LoCalMem: Type-Directed Adaptive Serialization for Location- and Content-Addressable Memory

MICHAEL RAINEY, Carnegie Mellon University, USA
MICHAEL H. BORKOWSKI, Purdue University, USA
MICHAEL VOLLMER, University of Kent, United Kingdom
CHAITANYA S. KOPARKAR, The MathWorks, Inc, USA
MIKAH KAINEN, Purdue University, USA
VIDUSH SINGHAL, Purdue University, USA

Functional programming languages support non-destructive updates via structural sharing, creating a fundamental tradeoff in memory representation: pointer-based heaps preserve sharing but degrade layout locality, while serialized heaps prioritize locality at the cost of duplication. Gibbon addresses this tradeoff by using adaptive serialization: as the program updates its heap, the memory manager improves locality through serialization, falling back to indirection where sharing is unavoidable.

In this work, we formalize key parts of Gibbon's memory model, including the statics and dynamics of adaptive serialization. We then present a unified, type-directed memory model with two realizations: location-addressable memory (LAM) for local execution and content-addressable memory (CAM) for persistence and distributed deduplication. The key abstraction in both models is a notion of explicit boundary datatypes mediating between sharing and serialization. In LAM, boundaries facilitate adaptive serialization and enable our soundness proofs. In CAM, the same boundaries enable an efficient chunking policy, whereby candidate chunk points are selected via rolling-hash cuts. We establish efficiency via a static bound on the number of tokens between consecutive boundaries, bounding reserialization costs.

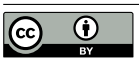
We work in the purely functional setting, where structural sharing is observationally transparent and content-addressed identity is well defined. We formalize both models with operational semantics and prove soundness with respect to ordinary functional values. For CAM, we prove a locality theorem: the work required for an update scales with the depth of the modified path rather than the size of the value. Consequently, large persistent values can be updated incrementally, with unchanged substructures shared automatically across versions and across machines. Together, these results show that a single boundary discipline can unify local structural sharing and distributed deduplication within one type-theoretic foundation.

CCS Concepts: • **Software and its engineering → Semantics**; **Data types and structures**; **Functional languages**.

Additional Key Words and Phrases: packed data representation, serialization, content-addressable memory, garbage collection, region-based memory management, content-defined chunking, operational semantics

ACM Reference Format:
Michael Rainey, Michael H. Borkowski, Michael Vollmer, Chaitanya S. Koparkar, Mikah Kainen, and Vidush Singhal. 2026. LoCalMem: Type-Directed Adaptive Serialization for Location- and Content-Addressable

Authors' Contact Information: Michael Rainey, Carnegie Mellon University, Pittsburgh, PA, USA, mrainey@andrew.cmu.edu; Michael H. Borkowski, Purdue University, West Lafayette, IN, USA, mhborkow@purdue.edu; Michael Vollmer, University of Kent, Canterbury, United Kingdom, m.vollmer@kent.ac.uk; Chaitanya S. Koparkar, The MathWorks, Inc, Natick, MA, USA, ckoparkar@gmail.com; Mikah Kainen, Purdue University, West Lafayette, IN, USA, mtkainen@gmail.com; Vidush Singhal, Purdue University, West Lafayette, IN, USA, singhav@purdue.edu.



ACM 2475-1421/2026/8-ART290
https://doi.org/10.1145/3828688

Memory. *Proc. ACM Program. Lang.* 10, ICFP, Article 290 (August 2026), 34 pages. https://doi.org/10.1145/3828688

1 Introduction

Functional programming languages often represent algebraic data types as pointer-linked heap objects to enable structural sharing: updating a tree rebuilds constructors only on the path from root to edit point, reusing unchanged subtrees. This representation is inherently pointer-based, as values contain references meaningful only within a single runtime's address space. In contrast, a fully serialized layout is compact and streamable, but sacrifices sharing because even small edits force the entire value to be re-encoded. Neither extreme is satisfactory for programs that must both compute efficiently and exchange data across storage, network, or machine boundaries. The cost of bridging the two representations is well documented. Serialization and deserialization can account for a significant fraction of runtime cost [11], and the associated cost has motivated language-level approaches that compute directly on serialized data [5].

The Gibbon project [28, 29] pioneered a middle path: computing directly on mostly serialized representations of algebraic data types. Gibbon compiles a strict, purely functional subset of Haskell to C via whole-program monomorphization. Values live in regions—contiguous, growable buffers—where constructors are packed in depth-first order with no per-node pointers. When sharing or out-of-order allocation demands it, the compiler introduces indirection pointers, tagged references from one region into another, as opt-in exceptions to the otherwise dense layout. Recent work [14] developed adaptive serialization, a runtime optimization that restores dense representations by splicing data back inline when sharing constraints permit. The resulting runtime biases toward cache-friendly, serialized layout while retaining structural sharing and recursive traversal over algebraic datatypes.

Two gaps remain in this line of work. First, although LoCal [28] formalizes Gibbon's core intermediate representation, the runtime mechanisms that manage indirection—wrapping, splicing, shallow duplication, moving garbage collection—have never been formalized. Second, Gibbon's pointers are ephemeral: they name offsets within a single runtime's store and become meaningless once a value must be persisted, transmitted, or compared across machines. Content-addressable storage names values by the cryptographic digest of their content, creating stable, location-independent identifiers. The idea has proven broadly useful, from Git [25] and IPFS [3] to Nix [8], Unison [26], and cryptocurrency [12, 17, 30]. Miller et al. [15] proposed the first integration of content-addressable storage into typed functional programming, but their approach focuses on authenticated datastructures, and their per-node hashing imposes $O(n)$ digest overhead on n-node structures. No prior work bridges Gibbon-style serialized computation with content-addressable persistence.

We address both gaps with a unified, type-directed runtime model whose central principle is the *boundary datatype*. We distinguish boundary datatypes, recursive types such as lists and trees whose constructors mark potential sharing points, from *alias datatypes* such as fixed-width tuples, which always materialize inline. In our *location-addressable memory* (LAM), boundary positions are where wrapper constructors enable structural sharing; the cost of shallow duplication is bounded by $\mathrm{MaxDupSzT}^{\tau}$, a static, type-dependent measure of tokens between consecutive boundaries. In our *content-addressable memory* (CAM), the same boundary positions serve as evaluation points for a content-defined chunking algorithm that determines where to form blocks, with a tunable parameter κ controlling granularity. This directly addresses the $O(n)$ digest overhead of fine-grained hashing [15] by amortizing digests across variable-sized, content-determined blocks.

Values flow between the two models via typed commit (LAM→CAM) and its inverse, dereference. The single measure $\mathrm{MaxDupSzT}^{\cdot}$ governs both duplication cost in LAM and chunk-point spacing in CAM, connecting the two models under one type-directed decomposition principle.

Like Gibbon, we restrict attention to purely functional values: values are immutable and updates are non-destructive. Immutability is what makes structural sharing observationally transparent and gives a content-addressed digest a fixed, well-defined meaning.

Our aim is foundational: we lay the groundwork for mechanized verification of the memory model pioneered by Gibbon. We specify what correct adaptive serialization and content-addressing *should* do, and prove that specification sound with respect to ordinary functional values. The conceptual foundations this requires—the static measure $\mathrm{MaxDupSzT}^{\cdot}$, the boundary/alias distinction, wrapping and splicing as mutual inverses, and shallow duplication—did not exist in precise form in prior work; we establish them and prove them sound.

