to a pointer) and not observable (i.e., we cannot convert a pointer to a non-pointer value). Talking about explicit pointer values in the language semantics makes our reasoning principles harder to apply directly in a formal setting. For instance, when combining program states, we have to reason explicitly about pointer independence, and Principle 3.2 is only valid up to pointer renaming, since the exact pointer names that are assigned to allocated regions may depend on the state of the entire heap. Instead, we would like a semantics that is phrased at a level of abstraction comparable to how programmers intuitively reason about the memory: what matters is the overall *structure* of the heap, and not the exact pointer values that occur in it.

The theory of *nominal sets* [17] was originally introduced to give a formal account of syntactic binding constructs and operations that respect alpha-equivalence. The reasoning principles it enables allow us to consider program states at the appropriate level of abstraction, forgetting about the exact values of pointers where they don't matter. In this section, we give a brief review of nominal sets and set up the notation we will use in the rest of the paper; Pitts' book [18] provides a thorough account of the theory, for those interested in more details. In Section 5, we demonstrate how to apply these concepts to come up with a better semantics for the language of Section 2. We begin by defining nominal sets:

Definition 4.1. Let $\mathrm{perm}(\mathbb{A})$ be the set of *permutations* of $\mathbb{A}$, i.e., bijective functions π that fix all but finitely many elements of $\mathbb{A}$. A *nominal set* is a triple $(X, \cdot, \mathrm{supp})$, where X is a set, and:

- The function $\cdot$ is a $\mathrm{perm}(\mathbb{A})$-action on X; that is, a function from $\mathrm{perm}(\mathbb{A}) \times X$ to X such that

$$1 \cdot x = x \qquad (\pi_1 \circ \pi_2) \cdot x = \pi_1 \cdot (\pi_2 \cdot x).$$

  We will often refer to this action as a *renaming* operation.
- The function supp maps each $x \in X$ to a finite set $\mathrm{supp}(x) \subset \mathbb{A}$, its *support*, which is the smallest set satisfying

$$\forall \pi.\, (\forall a \in \mathrm{supp}(x).\, \pi(a) = a) \Rightarrow \pi \cdot x = x.$$

  In other words, a permutation π that doesn't affect elements in $\mathrm{supp}(x)$ should leave x fixed under $\pi \cdot -$. Equivalently [17], we can require that there exist *some* finite set of atoms $A \subset \mathbb{A}$ (not necessarily minimal) that satisfies the same property. We say that such a set *supports* x.

We will usually use X to refer to the entire nominal set structure, letting the context determine which action we are considering.

In a standard application of nominal sets, $\mathbb{A}$ would be used as a set of variables, and the group action $\cdot$ would model the application of a permutation to all variables occurring in a syntax tree. The support of a syntax tree would typically be its set of free variables, and the fact that this support is finite means that we can always choose a fresh variable name not occurring in that syntax tree. In what follows, we will often refer to this use of nominal sets to explain the intuition behind the main concepts of the theory.

In the present context, the choice of name for a given block in memory is only relevant to the portions of the program state that need to refer to that block; as far as the rest of the state is concerned, the chosen name is invisible and can be changed to any other, as long as it does not conflict with other names present in the program state. In a sense, the parts of a state that do not refer to a block view that block as bound [13]. Nominal sets allow us to describe exactly in which parts of a program state a certain name is relevant and where it can be renamed.

Table 1 summarizes some basic constructions on nominal sets, which we now explain. The leftmost column, labeled "Trivial," applies to sets whose elements contain no atoms, such as $\mathbb{Z}$ or $\{\mathsf{nil}\}$; the corresponding permutation action doesn't do anything. Product, disjoint union, and finite subsets are defined by simple point-wise lifting: to rename a pair with elements from nominal sets X and Y, we rename each of the components, and similarly for the disjoint union and for a set of elements of a nominal set. The more complicated cases are the nominal sets arising from functions, $\mathrm{perm}(\mathbb{A})$ and $X \rightharpoonup_{\mathrm{fin}} Y$. We can see a permutation acting on functions as performing a "change of coordinates": to apply some transformation f in a new coordinate system, we first convert from the new coordinate system to the old one (that is, apply $\pi^{-1} \cdot -$), then apply the transformation f, and finally convert back to the new coordinate system by applying $\pi \cdot -$. Together, these constructions provide a rich vocabulary for building nominal sets with complex structure.

We'll often want to consider elements of nominal sets that don't share any atoms:

Definition 4.2. If x and y are elements of nominal sets X and Y, we say that they are *fresh* with respect to each other, written $x \# y$, if $\mathrm{supp}(x) \cap \mathrm{supp}(y) = \emptyset$.

As an example, if $a \in \mathbb{A}$ is an atom representing a variable name, and t is an element of a nominal set of syntax trees, then $a \# t$ simply asserts that a does not appear free in t.

For manipulating nominal sets, there are two kinds of functions that interact with the renaming operation $\cdot$ in a nice way.

Definition 4.3 (Equivariant functions)**.** Let X and Y be nominal sets. A function $f : X \to Y$ is said to be *equivariant* if it is compatible with the permutation action, i.e., if, for every $x \in X$ and $\pi \in \mathrm{perm}(\mathbb{A})$, we have $f(\pi \cdot x) = \pi \cdot f(x)$.

Equivariance formalizes the intuition that functions that operate on syntax should not depend on any particular choice of variable names. For instance, suppose that we implement a compiler optimization that removes unnecessary arithmetic operations: It could take an expression such as $0 + (a_1 + a_2)$, where a_1 and a_2 represent program variables, and produce $a_1 + a_2$ as a result. If this transformation is equivariant, we know that swapping a_1 and a_2 in

Trivial (e.g. $\mathbb{Z}$)	$\mathbb{A}$	$X \times Y$	$X + Y$	$\mathcal{P}_{\mathrm{fin}}(X)$ (finite subsets of X)
$\pi \cdot x = x$	$\pi \cdot a = \pi(a)$	$\pi \cdot (x, y) = (\pi \cdot x, \pi \cdot y)$	$\pi \cdot \iota_i(x) = \iota_i(\pi \cdot x)$	$\pi \cdot X' = \{\pi \cdot x \mid x \in X'\}$
$\mathrm{supp}(x) = \emptyset$	$\mathrm{supp}(a) = \{a\}$	$\mathrm{supp}(x, y) = \mathrm{supp}(x) \cup \mathrm{supp}(y)$	$\mathrm{supp}(\iota_i(x)) = \mathrm{supp}(x)$	$\mathrm{supp}\,(X') = \bigcup_{x \in X'} \mathrm{supp}(x)$

$\mathrm{perm}(\mathbb{A})$	$X \to_{\mathrm{fin}} Y$
$\pi \cdot \pi' = \pi \circ \pi' \circ \pi^{-1}$	$(\pi \cdot f)(x) = \pi \cdot f(\pi^{-1} \cdot x)$
$\mathrm{supp}(\pi') = \{a \in \mathbb{A} \mid \pi'(a) \neq a\}$	$\mathrm{supp}(f) = \mathrm{supp}(\mathrm{dom}(f)) \cup \mathrm{supp}(\mathrm{im}(f))$

Table 1. Some nominal set constructions. X and Y denote arbitrary nominal sets.

the input will produce the same effect on the result; hence, running our optimizer on $0 + (a_2 + a_1)$ would produce $a_2 + a_1$, and not a completely different expression such as $(a_2 + a_1) + 0 + 0$.

We can relax equivariance and consider more general functions, which respect renaming by permutations that do not affect names in some *fixed* finite set A:

Definition 4.4 (Finitely supported functions)**.** Let X and Y be nominal sets. A function $f : X \to Y$ is said to have *finite support* if there exists a finite set of atoms $A \subset \mathbb{A}$ such that, for $\pi \in \mathrm{perm}(\mathbb{A})$ and $x \in X$, if $\pi \# A$, then $f(\pi \cdot x) = \pi \cdot f(x)$. In other words, such a π should leave f fixed under the $\mathrm{perm}(\mathbb{A})$-action $- \cdot -\ : \mathrm{perm}(\mathbb{A}) \times (X \to Y) \to (X \to Y)$ defined by $(\pi \cdot f)(x) = \pi \cdot f(\pi^{-1} \cdot x)$. Finitely supported functions form a nominal set under this action, which we write $X \to_{\mathrm{fs}} Y$.

By definition (cf. Table 1), requiring $\pi \# A$ simply means that $\pi(a) = a$ for every $a \in A$. In terms of the syntax intuition, we could view A as a set of "global" names that are fixed, whereas all others are locally scoped and can be potentially renamed.