Contributions.

- **A typed runtime model integrating LAM and CAM** (§3–5): operational semantics for wrapping, splicing, shallow duplication, garbage collection, and content-addressed commit/dereference, together with soundness theorems establishing equivalence with ordinary functional values. Wrapper constructors at boundary positions enable structural sharing with duplication cost bounded by the static measure $\mathrm{MaxDupSzT}^{\tau}$.
- **Content-defined block decomposition on typed structures** (§5). We adapt content-defined chunking [16, 31] from byte streams to typed algebraic data, addressing the $O(n)$ digest overhead of per-node hashing [15] via variable-sized blocks with tunable granularity.
- **A locality theorem for typed chunking** (§5.3, Appendix B.2). A depth-d path edit affects $O(d/\kappa)$ blocks in expectation; spillover into s unchanged siblings is $s \cdot (\kappa - 1)$, independent of subtree depth thanks to a chunk-size cap. Specializing to balanced binary trees of n constructors gives $O(\log^2 n)$ work per edit (Corollary 5.3).

§2 gives an informal overview. §3–5 develop the formal model. §6 discusses related work and §7 concludes.

2 Overview

Our core type system is monomorphic, with primitive values, tuples, and algebraic datatypes, such as the datatype definition of a labeled binary tree

$$\textbf{data}\ \mathrm{itree} \;=\; \mathrm{Leaf}\ () \;\mid\; \mathrm{Branch}\ (\textbf{uint} \boxtimes \mathrm{itree} \times \mathrm{itree}).$$

A Branch node carries a label (an unsigned integer) and two children. The tuple annotations $\times$ and $\boxtimes$ control field access: $\times$ fields store a skip index enabling random access, while $\boxtimes$ fields are accessed sequentially (§3 gives the full treatment). A value of this type is

$$V_0 \;\triangleq\; \mathrm{Branch}\,\big(5, \mathrm{Branch}\ (2,\ \mathrm{Leaf}\ (),\ \mathrm{Leaf}\ ()), \mathrm{Branch}\ (9,\ \mathrm{Leaf}\ (),\ \mathrm{Leaf}\ ())\big).$$

Our runtime represents such values as a contiguous sequence of tokens, laid out in depth-first order:

$$H_0 \;\triangleq\; [\,\mathrm{Branch},\ 8,\ 5, \mathrm{Branch},\ 7,\ 2,\ \mathrm{Leaf},\ \mathrm{Leaf}, \mathrm{Branch},\ 12,\ 9,\ \mathrm{Leaf},\ \mathrm{Leaf}\,].$$

Here, position 0 holds the root constructor tag Branch. Positions 1, 4, and 9 are *skip indices* pointing to the start of right subtrees (to indices 8, 7, and 12 respectively), so that a traversal visiting only the right child can jump directly to it without scanning the left. The remaining positions store labels and constructor tags in left-to-right order.

Crucially, this token sequence is not self-delimiting: a token is only a constructor tag, a skip index, or a primitive literal, with nothing marking which is which or where a constructor's payload ends. Recovering V_0 from H_0 therefore requires the type itree, which determines how each token is read and how many each constructor consumes: the type is part of the runtime representation, not a static overlay checked before execution. This is what keeps the layout dense: because the

type already records each value's shape, the heap stores no sizes or end markers, only payload, and a full traversal is a linear scan over contiguous memory. The price is that fields are reached sequentially—reading the k-th means scanning the $k-1$ before it—which is precisely what the skip indices above buy back, restoring random access where it is worth the space.

2.1 Location-addressable memory with adaptive serialization

Some functional programs construct values in an order that is perfectly aligned with the serialized layout, such as the classic build-tree function.

$$\texttt{buildtree} : \texttt{uint} \to \texttt{itree}$$
$$\texttt{buildtree}\ n = \begin{cases} \text{Leaf}\ () & \text{if } n = 0 \\ \text{Branch}\ \big(n,\ \texttt{buildtree}\ (n-1),\ \texttt{buildtree}\ (n-1)\big) & \text{otherwise.} \end{cases}$$

Because `buildtree` allocates depth-first, the entire value fits contiguously in one region via bump-pointer allocation. In practice, however, functional programs often need to allocate values out of order, so such a compact layout is not always immediately achievable.

Adaptive serialization supports any order of construction for purely functional values, and heals fragmentation as it goes. The driving mechanism is the wrapper constructor, along with a global invariant that restricts the use of pointer types. For a given datatype, we enable sharing of its values via the introduction of the wrapping constructor, a reserved constructor to mark a boundary in the heap. For our binary tree type, we have the constructor $K^{\text{itree}*} : \textbf{laptr}\,\text{itree} \to \text{itree}$, where **laptr** represents the type constructor for location pointers. We restrict pointers to these constructors because pointer types commit, at the type level, to an indirection-based representation, effectively blocking serialization. Wrapping, by contrast, is a value-level choice that can later be undone.

Suppose a program updates the label in our value V_0 from 9 to 10: it must now reference parts of the old value. Our mostly serialized memory supports such incremental updates using a regioned store, where heaps may refer to one another via typed location pointers of the form $\langle r, i\rangle$, denoting an offset i within region r. When updating a value V_0, the runtime allocates a new region r_1 containing only the modified path from the root to the edit point, while the unchanged left subtree remains in the original region r_0:

$$r_0 \mapsto [\,\text{Branch},\ 8,\ 5,\ \underline{\text{Branch}},\ 7,\ 2,\ \text{Leaf},\ \text{Leaf},\ \text{Branch},\ 12,\ 9,\ \text{Leaf},\ \text{Leaf}\,]$$
$$r_1 \mapsto [\,\text{Branch},\ 5,\ 5,\ K^{\text{itree}*},\ \langle r_0, 3\rangle,\ \text{Branch},\ 9,\ 10,\ \text{Leaf},\ \text{Leaf}\,]$$

Region r_1 contains the rebuilt root and right subtree. The wrapper tag $K^{\text{itree}*}$ followed by the location pointer $\langle r_0, 3\rangle$ marks a sharing boundary: any function that accesses the left subtree must follow the pointer into r_0.

Wrapping introduces fragmentation into the heap, which is acceptable for local computation, but undesirable for serialization. Its inverse operation is splicing. If only the updated value V_1 survives garbage collection, the reused subtree of V_0 is spliced into V_1's new destination, producing a fresh region

$$r_2 \mapsto [\,\text{Branch},\ 8,\ 5,\ \text{Branch},\ 7,\ 2,\ \text{Leaf},\ \text{Leaf},\ \text{Branch},\ 12,\ 10,\ \text{Leaf},\ \text{Leaf}\,].$$

Otherwise, if both values survive, then the original structure remains intact. This policy ensures that splicing opportunistically reduces fragmentation.

For types admitting values of unbounded size, such as lists and trees, the availability of a wrappable boundary is essential: without one, splicing or duplicating a value would traverse an unbounded amount of memory, destroying the asymptotic complexity of the original program. Our

design therefore adopts a conservative baseline: *The collector is never forced to splice an object whose size may be unbounded. The mutator uses duplication as a fallback when sharing is not possible.*

Enabling sharing for all datatypes would enforce this policy but waste space on types of statically known size. A simple alias such as

$$\textbf{data}\ \text{height}\ =\ \text{Height}\ \textbf{uint}$$

would require two constructor tags (wrapper plus payload) when zero would suffice. We therefore distinguish boundary datatypes, which admit wrapping, from *alias datatypes*, which always materialize inline with their constructor tags elided (resembling Haskell's *newtype*, but additionally requiring a bounded representation).

A side effect of this design is that value duplication is no longer trivial. When the runtime creates the shared representation of V_1, it must copy enough of the old structure to set up the wrapper. The amount copied is bounded by MaxDupSzT^{τ}, a static measure determined solely by the type: it counts the maximum number of tokens between consecutive boundary datatype positions. For itree, $\text{MaxDupSzT}^{\text{itree}} = 2$ because it is a boundary type. No matter how large the tree, creating a shared copy never traverses more than two tokens per wrapper boundary. But for the tuple type ($\textbf{uint} \boxtimes \text{itree} \times \text{itree}$) it is $\underbrace{1}_{\text{skip}} + \underbrace{1}_{\text{label}} + \underbrace{2}_{\text{itree}} + \underbrace{2}_{\text{itree}} = 6$.

2.2 Content-addressable memory with controlled chunking

To persist, transmit, or share data across machines, the runtime must produce a stable, portable representation. This is the role of CAM. To commit a value to CAM, the runtime:

(1) traverses the value, splicing away any wrapper constructors and location pointers to produce a self-contained heap;
(2) canonically encodes the heap into a byte sequence;
(3) stores the byte sequence in the external CAM, receiving a cryptographic digest as the value's identifier.

The resulting digest is a content pointer: a stable name for the value that can be stored, communicated, or used on a different machine. Given a content pointer, any machine connected to the same CAM can retrieve the byte sequence, decode it, and reconstruct the value in local memory.

Committing an entire value as a single byte sequence would sacrifice sharing: even a small edit forces the whole value to be re-encoded and re-stored. At the other extreme, hashing every individual node maximizes sharing but incurs $O(n)$ digest overhead—the problem identified by Miller et al. [15]. To make the tradeoff concrete, consider three CAM-aware variants of our itree datatype:

Branch ($\textbf{uint} \boxtimes \textbf{captr}\ \text{itree} \times \textbf{captr}\ \text{itree}$)	(a) per-child digest
Branch ($\textbf{uint} \boxtimes \textbf{captr}\ (\text{itree} \times \text{itree})$)	(b) per-pair digest
Branch (...) | $\text{Branch}_{\text{ca}}$ ($\textbf{uint} \boxtimes \textbf{captr}\ (\text{itree} \times \text{itree})$)	(c) manual chunking

The placement of content pointers fixes the *granularity* of content addressing, trading sharing against per-digest overhead. A content pointer carries a cryptographic digest whose size is set by collision resistance, not chosen freely: 32 bytes (256 bits) in stores built for adversarial sharing, such as IPFS or Git's SHA-256 migration. Variant (a) pays this 32-byte cost at *every* node, dwarfing the handful of payload bytes a small constructor holds. Variant (b) halves the digest count but still spends one per branch, so the overhead stays proportional to the number of nodes. Variant (c) amortizes a single digest across a chunk of many nodes, which is what makes the overhead negligible—at the chunk sizes deployed stores use (8 KB in LBFS [16], 256 KiB in IPFS [3]), a 32-byte digest is well

under 1% of the chunk it names—but it bakes the chunk boundaries into the datatype, demanding manual discipline for every type.

Our runtime uses content-defined chunking. During the commit traversal, the runtime maintains a rolling hash that is updated at each boundary datatype constructor—precisely the same positions where wrapper constructors appear in LAM. Whenever the rolling hash satisfies a boundary condition ($h \bmod \kappa = 0$ for a tunable parameter κ), the runtime cuts the current chunk, stores it under its digest, and replaces it with a wrapper constructor $K^{\text{itree}\bullet}$ followed by a content pointer—the digest naming the chunk just stored. This is the CAM analogue of the $K^{\text{itree}*}$ wrapper in LAM: where LAM inserts a wrapper with a location pointer to share structure across regions, CAM inserts a wrapper with a content pointer to share structure across chunks. The result is a collection of variably sized chunks whose boundaries depend on the content of the data, not on fixed offsets.

Because chunk boundaries are determined by content, identical substructures produce identical chunks. When V_0 and V_1 are both committed to CAM, the chunks corresponding to their shared left subtree will have the same content and the same digest, and need be stored only once. Only the chunks containing the modified path (the root and right subtree) differ between the two versions.

We prove that a path edit of depth d (modifying d constructors from root to edit point) affects at most $O(d/\kappa)$ chunks in expectation, with spillover into each unchanged sibling subtree bounded by $\kappa - 1$ in expectation thanks to a chunk-size cap. The parameter κ controls the trade-off: larger κ means fewer, larger chunks (less metadata) but more content per affected chunk. For balanced binary trees of n constructors, choosing $\kappa = \lfloor \log_2 n \rfloor$ gives $O(\log^2 n)$ work per edit (Corollary 5.3).

3 Typed data in structural and linearized formats

In this section, we lay the groundwork of our LAM and CAM extensions, starting from a simple typed model of heaps. Throughout, values are immutable: the model has no mutable cells, so every update produces a new value rather than modifying one in place. Figure 1 presents the syntax of our language for modeling typed data. The type system (Figure 1(a)) is monomorphic, with types for primitive values (e.g., unsigned integers), tuples, and data constructors arising from algebraic data types. We distinguish two representations of values.

- *Structural values* (Figure 1(b)) are tree-shaped representations of values V. They are built from primitives, tuples, and applications of data constructors. Structural values serve as the high-level, type-directed reference representation against which we relate less structured forms, e.g., linearized and serialized.
- *Linearized values* (Figure 1(c)) are token-based representations given by a heap H. A heap H is a finite sequence of tokens. Each token T carries either a primitive literal, a heap index, or a constructor tag used in a data-constructor application.

Primitives. Primitives are atomic values with a fixed, statically known bit width. To keep our formalism simple, we use a single primitive **uint**, the type of unsigned integers. The development extends straightforwardly to additional primitives (e.g., booleans, bytes, floating point).

Tuples. We admit tuples of any arity, including 0 and 1. The empty tuple has type $()$ (akin to **unit** in Standard ML) and, in both its linearized and serialized forms, occupies no space. Singleton tuples are permitted for uniformity; they add no expressive power and do not complicate the metatheory.

For arity $n \geq 2$ we write an n-ary tuple type as $(\tau_1 \otimes_1 \tau_2 \otimes_2 \cdots \otimes_{n-1} \tau_n)$. The parentheses are the tuple constructor: they form a single n-field tuple. The symbol $\otimes$ acts as a separator that also selects the layout policy for its enclosing tuple. In particular, the chained occurrences of $\otimes$ do *not* denote a left-associating binary product. When context makes it harmless, we may drop the outer parentheses as a typographic shorthand, but the intended constructor is the parenthesised form.