To illustrate this, consider a function $e : \mathbb{A}^2 \to \mathbb{B}$ that tests two atoms for equality. This function is trivially equivariant, since renaming a pair of atoms according to the same permutation preserves equality. Now, suppose that we write a compiler pass to test whether some language primitive given by a global variable a_1 is mentioned anywhere in a piece of code; formally, this would involve using the partially applied function $e_{a_1} : \mathbb{A} \to \mathbb{B}$, which tests if some atom is equal to a_1. This function is *not* equivariant, since $e_{a_1}(a_1) = \mathsf{true}$, whereas renaming this argument to a distinct atom a_2 would result in false. However, this function does have finite support: any permutation that does not affect a_1 will preserve the behavior of e_{a_1}, hence its support is just $\{a_1\}$. More generally, partially applying an equivariant function of multiple arguments results in a function with finite support. Every equivariant function is trivially finitely supported: just take $A = \emptyset$.

We can find many examples of functions that do *not* have finite support—typically, functions that rely on additional structure imposed on atoms. For instance, if we enumerate all atoms, writing them $a_0, a_1, a_2, \ldots$, then the function defined by $a_n \mapsto a_{2n}$ does not have finite support. In terms of our language, such a function would correspond to manipulating pointers as integers, which we explicitly disallow.

With basic concepts of nominal sets in hand, we are now ready to review the construction that forms the cornerstone of the structured semantics in Section 5: the *free restriction algebra* of a nominal set [18, Section 9.5].

Definition 4.5 (Free restriction algebra)**.** Let X be a nominal set. Define an equivalence relation $\sim_\alpha$ on $\mathcal{P}_{\mathrm{fin}}(\mathbb{A}) \times X$ as follows:

$$\frac{\mathrm{supp}(x_1) \setminus A_1 = \mathrm{supp}(x_2) \setminus A_2 \qquad \pi \# \mathrm{supp}(x_1) \setminus A_1 \qquad \pi \cdot x_1 = x_2}{(A_1, x_1) \sim_\alpha (A_2, x_2)}$$

The *free restriction algebra* over X, written ρX, is the quotient of $\mathcal{P}_{\mathrm{fin}}(\mathbb{A}) \times X$ by $\sim_\alpha$. We write $[A, x]_\alpha$ for the equivalence class of the pair (A, x) in ρX. To simplify the notation, when $A = \emptyset$, we use x to refer to the equivalence class $[A, x]_\rho$ if no ambiguity arises. (The term "free restriction algebra" will be justified by Definition 4.10.)

Intuitively, the set A in (A, x) lists which atoms in x should be considered bound. Thus, in the above definition, we consider two pairs (A_1, x_1) and (A_2, x_2) to be equivalent if (1) the sets of atoms in x_1 and x_2 that are allowed to appear free are the same (notice that we ignore atoms that are listed in A_1 and A_2 but do not appear in x_1 or x_2), and (2) we can find a permutation π for transforming x_1 into x_2 without changing those atoms that we listed as being free. As stated above, spurious atoms in A do not matter—that is, if $a \notin \mathrm{supp}(x)$, then $[A, x]_\alpha = [A \cup \{a\}, x]_\alpha$.

It is not hard to show that this equivalence relation is invariant under renaming, which allows us to endow free restriction algebras with nominal structure.

Definition 4.6. Given a nominal set X, ρX carries a nominal structure satisfying

$$\pi \cdot [A, x]_\alpha = [\pi \cdot A, \pi \cdot x]_\alpha \quad \mathrm{supp}([A, x]_\alpha) = \mathrm{supp}(x) \setminus A.$$

Intuitively, to rename an element, it suffices to rename "under binders", and to apply the same operation to the variables that are marked as free. To compute the support of such an element, we compute $\mathrm{supp}(x)$, and then remove bound atoms (A).

In a discrete nominal set—one whose elements have empty support—this construction adds nothing new:

Lemma 4.7. *Let X be a nominal set such that $\forall x \in X.\ \mathrm{supp}(x) = \emptyset$. Then, $\rho X \cong X$.*

To use Definition 4.5 to support the new semantics, we must define some operations to manipulate values with bound names. To start, we notice that ρ is *functorial*, in the sense that we can apply well-behaved functions "under binders":

Definition 4.8. Let X, Y be nominal sets and $f : X \to_{\mathrm{fs}} Y$ a finitely supported function. We define a function $\rho f : \rho X \to_{\mathrm{fs}} \rho Y$ as follows: Given $x_\rho \in \rho X$, we choose A and x such that $x_\rho = [A, x]_\rho$ and $A \# f$, and set

$$\rho f [A, x]_\rho = [A, f(x)]_\rho.$$

Notice that we can always perform such a choice, since bound atoms can be freely permuted and there are infinitely many atoms. It can be shown that both equations are valid for all choices of x and A that satisfy the above constraints.

The result of a function f with finite support is mostly unaffected by renaming operations, as long as the corresponding permutation does not change elements in the support of f. Therefore, to guarantee that the above equations make sense and do not depend on a particular choice of bound atoms, we need to ensure that

the atoms in x that are bound $[A, -]_\rho$ do not appear in the support of f, which is precisely what we required above.

It is also useful to lift ρ to binary functions:

Definition 4.9. Given nominal sets X, Y, Z and $f : X \times Y \to_{\mathrm{fs}} Z$ with finite support, we define $\rho f : \rho X \times \rho Y \to_{\mathrm{fs}} \rho Z$ in the following way. Given $x_\rho \in \rho X$ and $y_\rho \in \rho Y$, we choose A_x, x, A_y and y such that $x_\rho = [A_x, x]_\rho$ and $y_\rho = [A_y, y]_\rho$, while ensuring that the bound names of x and y do not clash with each other, and do not appear in the support of f; formally:

$$(A_x, A_y) \# f \qquad A_x \# y \qquad A_y \# x.$$

Once again, such a choice of representatives can always be made. We then pose

$$\rho f([A_x, x]_\rho, [A_y, y]_\rho) = [A_x \cup A_y, f(x, y)]_\rho.$$

This equation is valid as long as the choice of representatives satisfies the above requirements.

The last construction we need, relating Definition 4.5, is what justifies the name "free restriction algebra": intuitively, it shows that we can always define an atom restriction operation on ρX which behaves like name restriction in the π-calculus:

Definition 4.10. Let X be a nominal set. The equivariant operation $\nu_X : \mathbb{A} \times \rho X \to \rho X$ is defined by

$$\nu_X(a, [A, x]_\rho) = [A \cup \{a\}, x]_\rho.$$

If $x \in \rho X$, we write $\nu a.\, x$ to mean $\nu_X(a, x)$.

Informally, an element $\nu a.\, x \in \rho X$ takes some $x \in \rho X$ that is parameterized by the choice of a fresh atom a and "quotients" x by all possible choices for a. We use this construction to describe elements of ρX by gradually specifying which atoms are bound. For instance, an expression like $\nu a_1.\, \nu a_2.\, \{a_1, a_2, a_3\}$ denotes a finite set with three atoms, two of which (a_1 and a_2) are local and can be freely renamed. We can then show the following:

Lemma 4.11. *Let X, Y, Z be nominal sets, $x \in \rho X$, and $y \in \rho Y$. Let $f \in X \to_{\mathrm{fs}} Y$ and $g \in X \times Y \to_{\mathrm{fs}} Z$ be functions with finite support. The following identities are valid for all $a_1, a_2, a \in \mathbb{A}$ ($(a_1\ a_2)$ denotes the permutation that swaps a_1 and a_2, leaving all other atoms fixed):*

$$\begin{aligned}
\nu a_1.\, x &= \nu a_2.\, (a_1\ a_2) \cdot x && \text{if } a_2 \# x \\
\mathrm{supp}(\nu a.\, x) &= \mathrm{supp}(x) \setminus a \\
\nu a_1.\, \nu a_2.\, x &= \nu a_2.\, \nu a_1.\, x \\
\nu a.\, x &= x && \text{if } a \# x \\
\rho f(\nu a.\, x) &= \nu a.\, \rho f(x) && \text{if } a \# f \\
\rho g((\nu a.\, x), y) &= \nu a.\, \rho g(x, y) && \text{if } a \# (g, y) \\
\rho g(x, (\nu a.\, y)) &= \nu a.\, \rho g(x, y) && \text{if } a \# (g, x)
\end{aligned}$$

5. A Semantics for Memory Safety

Now we have all the tools we need to formulate a new semantics for the language of Section 2 with a more abstract treatment of the heap. Using the basic constructions of Table 1 as building blocks, we can define a nominal set structure on the set of program states $\mathcal{S}$. We enrich this structure by applying the restriction construction of Definition 4.5 to mark some of the atoms appearing in a program state as local. From now on, we use $\mathcal{S}$ to denote the result of this construction—that is, $\mathcal{S} = \rho(\mathcal{L} \times \mathcal{M})$, where $\mathcal{L}$ and $\mathcal{M}$ are the original sets of local stores and memories with the evident nominal structures on them.