(a) Type System	Data-type identifiers	$\ni$	$\widetilde{\tau}$	
	Constructor tags	$\ni$	K	
	Types	$\ni$	τ, σ	$::= \mathbf{uint} \mid (\tau_1 \otimes_1 \cdots \otimes_{n-1} \tau_n) \mid \widetilde{\tau}$
	Tuple-type constructors	$\ni$	$\otimes$	$::= \times \mid \boxtimes$
	Data-type declarations	$\ni$	τd	$::= \mathbf{data}\ \widetilde{\tau} = K_1\ \tau_1 \mid \ldots \mid K_n\ \tau_n$
(b) Structural	Primitive values	$\ni$	u	
	Tree-shaped values	$\ni$	$\underline{V}$	$::= u \mid (\underline{V}_1, \ldots, \underline{V}_n) \mid K\ \underline{V}$
(c) Linearized	Heap indices	$\ni$	i, j	
	Heap tokens	$\ni$	T	$::= u \mid i \mid K$
	Heaps	$\ni$	H	$::= [T_1, \ldots, T_n]$

Fig. 1. Syntax for types and values.

We admit two choices for $\otimes$ (i.e., the layout policy):

- the *random-access* layout, denoted $\times$, which favors fast access to successive fields (e.g., by storing explicit field boundaries) at the cost of extra header space; and
- the *sequential* layout, denoted $\boxtimes$, which favors compactness (e.g., a minimal header with fields laid out for sequential scan) at the cost of increased traversal time to reach later fields.

Both variants are observationally equivalent at the structural level; they differ only in their linearized/serialized layout and thus in the time–space trade-off. Moreover, both tuple layouts store fields back-to-back. The random-access layout ($\times$) additionally carries a header array H of offsets, written before the fields, giving direct entry to selected fields; the sequential layout ($\boxtimes$) omits this header and relies on sequential scan.

For a tuple $V = (V_1, \ldots, V_n)$ of type $(\tau_1 \otimes_1 \cdots \otimes_{n-1} \tau_n)$ we designate a set $B \subseteq \{2, \ldots, n\}$ of *random-access fields* defined by $B \triangleq \{\, j \mid 2 \le j \le n \wedge \otimes_{j-1} = \times \,\}$: field j is random-access when the separator $\otimes_{j-1}$ preceding it is $\times$. The header H has length $|B|$ and stores, in increasing order, the starting token index of each V_j with $j \in B$. After H the linearization places $V_1, \ldots, V_n$ consecutively. (For the fully sequential layout $B = \varnothing$ and H is empty.)

Algebraic data types. A *data-type declaration* has the form $\mathbf{data}\ \widetilde{\tau} = K_1\ \tau_1 \mid \cdots \mid K_n\ \tau_n$,which introduces a data-type identifier $\widetilde{\tau}$ together with constructors K_i carrying payloads of type τ_i ($1 \le i \le n$). Data-type identifiers are nominal: a type is identified by its declared name, not by its structure. A nullary constructor is written with payload $()$. For later reference we use the word *object* to mean a value formed by a constructor application $K\,V$. Primitives and tuples have sizes that are determined solely by their types, whereas objects are the unique source of dynamic size variability in our model: the size of $K\,V$ depends on the payload V (and may be unbounded in the presence of recursion). We assume that the collection of data-type declarations is well-formed (Definition A.1).

Constructor metadata. We use two metafunctions to talk about the shape of a datatype declaration. For a declared datatype $\widetilde{\tau}$, the function $\mathrm{Ctr}(\widetilde{\tau})$ returns the finite list of *user constructors* together with their payload types, e.g. $\mathrm{Ctr}(\widetilde{\tau}) = [\, K_1\ \sigma_1, \ldots, K_m\ \sigma_m \,]$. By convention, $\mathrm{Ctr}()$ reflects only the constructors that appear in source programs, not reserved constructors that we introduce later. We define $\mathrm{Single}(\widetilde{\tau})$ to be $|\mathrm{Ctr}(\widetilde{\tau})| = 1$.

Maximum token bound. Given a type τ and a set R of data-type identifiers already visited on the descent, the function $\mathrm{MaxSzT}^{\tau}(R) \in \mathbb{N} \cup \{\infty\}$ (Figure 2) returns an upper bound on the number of heap tokens needed to represent any value of type τ. Primitives contribute one token. An empty

$$\mathrm{MaxSzT}^{\mathbf{uint}}(R) = 1,$$

$$\mathrm{MaxSzT}^{(\tau_1 \otimes_1 \cdots \otimes_{n-1} \tau_n)}(R) = |\{\, k \in \{1, \ldots, n-1\} \mid \otimes_k = \times \,\}| \;+\; \sum_{i=1}^{n} \mathrm{MaxSzT}^{\tau_i}(R),$$

$$\mathrm{MaxSzT}^{\widetilde{\tau}}(R) = \begin{cases} \infty, & \text{if } \widetilde{\tau} \in R, \\ \mathrm{MaxSzT}^{\sigma}\big(R \cup \{\widetilde{\tau}\}\big), & \text{if } \widetilde{\tau} \notin R \text{ and } \mathrm{Ctr}(\widetilde{\tau}) = [\, K\, \sigma \,], \\ 1 + \max\limits_{K\, \sigma \in \mathrm{Ctr}(\widetilde{\tau})} \mathrm{MaxSzT}^{\sigma}\big(R \cup \{\widetilde{\tau}\}\big), & \text{otherwise.} \end{cases}$$

Fig. 2. Maximum size in tokens of any value of a given type.

tuple contributes 0 tokens. For a nonempty tuple $(\tau_1 \otimes_1 \cdots \otimes_{n-1} \tau_n)$ the bound is the sum of the bounds of the fields plus a header cost equal to the number of random-access positions $\otimes_k = \times$.

For a datatype $\widetilde{\tau}$ we first guard against cycles: if $\widetilde{\tau} \in R$ the result is ∞. Otherwise we distinguish by the number of user constructors in the declaration. If $\mathrm{Ctr}(\widetilde{\tau}) = [\, K\, \sigma \,]$ (a single constructor), no explicit tag is needed and we account only for the payload, recursing as $\mathrm{MaxSzT}^{\sigma}\big(R \cup \{\widetilde{\tau}\}\big)$. If there are two or more constructors, we count one token for the constructor tag and take the maximum over payloads:

$$1 + \max_{K\, \sigma \in \mathrm{Ctr}(\widetilde{\tau})} \mathrm{MaxSzT}^{\sigma}\big(R \cup \{\widetilde{\tau}\}\big).$$

Thus $\mathrm{MaxSzT}^{\tau}(\emptyset)$ is finite exactly when the type graph reachable from τ is acyclic; when finite, it provides a static budget for heap tokens used in linearization.