The main motivation for this new definition is enabling a rigorous treatment of the operations for describing states in Section 3 and related reasoning principles. Specifically, we lift the state operator $*$ to work under binders with Definition 4.9, using the restriction operator ν of Definition 4.10 to limit the scope of block identifiers. By virtue of Lemma 4.11, the equivalence laws that we postulated for this operator are valid as equations, not just (informal) equivalences; for example:

$$\frac{(a\ a_1) \cdot s_1 = (a\ a_2) \cdot s_2 \qquad a \# (s_1, s_2)}{\nu a_1.\, s_1 = \nu a_2.\, s_2} \qquad \frac{a \# s}{\nu a.\, s = s}$$

To describe state components, two definitions are useful.

Definition 5.1. There are two functions on states, $\mathrm{vars} : \mathcal{S} \to \mathcal{P}_{\mathrm{fin}}(\langle \mathrm{var} \rangle)$ and $\mathrm{pub} : \mathcal{S} \to \mathcal{P}_{\mathrm{fin}}(\mathbb{A})$, defined as follows. The first one lists the local variables that belong to a given state; it is defined by lifting the following function, remembering that $\rho\mathcal{P}_{\mathrm{fin}}(\langle \mathrm{var} \rangle) \cong \mathcal{P}_{\mathrm{fin}}(\langle \mathrm{var} \rangle)$ by Lemma 4.7:

$$(l, m) \mapsto \mathrm{dom}(l)$$

The second, pub, describes the set of objects of some program state that are public—that is, the set of names in the domain of the heap that are not bound. Formally, we lift

$$(l, m) \mapsto \mathrm{dom}(m)$$

and compose it with supp. By construction, $\mathrm{pub}([A, (l, m)]_\rho)$ contains atoms in the domain of m that are listed in A. We call $\mathrm{pub}(s)$ the set of *globally named* objects of s.

To understand the meaning of these definitions, consider the states s_1 and s_2 mentioned above. Since x is the only local variable of s_1, we have $\mathrm{vars}(s_1) = \{x\}$. On the other hand, s_2 defines no local variables, so $\mathrm{vars}(s_2) = \emptyset$. It is clear that $\mathrm{pub}(s_2) = \{a_1, a_2\}$. As for $\mathrm{pub}(s_1)$, s_1 contains two memory blocks in it, referenced by atoms a_1 and a_3. However, since a_1 is bound in s_1, it cannot appear in $\mathrm{pub}(s_1)$, implying $\mathrm{pub}(s_1) = \{a_3\}$. This set does not contain a_2 either, because, even though a_2 is not bound in s_1, it is not associated to any block stored in s_1.

We can show that state union satisfies some natural algebraic laws:

Lemma 5.2. *The operator $*$ is associative and has an identity element $\mathsf{emp} = (\{\}, \{\})$, where $\{\}$ denotes a partial function with empty domain.*

Proof. It suffices to choose sufficiently fresh representatives so that the equations of Definition 4.9 apply. For instance:

$$\begin{aligned}
& ([A_1, s_1]_\rho * [A_2, s_2]_\rho) * [A_3, s_3]_\rho \\
= \ & [A_1 \cup A_2, s_1 * s_2]_\rho * [A_3, s_3]_\rho \\
= \ & [(A_1 \cup A_2) \cup A_3, (s_1 * s_2) * s_3]_\rho \\
= \ & [A_1 \cup (A_2 \cup A_3), s_1 * (s_2 * s_3)]_\rho \\
= \ & [A_1, s_1]_\rho * ([A_2, s_2]_\rho * [A_3, s_3]_\rho)
\end{aligned}$$

Showing that emp is an identity works similarly. □

Lemma 5.3. *Let $s_1, s_2 \in \mathcal{S}$. Suppose that $\mathrm{vars}(s_1) \cap \mathrm{vars}(s_2) = \emptyset$ and $\mathrm{pub}(s_1) \# \mathrm{pub}(s_2)$. Then, $s_1 * s_2 = s_2 * s_1$.*

Most of the constructions used in the original semantics are equivariant, so we lift them directly to construct a semantics on $\mathcal{S}$ (the only exception is allocation, which we interpret using ν, as explained below). As an example, consider the rule from Section 2 for assigning the result of an expression to a local variable. It uses two operations: computing the value of an expression according to a local store, and updating the value of a partial function for a given argument. The latter is trivially equivariant, while the former can be shown to be so by simple structural induction.

To define a semantics for allocation, we can leverage the power of the ν operator to model the choice of fresh atom for the new

memory block; by construction, this will yield a deterministic interpretation of alloc where no particular atom value is chosen (or, if you like, where all possible atoms are chosen at once) and the scope of this immaterial atom is restricted to a small portion of the program state.

$$\frac{\rho[\![e]\!](s) = n \qquad n > 0}{(s, x \leftarrow \mathsf{alloc}(e)) \Downarrow (\nu a.\, (x \to (a, 0)) * \{a \to 0^n\}) * s}$$

Here, $\rho[\![e]\!](s)$ denotes the result of lifting the expression evaluation operation to states with bound names and applying it to s. The expression $\nu a.\, (x \to (a, 0)) * \{a \to 0^n\}$ denotes a state (call it s') that has a single local variable x, whose value is a pointer $(a, 0)$, and a memory consisting of a single block of n zeros at location a. Since a is within the scope of ν, this name is local to s'. Hence, by writing $s' * s$ in the right-hand side of this rule, we are *adding* a new memory block to s, making it explicit that its name does not occur in the rest of the state.

Because a is hidden in s', we know by Lemma 4.11 that $\mathrm{supp}(s') = \emptyset$, since the other elements appearing on its definition (x, 0 and n) all have empty support; in particular, $\pi \cdot s' = s'$ for every permutation π. Hence, the rule defining allocation corresponds to an equivariant partial function $\mathcal{S} \rightharpoonup \mathcal{S}$. As a consequence:

Theorem 5.4 (Renaming machine states)**.** *The semantics of the language, seen as a function, is equivariant. Specifically, let $s \in \mathcal{S}$ be a program state and c a command. If $(s, c) \Downarrow s'$, then $(\pi \cdot s, c) \Downarrow \pi \cdot s'$ for every permutation π. If $(s, c) \Downarrow \mathsf{abort}$, then $(\pi \cdot s, c) \Downarrow \mathsf{abort}$. If c loops on s, then c loops on $\pi \cdot s$.*

Proof. By induction on the derivation of the execution relation $\Downarrow$. Most cases are trivial, since the corresponding semantics is defined by lifting, which preserves equivariance. The case for allocation follows from the previous observation. □

It is easy to see that the semantics is deterministic; we can also show that it respects name restriction, in the following sense:

Theorem 5.5 (Restriction)**.** *Let $a \in \mathbb{A}$ be an atom, $s \in \mathcal{S}$ a program state, and $c \in \langle \mathrm{comm} \rangle$ a command. If $(s, c) \Downarrow s'$ for some s', then $((\nu a.\, s), c) \Downarrow (\nu a.\, s')$. If $(s, c) \Downarrow \mathsf{abort}$, then $((\nu a.\, s), c) \Downarrow \mathsf{abort}$. If c loops on s, then it also loops on $\nu a.\, s$.*

Proof. The result follows trivially for almost all of the rules defining the semantics, since they are defined by lifting (Definition 4.8), and lifting functions commute with restriction, thanks to Lemma 4.11; the only exception is allocation. In that case, the newly allocated block has empty support, so we can apply Lemma 4.11 for the binary lift case and conclude. □

We discussed in Sections 1 and 3 that memory-safe languages enable local reasoning (which we argued to be the defining characteristic of memory safety). This claim is formalized in the following *frame theorems*, which allow us to analyze the effect of a program on some state by breaking the state into smaller components and analyzing each of them separately. These properties are *extensional*, in the sense that they talk directly about the observable behavior of programs of this language.