Example 3.1. *For the datatype* **data** bool = True () | False ()*,*

$$\mathrm{MaxSzT}^{\mathrm{bool}}(\emptyset) = 1 + \max\big(\mathrm{MaxSzT}^{()}(\{\mathrm{bool}\}),\ \mathrm{MaxSzT}^{()}(\{\mathrm{bool}\})\big) = 1 + \max(0, 0) = 1.$$

Example 3.2. *For the datatype* **data** ilist = Nil () | Cons (**uint** ⊠ ilist)*,*

$$\begin{aligned} \mathrm{MaxSzT}^{\mathrm{ilist}}(\emptyset) &= 1 + \max\Big(\mathrm{MaxSzT}^{()}(\{\mathrm{ilist}\}),\ \mathrm{MaxSzT}^{(\mathbf{uint} \boxtimes \mathrm{ilist})}(\{\mathrm{ilist}\})\Big) \\ \mathrm{MaxSzT}^{(\mathbf{uint} \boxtimes \mathrm{ilist})}(\{\mathrm{ilist}\}) &= 0 + \mathrm{MaxSzT}^{\mathbf{uint}}(\{\mathrm{ilist}\}) + \mathrm{MaxSzT}^{\mathrm{ilist}}(\{\mathrm{ilist}\}) \\ &= 0 + 1 + \infty \quad (\textit{since } \mathrm{ilist} \in R) \;=\; \infty. \end{aligned}$$

Alias types. The metafunction $\mathrm{Alias}(\widetilde{\tau}) : \mathbf{bool}$ decides whether $\widetilde{\tau}$ can be represented without an explicit constructor tag at run time (e.g. *newtype*-style records):

$$\mathrm{Alias}(\widetilde{\tau}) \;\stackrel{\mathrm{def}}{=}\; \mathrm{Single}(\widetilde{\tau}) \;\wedge\; \mathrm{MaxSzT}^{\widetilde{\tau}}(\emptyset) \;<\; \infty$$

The first conjunct requires exactly one user constructor (otherwise a tag is needed to distinguish constructors); the second requires a finite size bound.

Boundary types. A datatype is a *boundary* type when it is not an alias. We write $\mathrm{Bdy}(\widetilde{\tau}) \stackrel{\mathrm{def}}{=} \neg\, \mathrm{Alias}(\widetilde{\tau})$ and use it as the antecedent of every rule whose alias counterpart fires on $\mathrm{Alias}(\widetilde{\tau})$.

3.1 Treeification and linearization

We connect linearized heap representations with inductive (tree-shaped) values using a big-step *treeification* relation. Intuitively, treeification interprets a contiguous region of a heap as a value of a given type, reconstructing its inductive structure while tracking how much of the heap was consumed. Formally, the judgment

$$i; H \;\Downarrow_{\mathrm{T}}^{\tau}\; V; i'$$

$$\frac{H[i] = u}{i; H \Downarrow_{\mathrm{T}}^{\mathbf{uint}} u; i{+}1}\ \text{T-Uint} \qquad \frac{}{i; H \Downarrow_{\mathrm{T}}^{()} (); i}\ \text{T-Emp} \qquad \frac{i_1 = i + |B| \quad \forall_{j\in\{1,\ldots,n\}}.\ i_j; H \Downarrow_{\mathrm{T}}^{\tau_j} V_j; i_{j+1}}{i; H \Downarrow_{\mathrm{T}}^{(\tau_1 \otimes_1 \cdots \otimes_{n-1} \tau_n)} (V_1, \ldots, V_n); i_{n+1}}\ \text{T-Tup}$$

$$\frac{\mathrm{Alias}(\widetilde{\tau}) \quad \mathrm{Ctr}(\widetilde{\tau}) = [K\,\sigma] \quad i; H \Downarrow_{\mathrm{T}}^{\sigma} V; i'}{i; H \Downarrow_{\mathrm{T}}^{\widetilde{\tau}} K\,V; i'}\ \text{T-Alias} \qquad \frac{\mathrm{Bdy}(\widetilde{\tau}) \quad H[i] = K \quad (K\,\sigma) \in \mathrm{Ctr}(\widetilde{\tau}) \quad i{+}1; H \Downarrow_{\mathrm{T}}^{\sigma} V; i'}{i; H \Downarrow_{\mathrm{T}}^{\widetilde{\tau}} K\,V; i'}\ \text{T-Bdy}$$

Fig. 3. Treeification relation $i; H \Downarrow_{\mathrm{T}}^{\tau} V; i'$. Here B enumerates random-access fields.

states that, starting at heap index i, the heap H contains a well-formed value of type τ whose inductive representation is V, and that the representation occupies exactly the heap interval $[i, i')$. The index i' therefore serves as an *end witness* identifying the first position following the value.

Definition 3.3 (End witness). *The (partial) function*

$$\mathrm{EndWitness}^{\tau} : \mathbb{N} \times \mathrm{Heap} \rightharpoonup \mathbb{N}$$

is defined by $\mathrm{EndWitness}^{\tau}(i, H) = i'$ *iff* $\exists V.\ i; H \Downarrow_{\mathrm{T}}^{\tau} V; i'$.

End witnesses play a central role in both treeification and linearization: they allow the semantics to determine field boundaries without requiring prior knowledge of field sizes, even in the presence of nested tuples and recursive datatypes.

The treeification system is given in Figure 3. There is one rule for primitive values, two rules for tuples (empty and nonempty), one rule for ordinary constructor-tagged values, and one specialized rule for single-constructor (alias) datatypes.

Treeification of a nonempty tuple proceeds by skipping the header and then reconstructing each field in sequence. The start index of the first field is $i_1 = i{+}|B|$, where $B = \{\, j \mid 2 \le j \le n \wedge \otimes_{j-1} = \times \,\}$ enumerates the random-access fields, and subsequent field boundaries are computed by chaining end witnesses: $i_{j+1} = \mathrm{EndWitness}^{\tau_j}(i_j, H)$. The final end witness returned by the rule is $i' = i_{n+1}$.

Constructor treeification reads a constructor tag K from the heap, selects the corresponding payload type σ, recursively treeifies the payload, and returns the resulting end witness. The alias rule applies when the datatype has exactly one constructor and is acyclic. In this case, the constructor tag is implicit and omitted from the heap representation; treeification proceeds directly to the payload using the unique constructor. If the datatype participates in a cycle, the ordinary tagged rule is used instead to preserve soundness in the presence of wrapping and splicing.

The *linearization* relation is the inverse direction:

$$V; i; H \Downarrow_{\mathrm{L}}^{\tau} H'; i'.$$

It takes an inductive value V of type τ, an initial heap H with $i = |H|$, and produces an extended heap H' together with an end witness i'. The rules for linearization are given in Figure 4 and closely mirror those for treeification.

For tuples, linearization writes a header followed by the payload fields laid out back-to-back. Using the definition of B above, we first compute the chain of field boundaries $i_1 = i + |B|$, $i_{j+1} = \mathrm{EndWitness}^{\tau_j}(i_j, H)$, and write the header $H_{\mathrm{hdr}} = [\, i_j \mid j \in B \text{ in ascending order} \,]$. The payload is then generated by a sequence of recursive linearization calls, one per field, producing heaps $H_1, H_2, \ldots, H_{n+1}$ with final end witness i_{n+1}. We write $H_1 +\!\!+ H_2$ for heap concatenation.