First, an auxiliary lemma that explains how vars and pub behave with respect to execution:

Lemma 5.6. *Let $s, s' \in \mathcal{S}$ be program states and c a command, with $(s, c) \Downarrow s'$. If $\mathrm{fv}(c) \subseteq \mathrm{vars}(s)$, then $\mathrm{vars}(s') = \mathrm{vars}(s)$. Furthermore, the set of globally named objects of s, as well as its support, decrease; formally, $\mathrm{pub}(s') \subseteq \mathrm{pub}(s)$, and $\mathrm{supp}(s') \subseteq \mathrm{supp}(s)$.*

Proof. The first conclusion of the lemma makes sense intuitively, but the second one might seem strange, given that c could allocate new memory. However, since the semantics of allocation is given by the ν operator, it does not introduce new visible block names. All other commands preserve the set of globally visible objects, with the exception of free, which may remove an object. The result thus follows by straightforward induction. □

Now, the first frame theorem states that we can always extend the initial state of a successful execution; moreover, this additional state will be left intact at the end of the execution but won't affect the rest of the result in any way. Formally:

Theorem 5.7 (Frame OK)**.** *Suppose that $(s_1, c) \Downarrow s_1'$, with $s_1, s_1' \in \mathcal{S}$, and $\mathrm{fv}(c) \subseteq \mathrm{vars}(s_1)$. Then, for every $s_2 \in \mathcal{S}$ such that $\mathrm{pub}(s_1) \# \mathrm{pub}(s_2)$,* [1] *$(s_1 * s_2, c) \Downarrow (s_1' * s_2)$.*

Proof. By induction on the size of the derivation of $\Downarrow$. In each case, we choose representatives for s_1 and s_2 so that their bound atoms do not conflict, and we analyze the result of applying the various lifted operations to them. We analyze a few representative cases. We begin by noticing that, since $\mathrm{fv}(c) \subseteq \mathrm{vars}(s_1)$, we have

$$\rho[\![e]\!](s_1 * s_2) = \rho[\![e]\!](s_1).$$

For a command like $[e] \leftarrow e'$, which stores a value in memory, we know that the stored location must already have been defined in s_1 (otherwise the command would have resulted in abort), hence extending it with s_2 does not change the effect of the command on the original state. The same observation applies to loading a value from memory. When freeing a region of memory ($c = \mathsf{free}(e)$), we additionally use our hypothesis to guarantee that no portions of s_2 will be removed. Finally, if $c = c_1; c_2$, we use Lemma 5.6 to reestablish the hypotheses needed to proceed inductively and conclude. □

A similar argument yields a second frame theorem, which intuitively says that we cannot make a non-terminating execution into a terminating one by adding data to its initial state:

Theorem 5.8 (Frame Loop)**.** *Suppose c loops on s_1, for $s_1 \in \mathcal{S}$ and $c \in \langle \mathrm{comm} \rangle$. Then c loops on $s_1 * s_2$, for every $s_2 \in \mathcal{S}$ with $\mathrm{pub}(s_1) \# \mathrm{pub}(s_2)$.*[2]

The two frame theorems we've stated so far don't actually tell us anything fundamentally different about memory-safe languages, compared to non-memory-safe ones. For instance, it should be possible to derive an analog of these two theorems for a fragment of C that does not allow low-level tricks such as casting pointers to integers (modulo memory-exhaustion errors). What makes the memory-safe language special is that extending a state doesn't affect the *error behavior* of programs:

Theorem 5.9 (Frame Error)**.** *Suppose that $(s_1, c) \Downarrow \mathsf{abort}$, with $\mathrm{fv}(c) \subseteq \mathrm{vars}(s_1)$, and suppose $s_1 \# \mathrm{pub}(s_2)$ for some $s_2 \in \mathcal{S}$. Then $(s_1 * s_2, c) \Downarrow \mathsf{abort}$.*

Proof. By induction on the $\Downarrow$ derivation. The most interesting cases are commands that can generate errors directly, such as storing to

[1] Requiring that $\mathrm{pub}(s_1) \# \mathrm{pub}(s_2)$ intuitively means that no memory block is stored in s_1 and s_2 at the same time. This is a bit stronger than the usual hypothesis of the familiar "frame property" [25] of separation logic, which only requires memories to store distinct *addresses* (as opposed to entire objects), but it is necessary in our context because deallocation frees an entire object at once, as opposed to a single address, as it is done in some presentations of separation logic.

[2] This hypothesis is not, strictly speaking, necessary, but omitting it would require a stronger version of Theorem 5.7, which we don't need here.

a memory location or loading from it; most other cases are similar to Theorem 5.7.

Because $s_1 \# \mathrm{pub}(s_2)$, we know that a pointer value obtained by evaluating an expression cannot point to an object in $\mathrm{pub}(s_2)$. So if, for instance, loading through that pointer raises an error when running on s_1 alone, it will also raise one when running on $s_1 * s_2$, and similarly for the other commands that manipulate memory. If the hypothesis were omitted, it could be the case that the execution of c on s_1 alone tries to access an object that does not exist in s_1 but does exist in s_2. In this case, execution would succeed on $s_1 * s_2$ even though it failed on s_1 alone.

The other interesting case is when $c = c_1; c_2$. The error might have occurred when executing either c_1 or c_2. If it was in c_1, we can apply the induction hypothesis and conclude directly; otherwise, we can find s_1' such that $(s_1, c_1) \Downarrow s_1'$. Combining Theorem 5.7 with Lemma 5.6 and the induction hypothesis, we obtain

$$(s_1 * s_2, c_1) \Downarrow (s_1' * s_2) \qquad (s_1' * s_2, c_2) \Downarrow \mathsf{abort}.$$

Combining these two facts yields $(s_1 * s_2, c_1; c_2) \Downarrow \mathsf{abort}$, which concludes this case. □

On its own, the fact that erroneous executions are preserved by state extension might not seem very interesting. But, when combined with the other two results, it has the following deep consequence, which summarizes Principles 3.1 and 3.2 in a single result:

Corollary 5.10 (Noninterference)**.** *Let $s_1, s_{21}, s_{22}, s' \in \mathcal{S}$ be program states and $c \in \langle \mathrm{comm} \rangle$ a program. Suppose that $\mathrm{fv}(c) \subseteq \mathrm{vars}(s_1)$, $s_1 \# \mathrm{pub}(s_{21})$, $\mathrm{pub}(s_1) \# \mathrm{pub}(s_{22})$, and $(s_1 * s_{21}, c) \Downarrow s'$. Then we can find $s_1' \in \mathcal{S}$ such that $s' = s_1' * s_{21}$ and $(s_1 * s_{22}, c) \Downarrow s_1' * s_{22}$. Informally, the inaccessable parts of the state, s_{21} and s_{22}, can be replaced arbitrarily without affecting the program's result.*

Proof. Notice that $s_1 \# \mathrm{pub}(s_{21})$ implies $\mathrm{pub}(s_1) \# \mathrm{pub}(s_{21})$, one of the hypotheses needed for Theorems 5.7 and 5.8. Consider the result of executing c on s_1. It can't be the case that c loops on s_1 or that $(s_1, c) \Downarrow \mathsf{abort}$. For otherwise we would be able to apply Theorems 5.8 and 5.9 to conclude that c loops on $s_1 * s_{21}$ or that $(s_1 * s_{21}, c) \Downarrow \mathsf{abort}$, contradicting our hypothesis since the semantics is the deterministic. Therefore, there must be a state s_1' such that $(s_1, c) \Downarrow s_1'$. Thanks to Theorem 5.7, we can conclude that $(s_1 * s_{2i}, c) \Downarrow s_1' * s_{2i}$ for $i = 1, 2$. Once again, because the semantics is deterministic, this implies $s' = s_1' * s_{21}$, which concludes our proof. □

6. A Taste of Program Verification

To further explore the kind of reasoning supported by our semantics, we demonstrate how we can use it for simple program verification. Rather than developing full-blown variants of separation logic and its assertion language that are tailored for this semantics (a larger project that is out of the scope of this paper), we reason directly over individual *program states*. Fortunately, name restriction provides a convenient language for describing data structures without caring about the exact pointer values they contain, partially making up for the lack of a proper program logic.

Before working out a concrete example, we recast some of the reasoning principles from above in the form of inference rules. Since these are mostly direct consequences of the results of Section 5, or well-known principles of Hoare logic, we state them without proof. We begin with an analog of Hoare triples:

Definition 6.1 (Triples)**.** Suppose s and s' are program states, and c is a command. We write $\{s\}\ c\ \{s'\}_i$ to indicate that c, started in state s, "potentially terminates" in state s'. The exact meaning of the triple depends on the index $i \in \{t, l, e, le\}$, which lists the effects that can happen during execution: the triple can be a *total correctness* result ($i = t$, meaning that the program terminates in a non-erroneous state), or it can allow for infinite loops, errors, or both ($i \in \{l, e, le\}$).

$$\begin{aligned}
\{s\}\ c\ \{s'\}_t &\iff (s, c) \Downarrow s' \\
\{s\}\ c\ \{s'\}_l &\iff (\forall o.\, (s, c) \Downarrow o \Rightarrow o = s') \\
\{s\}\ c\ \{s'\}_e &\iff (s, c) \Downarrow s' \vee (s, c) \Downarrow \mathsf{abort} \\
\{s\}\ c\ \{s'\}_{le} &\iff (\forall s''.\, (s, c) \Downarrow s'' \Rightarrow s'' = s')
\end{aligned}$$

We order triple indices as $t \le l, e \le le$, writing $i_1 \vee i_2$ for the least upper bound of two indices. These indices induce a form of subsumption on triples:

$$\frac{\{s\}\ c\ \{s'\}_i \qquad i \le i'}{\{s\}\ c\ \{s'\}_{i'}}$$

Some reasoning principles carry directly from Hoare logic:

$$\frac{\{s\}\ c_1\ \{s'\}_{i_1} \qquad \{s'\}\ c_2\ \{s''\}_{i_2}}{\{s\}\ c_1; c_2\ \{s''\}_{i_1 \vee i_2}} \qquad \frac{}{\{s\}\ \mathsf{skip}\ \{s\}_i}$$

Theorem 5.5 justifies an analog of the familiar rule of auxiliary variable elimination:

$$\frac{\{s\}\ c\ \{s'\}_i}{\{\nu a.\, s\}\ c\ \{\nu a.\, s'\}_i}$$

In the absence of errors (i.e., when $i \in \{t, l\}$), we have a version of the frame rule from separation logic:

$$\frac{\{s_1\}\ c\ \{s_1'\}_i \qquad i \in \{t, l\} \qquad \mathrm{fv}(c) \subseteq \mathrm{vars}(s_1) \qquad \mathrm{pub}(s_1) \# \mathrm{pub}(s_2)}{\{s_1 * s_2\}\ c\ \{s_1' * s_2\}_i}$$

Moreover, in our new setting, we can *extend* the frame rule to work even in the presence of errors, by strengthening its precondition:

$$\frac{\{s_1\}\ c\ \{s_1'\}_i \qquad \mathrm{fv}(c) \subseteq \mathrm{vars}(s_1) \qquad s_1 \# \mathrm{pub}(s_2)}{\{s_1 * s_2\}\ c\ \{s_1' * s_2\}_i}$$

This rule forms a basis for partial reasoning in a memory-safe setting. Consider the following tautology, valid for all programs c and states s:

$$\exists s'.\, \{s\}\ c\ \{s'\}_{le}$$

Informally, we can see this principle as saying that it is always safe to assume that our program produces some answer we know nothing about, as long as we are willing to include errors and non-termination as acceptable outcomes. In a more conventional Hoare logic setting, this would correspond to the following proof principle, valid for any precondition p:

$$\frac{}{\{p\}\ c\ \{\mathsf{true}\}}\ \textsc{Taut}$$

Unfortunately, in separation logic, the soundness of the frame rule relies crucially on the fact that specifications guarantee that no memory errors occur during execution, something that can only be enforced in that setting through proof, and that is certainly *not* valid for all programs. Therefore, we must reject the TAUT rule above to guarantee that proofs remain valid.

We can appreciate the benefit of this rule by analyzing its implications for the plugin example of Section 3. Let s be the portion of the program state that the plugin is supposed to manipulate. Suppose that all the variables of plugin are defined in s (which can be trivially enforced by static analysis). According to the above rule, we can assume that there exists some state s' such that

$$\{s\}\ \mathsf{plugin}\ \{s'\}_{le}.$$

Allocating and initializing the p object on the same starting state s would have the effect of adding a state component $s_p = \nu a.\,(p \to (a, 0)) * \{a \to 42\}$ to it. But by construction, this new component has no globally visible objects (the only atom that appears in it, a, is bound). Therefore, the preconditions of the weak frame rule above are trivially satisfied, allowing us to conclude that

$$\{s * s_p\}\ \mathsf{plugin}\ \{s' * s_p\}_{le}.$$

This result is a formal statement expressing the kind of isolation guarantee that we expect to hold in a memory-safe setting.

To conclude with something a little more interesting, let's consider a classical example that is often used to illustrate separation logic: verifying the total correctness of a list reverse function. The following program listrev starts with a linked list of values whose head is stored in variable x; it performs an in-place reversal of that list and terminates with a pointer to the reversed list in the variable r, which we assume to be initialized with nil:

```
while x ≠ nil do
  y ← [x + 1]; [x + 1] ← r; r ← x;
  x ← y; y ← nil
end
```

(The last command can be omitted, but this would complicate the proof, since our specifications need to mention the state of all local variables.)

To specify and verify this program, we use the following helper functions, which build an entire program state representing a list of values vs:

$$\begin{aligned} h(a, [\,]) &= \mathsf{nil} \\ h(a, v :: vs') &= (a, 0) \\ b(a, [\,]) &= \mathsf{emp} \\ b(a, v :: vs') &= \nu a'.\,\{a \to v, h(a', vs')\} * b(a', vs') \\ l(x, vs) &= \nu a.\,(x \to h(a, vs)) * b(a, vs) \end{aligned}$$

Here, $[\,]$ denotes the empty list and $v :: vs'$ the list obtained by adding element v in front of the list vs'. The function $h : \mathbb{A} \times \mathrm{list}(\mathcal{V}) \to \mathcal{V}$ (for "head") takes as input an atom a and a list of values vs, and returns a pointer to the head of list vs. If the list is empty, that pointer is just nil; otherwise, we return a pointer to the region denoted by a, where by convention we will store the head of the list.

Function $b : \mathbb{A} \times \mathrm{list}(\mathcal{V}) \to \mathcal{S}$ (for "body") takes as input an atom a and a list vs and produces a program state that has no variables, just a memory where vs is stored, starting at the region denoted by a. If the list is empty, there is nothing to store, so we just return the empty state emp. If vs is not empty, it has the form $v :: vs'$. We first generate a fresh atom a' with the ν operator, for holding the tail vs'. We then create a new state for storing vs' at that location, with the recursive call $b(a', vs')$. Finally, we store the head of the list at location a: the expression $\{a \to v, h(a', vs')\}$ represents a program state consisting of a single cons cell, whose first element is precisely v and whose second element points to the rest of the list.

The last function, $l : \langle \mathrm{var} \rangle \times \mathrm{list}(\mathcal{V}) \to \mathcal{S}$, takes as input a variable and a list of values. It generates a fresh atom a that can be used for storing the head of the list, uses b to construct the state that stores vs, and stores a pointer to the head of the list at x.

The interesting thing about the functions that we've just defined is that they completely describe a piece of state that represents a linked list, but without referring to any concrete pointer values. Normally, such a description would be given as a *predicate* over program states, in order to avoid describing lists that are stored only at a particular location of the memory. Here, because exact atom values are immaterial, we are able to describe the process of storing a list in memory as a *function*. (One limitation of this definition is that it doesn't allow for representing lists whose values may point to the list's own cons cells. We could address this shortcoming by mentioning explicitly where all cons cells are stored, or by considering lists whose elements are either cons cells or indices into the list, which would be interpreted as the corresponding pointer while the heap is built.)

We can now show the following by induction on vs:

Lemma 6.2. *The heap-building functions h, b and l are equivariant. Furthermore, let $x \in \langle \mathrm{var} \rangle$, $a \in \mathbb{A}$, and $vs \in \mathrm{list}(\mathcal{V})$. Then:*

$$\begin{aligned} &\mathrm{supp}(h(i, vs)) \subseteq \{i\} && \mathrm{supp}(l(x, vs)) \subseteq \mathrm{supp}(vs) \\ &\mathrm{supp}(b(i, vs)) \subseteq \{i\} \cup \mathrm{supp}(vs) && \mathrm{pub}(l(x, vs)) \subseteq \emptyset \\ &\mathrm{pub}(b(i, vs)) \subseteq \{i\} && \mathrm{vars}(l(x, vs)) = \{x\} \\ &\mathrm{vars}(b(i, vs)) = \emptyset \end{aligned}$$

The specification of the list reverse function is

$$\begin{aligned} &\{l(x, vs) * (r \to \mathsf{nil}) * (y \to \mathsf{nil})\} \\ \mathsf{listrev}\quad &\{(x \to \mathsf{nil}) * l(r, \mathrm{rev}(vs)) * (y \to \mathsf{nil})\}_t, \end{aligned}$$

where vs is universally quantified and $\mathrm{rev} : \mathrm{list}(\mathcal{V}) \to \mathrm{list}(\mathcal{V})$ is the mathematical list reversal function. We don't formalize proof rules for state-modifying commands, arguing instead directly through the semantics; a complete, useful set of proof rules in this style is left for future work. We can satisfy this obligation by proving a generalization

$$\begin{aligned} &\{l(x, vs_x) * l(r, vs_r) * (y \to \mathsf{nil})\} \\ \mathsf{listrev}\quad &\{(x \to \mathsf{nil}) * l(r, \mathrm{rev}(vs_x) +\!\!+ vs_r) * (y \to \mathsf{nil})\}_t, \end{aligned}$$

by structural induction on vs_x, where $+\!\!+$ denotes list concatenation. If $vs_x = [\,]$, the state at the beginning of the loop becomes

$$\begin{aligned} &(\nu a.\,(x \to \mathsf{nil}) * \mathsf{emp}) * l(r, vs_r) * (y \to \mathsf{nil}) \\ &= (x \to \mathsf{nil}) * l(r, \mathrm{rev}([\,]) +\!\!+ vs_r) * (y \to \mathsf{nil}). \end{aligned}$$

Since the value of x is nil, the condition of the loop will evaluate to false, leading to the ending state that we wanted.