For constructor values, the alias and boundary cases again diverge. Alias types omit the constructor tag and linearize only the payload. Boundary types first insert the constructor tag at position i and then linearize the payload starting at $i{+}1$.

$$\frac{}{u; i; H \Downarrow_{\mathrm{L}}^{\mathbf{uint}} H +\!\!+ [u]; i{+}1}\ \text{L-Uint} \qquad \frac{}{(); i; H \Downarrow_{\mathrm{L}}^{()} H; i}\ \text{L-Emp} \qquad \frac{\begin{array}{c} H_1 = H +\!\!+ [\, i_j \mid j \in B\,] \quad i_1 = i + |B| \\ \forall_{j \in \{1,\ldots,n\}}.\ V_j; i_j; H_j \Downarrow_{\mathrm{L}}^{\tau_j} H_{j+1}; i_{j+1} \end{array}}{(V_1, \ldots, V_n); i; H \Downarrow_{\mathrm{L}}^{(\tau_1 \otimes_1 \cdots \otimes_{n-1} \tau_n)} H_{n+1}; i_{n+1}}\ \text{L-Tup}$$

$$\frac{\mathrm{Alias}(\widetilde{\tau}) \quad (K\,\sigma) \in \mathrm{Ctr}(\widetilde{\tau}) \quad V;\ i; H \Downarrow_{\mathrm{L}}^{\sigma} H'; i'}{K\,V;\ i; H \Downarrow_{\mathrm{L}}^{\widetilde{\tau}} H'; i'}\ \text{L-Alias} \qquad \frac{\begin{array}{c} \mathrm{Bdy}(\widetilde{\tau}) \quad (K\,\sigma) \in \mathrm{Ctr}(\widetilde{\tau}) \\ H_{\mathrm{tag}} = H +\!\!+ [K] \quad V;\ i{+}1; H_{\mathrm{tag}} \Downarrow_{\mathrm{L}}^{\sigma} H''; i' \end{array}}{K\,V;\ i; H \Downarrow_{\mathrm{L}}^{\widetilde{\tau}} H''; i'}\ \text{L-Bdy}$$

Fig. 4. Linearization relation $V; i; H \Downarrow_{\mathrm{L}}^{\tau} H'; i'$. The header $[\, i_j \mid j \in B\,]$ stores the start index i_j of each random-access field $j \in B$ in ascending order; both layouts place fields back-to-back.

3.2 Correctness: round trips between trees and heaps

Correctness: round trips between trees and heaps. The relations $\Downarrow_{\mathrm{T}}^{\tau}$ and $\Downarrow_{\mathrm{L}}^{\tau}$ are intended to be mutual inverses on the segment they describe. There are two directions to correctness:

- **(Soundness / re-read)** If we linearize a tree value into a heap segment, then treeifying that segment recovers the same tree value.
- **(Completeness / re-emit)** If a heap segment treeifies to a tree value, then linearizing that tree value re-emits a unique heap segment. This segment agrees with the original segment everywhere except possibly on tuple headers (because treeification does not check tuple headers for correctness).

3.2.1 *Segment soundness.*

Lemma 3.4 (Segment soundness (linearize then treeify)). *If $|H| = i$ and $V; i; H \Downarrow_{\mathrm{L}}^{\tau} H'; i'$, then $i; H' \Downarrow_{\mathrm{T}}^{\tau} V; i'$.*

To prove segment soundness, we first observe that linearization only appends tokens at indices $\geq i$ and never modifies positions below i. Consequently, $H'[0..i) = H[0..i)$ and $|H'| = i'$. The proof then proceeds by induction on the derivation of $V; i; H \Downarrow_{\mathrm{L}}^{\tau} H'; i'$.

3.2.2 *Segment completeness.*

Linearizing after treeifying a segment may not produce the exact same segment on a token-for-token basis; treeifying ignores the header tokens for a tuple, but linearization writes out the correct header tokens. Thus a segment with an incorrect header will be well-typed in our system (as defined by treeification), while the output of treeifying and then linearizing will be uniquely determined and can be taken as canonical, as the next lemma shows.

Lemma 3.5 (Segment completeness (treeify then linearize)). *Suppose $i; H \Downarrow_{\mathrm{T}}^{\tau} V; i'$. Then for* any *heap H_0 such that $|H_0| = i$, there exists a derivation*

$$V; i; H_0 \Downarrow_{\mathrm{L}}^{\tau} H'; i'$$

whose output end witness is exactly i'.

Moreover, the segment written by treeification is unique: given any other derivation $V; i; H_0'' \Downarrow_{\mathrm{T}}^{\tau} H''; i''$, then $i' = i''$ and $H'[i, i') = H''[i, i')$.

The proof depends on an invariance property of linearization: none of the rules in Figure 4 inspect the contents of the input heap below the current write index. They only extend the heap at indices $\geq i$ via concatenation. Consequently, if $|H_0| = |H_0''| = i$, then any linearization derivation starting from H_0 can be replayed starting from H_0'', producing the same segment from index i onward and the same end witness i'.

3.2.3 *Maximum token-size bound.*

Lemma 3.6 (MaxSzT bounds the token growth of linearization). *Fix a type τ and let $M \triangleq \mathrm{MaxSzT}^{\tau}(\emptyset)$. Assume $|H| = i$ and*

$$V; i; H \Downarrow_{\mathrm{L}}^{\tau} H'; i'.$$

Then:

(i) the number of tokens emitted by linearization is bounded by M:

$$i' - i \;\le\; M,$$

interpreting the inequality as trivially true when $M = \infty$; and

(ii) in particular, if $M < \infty$ then linearization grows the heap by at most M tokens, i.e. $|H'| - |H| \le M$ (since $|H| = i$ and the rules ensure the final witness satisfies $i' = |H'|$).

Our proof relies on a *monotonicity* property of $\mathrm{MaxSzT}^{\cdot}$: for any type σ and sets $R \subseteq R'$,

$$\mathrm{MaxSzT}^{\sigma}(R) \;\le\; \mathrm{MaxSzT}^{\sigma}(R').$$

Adding types to the visited set can only cause us to encounter a previously visited type sooner, returning ∞ instead of continuing the recursion.

4 Closed-World Store: Opaque, Location-Addressed

The linear heap model developed so far represents values as contiguous segments, written once and never aliased. As such, the model cannot express *structural sharing* because every value is fully materialized at the point it is constructed. Even purely functional updates, such as extending a list, modifying a subtree, or rebuilding a record with one changed field, must re-encode the entire value from scratch. Large substructures that are logically shared across versions are physically duplicated in the heap. To support structural sharing, we generalize heaps into a *closed-world store* that supports explicit, typed aliasing while retaining a simple, append-only operational model.

4.1 A regioned store with location-based addressing

We replace a single global heap with a *regioned store* (Figure 5). A store S maps a finite set of *region identifiers* r to heaps H_r. Each region is written independently and grows monotonically by appending new tokens. Indices within a region are therefore stable once allocated. The key extension is the introduction of *location-based addresses* $\langle r, i \rangle$, which denote a reference to index i within region r. These addresses are first-class tokens: they may appear inside values, be stored in heaps, and be manipulated by the semantics. To track such references precisely, we extend the type system with a pointer type **laptr** τ, read as "a location whose target treeifies as a τ". At the level of tree-shaped values, **laptr** τ values are simply addresses. Deep equality, when needed, is obtained by dereferencing the address and treeifying the target at type τ. Treeification and linearization of location-based addresses are intentionally trivial: each reads or writes exactly one token (Figure 5(d)). This ensures that introducing aliasing does not complicate the core tree/heap correspondence. The static size analysis reflects this choice: a location-based address always occupies exactly one token, independent of the size of the value it refers to.

Figure 6 defines the primitive operations for extending a store. The operation $\mathrm{Ins}(S, r, i, T)$ writes a single token T at index i in region r. If the region does not yet exist, it is created implicitly. Otherwise, the token is added to the existing heap for r. The batched variant $\overline{\mathrm{Ins}}$ writes a contiguous sequence of tokens starting at index i.

$$
\begin{array}{llcll}
\text{(a) Regions} & \text{Region identifiers} & \ni & r & \\
 & \text{Location-based addresses} & \ni & \langle r, i\rangle & \\
\text{(b) Type System} & \text{Types} & \ni & \tau & ::= \ \ldots \mid \mathbf{laptr}\ \tau \\
\text{(c) Structural} & \text{Tree-shaped values} & \ni & V & ::= \ \ldots \mid \langle r, i\rangle \\
\text{(c) Linearized} & \text{Tokens} & \ni & T & ::= \ \ldots \mid \langle r, i\rangle \\
 & \text{Mostly linearized stores} & \ni & S & ::= \{r_1 \mapsto H_1, \ldots, r_n \mapsto H_n\}
\end{array}
$$

$$
\mathrm{MaxSzT}^{\mathbf{laptr}\ \tau}(R) = 1 \quad \frac{H[i] = \langle r, i'\rangle}{i; H \Downarrow_{\mathrm{T}}^{\mathbf{laptr}\ \tau} \langle r, i'\rangle; i+1}\ \textsc{T-Laptr} \qquad \frac{}{\langle r, i'\rangle; i; H \Downarrow_{\mathrm{L}}^{\mathbf{laptr}\ \tau} H +\!\!+ [\langle r, i'\rangle]; i+1}\ \textsc{L-Laptr}
$$

Fig. 5. Syntax extensions for regions and location-based addressing.

$$
\mathrm{Ins}(S, r, i, T) = \begin{cases} S \cup \{\, r \mapsto \{i \mapsto T\}\,\} & \text{if } r \notin \mathrm{dom}(S), \\ S \cup \{\, r \mapsto (H \cup \{i \mapsto T\})\,\} & \text{if } S[r] = H \text{ and } i \notin \mathrm{dom}(H). \end{cases}
$$

$$
\overline{\mathrm{Ins}}(S, r, i, [T_1, \ldots, T_n]) = \begin{cases} S \cup \{\, r \mapsto A\,\} & \text{if } r \notin \mathrm{dom}(S), \\ S \cup \{\, r \mapsto (H \cup A)\,\} & \text{if } S[r] = H, \\ & \text{and } J \cap \mathrm{dom}(H) = \emptyset. \end{cases}
$$

$$
\text{where } A = \{\, i + k - 1 \mapsto T_k \mid k = 1, \ldots, n\,\}, \quad J = \{\, i, \ldots, i + n - 1\,\}.
$$

Fig. 6. Store-update operations.

4.2 Adaptive linearization: wrapping and splicing

The regioned store and location-based addresses give us *structural sharing*: a value may reuse an existing substructure by storing an address $\langle r, i\rangle$ instead of copying its tokens. This expressive power is necessary, but it immediately introduces a representation choice that was absent in the fully linearized heap.

At every occurrence of a compound subvalue, the runtime must decide whether to:

- *inline* the subvalue by linearizing its tokens contiguously with the surrounding context, or
- *refer* to an already-allocated representation via an address.

Inlining maximizes contiguity and is ideal for serialization and content-addressable storage. Referring maximizes sharing and is essential for efficient functional updates. These goals are in tension, and neither choice can be fixed once and for all.

The ideal case is a functional program that constructs values in the same order as the linearized representation, such as the recursive `buildtree` function defined in § 2.1. Many programs do not enjoy this property. In particular, functions that *rearrange* existing data—rather than building it bottom-up—tend to allocate values out of order.

A simple example is list reversal: `reverse` : List $\alpha \longrightarrow$ List α. A typical functional implementation consumes the input list from left to right while producing output nodes that conceptually belong at the end of the result. The logical structure of the result is therefore determined only after intermediate nodes have already been allocated.

More generally, out-of-order allocation arises whenever a value is:

- rebuilt by traversing an existing structure in a non-structural order (e.g., reverse, zip, flatten),
- assembled incrementally from multiple sources (e.g., concatenation, merging),
- or partially reused across versions (persistent updates).

In these cases, insisting on contiguous linearization would require either backpatching previously written heap segments or copying large substructures wholesale—both incompatible with our append-only store model.

Wrapping as a deferral mechanism. To accommodate such programs, we make the representation choice explicit and revisable by introducing *wrapping constructors*. A wrapping constructor marks a point where the mutator chooses sharing over contiguity, deferring the final layout decision.

Concretely, for each boundary datatype $\widetilde{\tau}$ we reserve a constructor $K^{\widetilde{\tau}*} : \mathbf{laptr}\ \widetilde{\tau} \to \widetilde{\tau}$. A value of the form $K^{\widetilde{\tau}*}\ \langle r, i\rangle$ is a *wrapped object*. It inhabits the same source type as an ordinary constructor application, but its payload is stored elsewhere in the store.

Wrapping differs fundamentally from the pointer type $\mathbf{laptr}\ \tau$. The pointer type commits, at the type level, to a representation as a single address token. Wrapping, by contrast, is a value-level choice: it preserves the surrounding datatype, recording only that the current occurrence is indirect.

Splicing and re-linearization. Wrapping introduces fragmentation into the heap, which is acceptable for local computation but undesirable for serialization and global storage. The inverse operation, *splicing*, is therefore performed by the garbage collector.

When the collector encounters a wrapped object $K^{\widetilde{\tau}*}\ \langle r, i\rangle$, it may replace the wrapper by the token segment located at $\langle r, i\rangle$, thereby restoring contiguity. To preserve sharing, the collector maintains a forwarding map F of already-visited addresses:

- if $\langle r, i\rangle \in \mathrm{dom}(F)$, the wrapper is preserved, ensuring that shared substructures remain shared;
- otherwise, the wrapper is spliced, and the target segment is copied inline.

This policy ensures that splicing is opportunistic but safe: it improves layout locality without duplicating shared structure.

Example 4.1 (Wrapping and splicing an `ilist` tail). *We use the int-list* ilist *declaration from Example 3.2. Consider a store* $S = \{r_{\mathsf{tail}} \mapsto H_{\mathsf{tail}}, r_{\mathsf{front}} \mapsto H_{\mathsf{front}}\}$, *where* $H_{\mathsf{tail}} = [\,\mathrm{Cons},\ 321,\ \mathrm{Nil}\,]$. *The front heap stores the head* $\mathrm{Cons}(123,\ \cdot)$ *and wraps a local pointer into the tail heap:*

$$H_{\mathsf{front}} = [\,\mathrm{Cons},\ 123,\ K^{\mathrm{ilist}*},\ \langle r_{\mathsf{tail}}, 0\rangle\,],$$

where heap index 0 *is interpreted as "the element at index* 0 *of* H_{tail}*" (i.e., the start of the tail value). When spliced, these two heaps represent the structural value* $\mathrm{Cons}\big(123,\ \mathrm{Cons}(321,\ \mathrm{Nil})\big)$.

4.3 Duplication as a fallback for sharing

When sharing is structurally impossible—because the type at a reuse site admits no wrapping constructor—the mutator falls back to *shallow duplication*. Shallow duplication is a two-phase process (Figure 7): it first *shallow-treeifies* the source value to obtain a finite tree exposing only the outer shape, and then *linearizes* that tree at the destination. The key invariant is that shallow treeification halts at every wrappable boundary, replacing the subvalue there by a wrapper rather than traversing through it. Duplication therefore copies $O(\mathrm{MaxDupSzT}^{\tau})$ tokens—bounded by the type structure alone, independent of data size.