If, on the other hand, $vs_x = v :: vs'_x$, the starting state becomes

$$\begin{aligned} s_0 = \nu a_1.\,\nu a_2.\,\nu a_3.\,&(x \to (a_1, 0)) * \{a_1 \to v, h(a_2, vs'_x)\} \\ &* b(a_2, vs'_x) * (r \to h(a_3, vs_r)) * b(a_3, vs_r) * (y \to \mathsf{nil}). \end{aligned}$$

The induction hypothesis tells us that

$$\begin{aligned} &\{l(x, vs'_x) * l(r, v :: vs_r) * (y \to \mathsf{nil})\} \\ \mathsf{listrev}\quad &\{(x \to \mathsf{nil}) * l(r, \mathrm{rev}(vs'_x) +\!\!+ v :: vs_r) * (y \to \mathsf{nil})\}_t. \end{aligned}$$

Since $\mathrm{rev}(vs_x) +\!\!+ v :: vs_r = \mathrm{rev}(v :: vs_x) +\!\!+ vs_r$, and the loop condition evaluates to true on the starting state, it suffices to show that the loop body body satisfies $\{s_0\}$ body $\{s_1\}$, where $s_1 = l(x, vs'_x) * l(r, v :: vs_r) * (y \to \mathsf{nil})$. Once again, the equations of Lemma 4.11 yield the equivalent expression

$$\begin{aligned} s_1 = \nu a_1.\,\nu a_2.\,\nu a_3.\,&(x \to h(a_2, vs'_x)) * b(a_2, vs'_x) \\ &* (r \to (a_1, 0)) * \{a_1 \to v, h(a_3, vs_r)\} * b(a_3, vs_r) \\ &* (y \to \mathsf{nil}). \end{aligned}$$

Thanks to Theorem 5.5 and the corresponding proof rule, it suffices to show that the specification is valid for universally quantified and sufficiently fresh atoms a_1, a_2 and a_3, as opposed to "local" ones. Furthermore, since the heap fragments denoted by $b(a_2, vs'_x)$ and $b(a_3, vs_r)$ are not affected by the execution of the body, and we do not allow memory errors in this program, we can appeal to the strong version of the frame rule (and Lemmas 5.3 and 6.2) to remove these fragments from consideration, focusing on the part of the state that changes. Figure 2 sketches how the body transforms the state, finishing the case and the proof.

$$
\begin{aligned}
&\{(x \to (a_1, 0)) * \{a_1 \to v, h(a_2, vs'_x)\} * (r \to h(a_3, vs_r)) * (y \to \mathsf{nil})\} && y \leftarrow [x+1] \\
&\{(x \to (a_1, 0)) * \{a_1 \to v, h(a_2, vs'_x)\} * (r \to h(a_3, vs_r)) * (y \to \boxed{h(a_2, vs'_x)})\} && [x+1] \leftarrow r \\
&\{(x \to (a_1, 0)) * \{a_1 \to v, \boxed{h(a_3, vs_r)}\} * (r \to h(a_3, vs_r)) * (y \to h(a_2, vs'_x))\} && r \leftarrow x \\
&\{(x \to (a_1, 0)) * \{a_1 \to v, h(a_3, vs_r)\} * (r \to \boxed{(a_1, 0)}) * (y \to h(a_2, vs'_x))\} && x \leftarrow y \\
&\{(x \to \boxed{h(a_2, vs'_x)}) * \{a_1 \to v, h(a_3, vs_r)\} * (r \to (a_1, 0)) * (y \to h(a_2, vs'_x))\} && y \leftarrow \mathsf{nil} \\
&\quad \{(x \to h(a_2, vs'_x)) * \{a_1 \to v, h(a_3, vs_r)\} * (r \to (a_1, 0)) * (y \to \boxed{\mathsf{nil}})\} \\
&\quad = \{(x \to h(a_2, vs'_x)) * (r \to (a_1, 0)) * \{a_1 \to v, h(a_3, vs_r)\} * (y \to \mathsf{nil})\}
\end{aligned}
$$

Figure 2. Correctness of the list reversal loop body. The expression on the left denotes the program state before executing the corresponding command on the right. The highlighted parts on each state expression show what changed when executing a command. The last equation follows from commuting the components of the state expression, using Lemmas 5.3 and 6.2.

7. Memory Safety in Practice

We expect our results to carry over to more realistic languages and systems that enforce spatial and temporal memory safety and guarantee that pointers cannot be observed—e.g., managed languages with garbage collection—modulo the presence of implementation bugs and tricky features that are not too hard to avoid by programming carefully (such as OCaml's `Obj.magic`). However, many practical systems enforce some degree of memory safety, without providing all the guarantees required for the results of Section 5 to hold. In this section, we examine how these results could change by modifying some features of the language of Section 2. It is beyond the scope of the present work to make a formal model to state all the claims made here and justify them through proof; instead, we keep this discussion mostly at an informal level.

7.1 Observing Pointer Values

The language studied above maintains a complete separation between pointers and other kinds of values. In real languages, however, this separation is often only one way. For example, some tools for enforcing memory safety of C programs [8, 14] allow pointer-to-integer casts, a feature required by many low-level C idioms; additionally, languages considered as memory safe often include features that must be avoided to completely enforce this separation (e.g., some implementations of `hashCode()` in Java, or OCaml's unsafe `Obj.magic` primitive). To model such features, we depart from the structured semantics of Section 5, going back to the explicitly named version of Section 2 for the reminder of this subsection. We extend the syntax of expressions with a cast operator

$$\langle \mathrm{exp} \rangle ::= \cdots \mid \mathsf{cast}(\langle \mathrm{exp} \rangle) \mid \cdots$$

and we assume that we have some function $[\![\mathsf{cast}]\!] : \mathbb{A} \times \mathbb{Z} \to \mathbb{Z}$ for converting a pointer to an integer, which we use to define the semantics of cast:

$$[\![\mathsf{cast}(e)]\!](l) = [\![\mathsf{cast}]\!](a, n) \qquad \text{if } [\![e]\!](l) = (a, n)$$

The reason we have to abandon the structured semantics based on nominal sets here is that, after we introduce cast, expression evaluation ceases to be an equivariant operation, something we crucially relied upon in Section 5. Many important properties of the language are now lost: specifically, Theorems 5.7 to 5.9 do not hold, since the state of unreachable parts of the heap may influence integer values observed by the program. An important consequence is that secrecy (Principle 3.2) is not valid in this language: An attacker could exploit the pointer channel to learn about data they shouldn't have access to.

Nevertheless, we can still guarantee the *integrity* of certain parts of the program state:

Theorem 7.1 (Weak Frame)**.** *Let* $(l_1, m_1), (l_2, m_2), (l', m') \in \mathcal{S}$ *be states and* $c \in \langle \mathrm{comm} \rangle$ *a command such that* $\mathrm{fv}(c) \subseteq \mathrm{dom}(l_1)$, $(l_1, m_1) \# \mathrm{dom}(m_2)$ *and* $(l_1 \cup l_2, m_1 \cup m_2, c) \Downarrow (l', m')$*. Then we can find* $(l'_1, m'_1) \in \mathcal{S}$ *such that* $l' = l'_1 \cup l_2$ *and* $m' = m'_1 \cup m_2$*.*

Proof. By induction on the derivation of $\Downarrow$. The proof follows those of Theorems 5.7 and 5.9, but without the extra structure given by the nominal set semantics. □

Notice that this result is strictly weaker than previous ones, such as Corollary 5.10: it does not allow us to assert *anything* about a related execution that starts at a *different* extended state $(l_1 \cup l'_2, m_1 \cup m'_2)$. To understand why this is the case, consider what would happen if we tried to show that $(l_1, m_1, c) \Downarrow (l''_1, m''_1)$—in other words, that c produces *some* result on (l_1, m_1) even in the absence of the other state component, (l_2, m_2). In the alloc case, we cannot guarantee that the fresh atom value returned for the smaller state will be the same as the one returned for the bigger one, which would require us to generalize our result to allow for atom permutations when comparing both executions. But because we allow casts, expression evaluation ceases to be an equivariant operation, meaning that the result for the bigger state could be completely different from the result for the smaller one after a local assignment, for instance.

7.2 Uninitialized Memory

Traditionally, memory-safe languages require variables and objects to be initialized before they are used—this is part of what is meant by temporal memory safety. But this can degrade performance for some applications, leading many systems to drop this feature—including not only standard implementations of C, but also implementations that provide some memory-safety guarantees [8, 14].

The problem with uninitialized memory is that it breaks the abstraction of the program state as consisting solely of the local variables and allocated objects: the entire memory becomes relevant to the execution of a program, and reasoning locally becomes much harder. By inspecting old values living in uninitialized memory, an attacker can learn about parts of the program state they shouldn't have access to, a direct violation of secrecy (Principle 3.2). This issue becomes even more severe if old pointers or other kinds of capabilities are allowed to occur in fresh memory, since they could be used to access restricted resources directly, leading to potential integrity violations as well.

7.3 Infinite Memory

The most idealized aspect of the language we consider is that memory can grow indefinitely and allocation never fails. In practice, computers have finite resources, and there is a bound on how much memory a program can use, forcing allocation to be partial. This

means that Theorem 5.7 can never hold in a real programming language as is, since executing a program in an extended state can cause it to run out of memory. Nevertheless, it should be possible to weaken that result to hold in a more realistic setting. For instance, if the allocator we consider were partial, we could explicitly require in addition to the other hypotheses of Theorem 5.7 that executing the program in an extended state does not result in an error (i.e., that $\neg(s_1 * s_2, c) \Downarrow \mathsf{abort}$). In any case, the best we can hope for in this setting is enforcing secrecy and integrity modulo memory-exhaustion errors.

How this limitation affects secrecy and integrity in practice will depend on the particular memory-safe system under consideration. An adversary that can cause a system to allocate large regions of memory could launch a denial-of-service attack, which could be seen as a violation of integrity. Furthermore, every memory-exhaustion error that is observed leaks one bit of information about the state of the entire system; depending on how such errors are triggered and how often, they could be used by adversaries to learn about data that should remain hidden. In Java for instance, running out of memory triggers an `OutOfMemoryError` exception, which can be caught and handled without causing the program to terminate. Hence, in that setting, we could use memory exhaustion to continuously monitor the state of a program during execution.

This discussion suggests that if such vulnerabilities are a real concern for a system, they need to be treated with special care. One approach would be to provide a mechanism to limit the amount of memory an untrusted component can allocate (as done for instance by Yang and Mazières [24]), so that exhausting the memory allotted to that component doesn't reveal information about the state of the rest of the system and prevents denial-of-service attacks. Another idea is to develop *quantitative* versions of the results discussed here, allowing us to place bounds on the amount of information that can be leaked from a system.

7.4 Dangling Pointers and Freshness

In Section 2, we imposed strong freshness requirements in the allocation rule, mandating not just that the atom assigned to the new object not correspond to any existing object, but that it not be present *anywhere else* in the program state; this assumption is a crucial ingredient for ensuring temporal memory safety and preventing use of a pointer after freeing the corresponding object. A language with garbage collection meets this requirement by guaranteeing that only unreachable objects are freed, but some memory-safe implementations of languages with manual memory management [8, 14] drop it—once again, for performance reasons.

To see how the results of Section 5 would be affected in such a setting, consider the following program

$$x \leftarrow \mathsf{alloc}(1); z \leftarrow (y = x)$$

and suppose we start it executing on a state where y holds a dangling pointer—that is, one whose corresponding block has been freed. Depending on the behavior of the allocator and on the state of the memory, the pointer assigned to x could be equal to y or not. Since this outcome depends on the entire state of the system, not just the reachable memory, Theorems 5.7 to 5.9 do not hold anymore. Furthermore, an attacker with detailed knowledge of the allocator implementation could disclose information about the entire program state by just testing pointers for equality, another potential violation of secrecy.

Weakening freshness guarantees can also have implications for integrity. The problem is that it becomes much harder to guarantee that memory blocks are properly isolated from certain parts of the code: for instance, a newly allocated block might be reachable through a dangling pointer that is controlled by an attacker, allowing them to access that block even if they were not supposed to.

In practice, dangling pointers can have disastrous consequences for overall system security, independently of the freshness issues just described: referencing a pointer to a freed block can violate allocator invariants, enabling many use-after-free attacks that are widely used in practice [22].

8. Related Work

Formal definitions of memory safety typically state the absence of certain dangerous low-level behaviors (such as dangling pointer accesses), in the form of classical progress and preservation theorems or refinements of state transition systems. Being memory safe according to these definitions is valuable in practice because it prevents many attacks that rely crucially on these low-level features, leading to a vast body of work on proving that various techniques for enforcing memory safety are correct according to these definitions; we review a few of them here.

Memory safety is a common concern in low-level systems and languages, leading to a variety of techniques for bringing its benefits to that setting. Approaches include: Cyclone [11], a language featuring a region-based type system for doing safe manual memory management; CCured [16], a program transformation that adds temporal memory safety to C by refining its pointer type to distinguish between various degrees of pointer safety; SoftBound [14], an instrumentation technique for C programs for enforcing spatial memory safety, including the prevention of sub-object bounds violations; CETS [15], a compile-time pass for preventing temporal memory-safety violations in C programs, including accessing dangling pointers into freed heap regions and stale stack frames; and the PUMP [9], a hybrid hardware/software approach for programming various security policies using metadata tags, which can be used to enforce memory safety [4]. These techniques explore different points of the design space, varying significantly in terms of the provided guarantees, applicability and performance impact. All of them include soundness proofs relating them to notions of memory safety as described above.

Mezzo [19] is a concurrent dialect of ML which rules out errors such as data races while enabling certain idioms that are usually not possible in a purely functional setting, such as strong updates (changing the *type* of a reference when assigning to it) and gradual, stateful initialization of immutable data structures. Thus, it provides some guarantees that are beyond what is usually seen as defining memory safety. The soundness proofs for a fragment of the type system [5] include a progress and preservation result, as well as the absence of data races, in the sense that every piece of data can only be written by at most one thread concurrently.

In this paper, we advocate for a different approach, trying to study the formal *consequences* of the enforcement mechanisms for high-level program behavior, rather than the low-level execution invariants they maintain. In other words, we aim for a more *extensional* theory of memory safety, in the sense that it is closely tied to the results programs produce and the effect they have on their environment, contrasting with the previous *intensional* results, which care more deeply about implementation details and *how* programs compute. While we feel that the value of our approach had not been explicitly recognized before in the context of memory safety, some previous work has formally characterized memory safety in terms that are closer in spirit to ours. We mention some of it here.

L^3 [3] is an earlier design of a core calculus featuring a linear type system for strong updates and aliasing control. Like Mezzo, its safety guarantees go beyond "classical" memory safety. Its soundness proof develops a rich theory of logical relations that features reasoning principles similar to the frame results of Theorems 5.7

to 5.9, showing that well-typed programs can always be executed in an extended environment without affecting it. That is, it enforces integrity (Principle 3.1). On the other hand, their interpretation does not make any secrecy results evident.

Yarra [21] is an extension of C that allows programmers to protect data structures by marking their types as *critical*. Reads and writes to critical data must be explicitly annotated on the program. These annotations instruct the runtime system to check that the value stored at the critical location has not been influenced by any operations that were not explicitly marked as critical, halting the program otherwise. This implies that it is possible to guarantee that a piece of untrusted code won't compromise the *integrity* of critical data by checking that that code doesn't include any critical write operations. The authors formalize this reasoning principle as a program logic that validates a frame rule. On the other hand, Yarra's mechanism was not designed to prevent untrusted code from accessing critical data through a non-critical read, which could result in secrecy violations.

Separation logic [20, 25] was introduced as an extension to Hoare logic for verifying programs that manipulate complex, dynamic data structures. An important ingredient for the soundness of separation logic is ensuring that programs do not cause any memory errors when executed; in this sense, we can see separation logic as providing memory safety. Following the discussion in Sections 3 and 5, however, we can see that these programs are memory safe in a much *deeper* sense: the fact that the frame rule is sound for separation logic is what enables local reasoning and makes it interesting; indeed, the frame rule is often portrayed as its most important rule. The crucial difference between the guarantees of separation logic and the ones studied in this paper is that separation logic requires detailed manual proofs for all parts of a program, whereas we offer guarantees that can be derived with less detailed reasoning and partial information.

In recent work [2], Agten *et al.* have shown how to strengthen the guarantees of separation logic in the presence of unverified code. A program transformation inserts checks at points where verified and unverified portions of a program call each other to ensure that the assumptions required by the verified part are met during execution. They prove that the transformation is safe in the sense that no low-level memory-safety violations occur during execution, but do not further analyze the consequences of this fact. In particular, if we wanted to use their results to show that an object is not affected by some component of a program (such as the plugin of Section 6), we would have to completely verify that component. The problem is that their safety theorem does not assert anything about the final state of a program with unverified components. Furthermore, their technique does not try to enforce secrecy, since it allows context code to read private memory that is owned by the verified module.

Benton and Tabereau [6] show how to compile a simple higher-order language while validating a semantic interpretation of types. This interpretation guarantees, among other things, that executing that code in some memory context preserves arbitrary invariants on that context. Furthermore, because the interpretation is *relational*, it can express both secrecy and integrity guarantees.

The more extensional approach has proved fruitful in other areas as well. Morrisett *et al.* [13] propose to formulate the correctness of garbage collection from a semantic standpoint, by requiring that collection algorithms leave the program in a state that is observationally equivalent to the state before collection. Their formalization treats heap locations up to alpha-equivalence, also emphasizing the fact that locations have no structure other than their identities. Abadi and Plotkin [1] study the effectiveness of address randomization by proving a full-abstraction result in the context of a core calculus, guaranteeing that observationally equivalent programs in the calculus are equivalent with high probability when translated to a lower-level calculus. One difference between these approaches and ours is that they guarantee that *any* high-level reasoning principles available in a language are still available after applying a given transformation or technique, whereas we propose to study which reasoning principles memory safety provides exactly.

9. Conclusions and Future Work

We have presented a structured semantics for understanding memory safety and explored some of its consequences, formalizing intuitive reasoning principles that we consider to be the essence of memory-safe systems when compared to unsafe ones.

This approach may generalize to other settings and good programming practices; we mention a few possibilities here. We could try to understand what can be guaranteed in systems that are memory safe in some aspects but not others: a language that permits pointer-to-integer casts could implement a randomized allocator that provides some form of statistical noninterference, for example. Our semantics formalizes the common intuition that pointers behave as capabilities, hinting at how we could better understand capability-based programming. More generally, compartmentalization and the principle of least privilege are seen as important tools for building robust systems, and it would be good to develop a theory for measuring how compartmentalized a given system is, and what concrete consequences a given compartmentalization strategy has on the overall behavior of the program.

The language used in this paper has a simple storage model, with just global variables and heap-allocated objects. Realistic languages, in contrast, have a much richer notion of state—typically including procedures with stack-allocated local variables, as well as objects that can be partitioned into contiguously allocated sub-objects. In terms of memory safety, this means that these languages have a richer vocabulary for describing resources that programs can access, and programmers could benefit from capability-based reasoning in the style of Principles 3.1 and 3.2 to better understand how these resources are manipulated.

For example, in a memory-safe language with procedures and stack variables, it should be possible to assert that the behavior of a procedure depends only on the arguments that it is given, the global variables it uses, and the portions of the state that are reachable from these values; if the caller of that procedure has a private object that is not passed as an argument, it should not affect or be affected by the call. Additionally, languages such as C allow for objects consisting of contiguously allocated sub-objects for improved performance. Some systems [8, 14] add spatial memory safety to C while allowing programmers to *downgrade capabilities*; that is, narrowing the range of a pointer so that it can't be used to access outside of a sub-object's bounds. It would be interesting to refine our model to take into account the reasoning tools enabled by this feature.

The main motivation behind our work was trying to understand formally the benefits of memory safety for informal and partial reasoning; however, our approach hints at ways in which we could improve formal verification using proof assistants as well. One promising idea is to leverage the guarantees of memory safety to obtain formal proofs of program correctness modulo errors in contexts where complete verification is too expensive or not possible (for instance, for programs with a plugin mechanism), perhaps building upon versions of separation logic that can reason about exceptions (e.g., [10]). We would also like to explore the limits and relative advantages of doing program verification purely equationally, without the support of a specialized assertion language, in the style of Section 6.

References

[1] M. Abadi and G. D. Plotkin. On protection by layout randomization. *ACM Trans. Inf. Syst. Secur.*, 15(2):8, 2012. URL http://dblp.uni-trier.de/db/journals/tissec/tissec15.html#AbadiP12.

[2] P. Agten, B. Jacobs, and F. Piessens. Sound modular verification of C code executing in an unverified context. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 581–594, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. . URL http://doi.acm.org/10.1145/2676726.2676972.

[3] A. Ahmed, M. Fluet, and G. Morrisett. L^3: A linear language with locations. *Fundam. Inform.*, 77(4):397–449, 2007. URL http://content.iospress.com/articles/fundamenta-informaticae/fi77-4-06.

[4] A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *36th IEEE Symposium on Security and Privacy (Oakland S&P)*. IEEE, May 2015.

[5] T. Balabonski, F. Pottier, and J. Protzenko. Type soundness and race freedom for Mezzo. In M. Codish and E. Sumii, editors, *Functional and Logic Programming – 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2014. ISBN 978-3-319-07150-3. . URL http://dx.doi.org/10.1007/978-3-319-07151-0_16.

[6] N. Benton and N. Tabareau. Compiling functional types to relational specifications for low level imperative code. In A. Kennedy and A. Ahmed, editors, *TLDI*, pages 3–14. ACM, 2009. ISBN 978-1-60558-420-1. URL http://dblp.uni-trier.de/db/conf/tldi/tldi2009.html#BentonT09.

[7] N. Benton and U. Zarfaty. Formalizing and verifying semantic type soundness of a simple compiler. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '07, pages 1–12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-769-8. . URL http://doi.acm.org/10.1145/1273920.1273922.

[8] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. HardBound: Architectural support for spatial safety of the C programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 103–114, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. . URL http://doi.acm.org/10.1145/1346281.1346295.

[9] U. Dhawan, C. Hriţcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon. Architectural support for software-defined metadata processing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 487–502, 2015. URL http://ic.ese.upenn.edu/abstracts/sdmp_asplos2015.html.

[10] C. Gherghina and C. David. A specification logic for exceptions and beyond. In A. Bouajjani and W. Chin, editors, *Automated Technology for Verification and Analysis – 8th International Symposium, ATVA 2010, Singapore, September 21-24, 2010. Proceedings*, volume 6252 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2010. ISBN 978-3-642-15642-7. . URL http://dx.doi.org/10.1007/978-3-642-15643-4_14.

[11] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. *SIGPLAN Not.*, 37 (5):282–293, May 2002. ISSN 0362-1340. . URL http://doi.acm.org/10.1145/543552.512563.

[12] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). 100:1–77, 1992.

[13] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 66–77, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7. . URL http://doi.acm.org/10.1145/224164.224182.

[14] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. . URL http://doi.acm.org/10.1145/1542476.1542504.

[15] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In J. Vitek and D. Lea, editors, *ISMM*, pages 31–40. ACM, 2010. ISBN 978-1-4503-0054-4. URL http://dblp.uni-trier.de/db/conf/iwmm/ismm2010.html#NagarakatteZMZ10.

[16] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005. ISSN 0164-0925. . URL http://doi.acm.org/10.1145/1065887.1065892.

[17] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:2003, 2002.

[18] A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, New York, NY, USA, 2013. ISBN 1107017785, 9781107017788.

[19] F. Pottier and J. Protzenko. Programming with permissions in Mezzo. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, pages 173–184, Sept. 2013.

[20] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9. URL http://dl.acm.org/citation.cfm?id=645683.664578.

[21] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker, and B. G. Zorn. Modular protections against non-control data attacks. *Journal of Computer Security*, 22(5):699–742, 2014. . URL http://dx.doi.org/10.3233/JCS-140502.

[22] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4. . URL http://dx.doi.org/10.1109/SP.2013.13.

[23] The Coq Development Team. *The Coq Reference Manual, version 8.4*, Aug. 2012. Available electronically at http://coq.inria.fr/doc.

[24] E. Z. Yang and D. Mazières. Dynamic space limits for Haskell. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 588–598, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. . URL http://doi.acm.org/10.1145/2594291.2594341.

[25] H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, FoSSaCS '02, pages 402–416, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43366-X. URL http://dl.acm.org/citation.cfm?id=646794.704850.