Header discipline for tuples. Tuples admit no wrapping constructors, so duplication must traverse every field. The ST-Tup rule locates each field by reading the tuple header (for random-access fields, $j \in B$) or by calling EndWitness on the preceding field. To keep this traversal data-independent we impose:

> *For every tuple type* $(\tau_1 \otimes_1 \cdots \otimes_{n-1} \tau_n)$ *and every* $j \in \{1, \ldots, n-1\}$*: if* $\mathrm{MaxSzT}^{\tau_j} = \infty$*, then* $\otimes_j = \times$.

Under this discipline, EndWitness is invoked only on bounded-size types, so shallow treeification runs in $O(|\tau|)$ time—linear in the type structure, constant in the data.

$$\frac{\langle r_I, i_I\rangle; S \Downarrow_{\mathrm{ST}}^{\tau} V \quad S[r_O] = H_O \quad V; i_O; H_O \Downarrow_{\mathrm{L}}^{\tau} H'_O; i'_O \quad S' = S \cup \{r_O \mapsto H'_O\}}{\langle r_I, i_I\rangle; \langle r_O, i_O\rangle; S \Downarrow_{\mathrm{SD}}^{\tau} S'; i'_O}$$

$$\frac{S[r][i] = u}{\langle r, i\rangle; S \Downarrow_{\mathrm{ST}}^{\mathbf{uint}} u}\ \text{ST-Uint} \qquad \frac{}{\langle r, i\rangle; S \Downarrow_{\mathrm{ST}}^{()} ()}\ \text{ST-Emp}$$

$$S[r] = H \quad i_1 = i + |B| \quad \forall j \in \{2, \ldots, n\}.\ i_j = \begin{cases} H[i + |\{\, k \in B \mid k < j \,\}|] & \text{if } j \in B \\ \mathrm{EndWitness}^{\tau_{j-1}}(i_{j-1}, H) & \text{otherwise} \end{cases}$$
$$\frac{\forall j \in \{1, \ldots, n\}.\ \langle r, i_j\rangle; S \Downarrow_{\mathrm{ST}}^{\tau_j} V_j}{\langle r, i\rangle; S \Downarrow_{\mathrm{ST}}^{(\tau_1 \otimes_1 \cdots \otimes_{n-1} \tau_n)} (V_1, \ldots, V_n)}\ \text{ST-Tup}$$

$$\frac{\mathrm{Alias}(\widetilde{\tau}) \quad \mathrm{Ctr}(\widetilde{\tau}) = [\, K\, \sigma\,] \quad \langle r, i\rangle; S \Downarrow_{\mathrm{ST}}^{\sigma} V}{\langle r, i\rangle; S \Downarrow_{\mathrm{ST}}^{\widetilde{\tau}} K\, V}\ \text{ST-Alias} \qquad \frac{\mathrm{Bdy}(\widetilde{\tau})}{\langle r, i\rangle; S \Downarrow_{\mathrm{ST}}^{\widetilde{\tau}} K^{\widetilde{\tau}*}\, \langle r, i\rangle}\ \text{ST-Bdy}$$

Fig. 7. Shallow duplication $\Downarrow_{\mathrm{SD}}^{\tau}$ (top) factors through shallow treeification $\Downarrow_{\mathrm{ST}}^{\tau}$ (bottom) and linearization. The ST-Bdy rule halts at wrappable boundaries; the ST-Tup rule locates fields via header lookup ($j \in B$, where $B = \{\, j \mid 2 \le j \le n \ \wedge\ \otimes_{j-1} = \times \,\}$ is the set of random-access fields) or EndWitness otherwise.

$$\mathrm{MaxDupSzT}^{\mathbf{uint}} = 1,$$

$$\mathrm{MaxDupSzT}^{(\tau_1 \otimes_1 \cdots \otimes_{n-1} \tau_n)} = |\{\, k \in \{1, \ldots, n-1\} \mid \otimes_k = \times \,\}| \ +\ \sum_{i=1}^{n} \mathrm{MaxDupSzT}^{\tau_i},$$

$$\mathrm{MaxDupSzT}^{\widetilde{\tau}} = \begin{cases} 2, & \text{if } \mathrm{Bdy}(\widetilde{\tau}), \\ \mathrm{MaxDupSzT}^{\sigma}, & \text{if } \mathrm{Alias}(\widetilde{\tau}) \text{and } \mathrm{Ctr}(\widetilde{\tau}) = [\, K\, \sigma\,] \end{cases}$$

Fig. 8. Maximum token bound for shallow duplication. Non-alias datatypes contribute 2 tokens (wrapper tag and address) since shallow treeification halts at these boundaries.

Example 4.2 (Shallow treeification of an `ilist` payload). *We reuse* ilist *from Example 3.2 and the two-element list value* $\mathrm{Cons}\big(123,\ \mathrm{Cons}(321,\ \mathrm{Nil}())\big)$ *stored contiguously in* $S = \{\, r \mapsto [\,\mathrm{Cons},\ 123,\ \mathrm{Cons},\ 321,\ \mathrm{Nil}\,]\,\}$. *Shallow-treeifying the outer payload at* $\langle r, 1\rangle$ *with type* $(\mathbf{uint} \boxtimes \mathrm{ilist})$ *applies ST-Tup:* $B = \emptyset$, *so* $i_1 = 1$ *and* $i_2 = \mathrm{EndWitness}^{\mathbf{uint}}(1, H) = 2$. *The first field yields* 123 *by ST-Uint; the second halts at the* ilist *boundary by ST-Bdy, producing*

$$\langle r, 1\rangle; S \Downarrow_{\mathrm{ST}}^{(\mathbf{uint} \boxtimes \mathrm{ilist})} \big(123,\ K^{\mathrm{ilist}*}\, \langle r, 2\rangle\big).$$

Only the outer pair is materialized; the tail list is left in place.

Bounding duplication cost. The metafunction $\mathrm{MaxDupSzT}^{\tau}$ (Figure 8) gives a static upper bound on the tokens materialized by shallow duplication at type τ: it charges for tuple headers and traversal through non-wrappable structure, and resets at each wrapping constructor. Under the header discipline, the time cost of shallow treeification is $O(\mathrm{MaxDupSzT}^{\tau})$—bounded by the type alone.

Remark 4.3 (Reducing duplication cost). *When shallow duplication through tuple fields dominates runtime cost, we can factor the tuple into its own datatype with a reserved wrapping constructor.*

This adds a constant-size tag and read barrier but creates a new sharing boundary, typically reducing duplication cost substantially.

4.4 Moving garbage collection with adaptive linearization

Wrapping and splicing defer layout decisions during mutation, but fragmentation accumulates and must eventually be resolved. The garbage collector therefore serves a dual role: it reclaims unreachable data and re-linearizes reachable values, reconciling sharing with contiguity. Three aspects distinguish our collector from textbook copying GC: evacuation is typed, with τ guiding how many tokens are consumed and produced; wrappers are first-class runtime artifacts that may be removed or preserved during collection; and linearization decisions are revisited at collection time, not fixed at allocation. The core mechanism is a simple policy on wrappers: splice on first sight (eliminate the indirection when the target has not yet been copied), keep otherwise (preserve the wrapper to avoid duplicating a shared value). This differs from both traditional copying GC (which always preserves indirection) and eager flattening (which always splices).

The evacuation judgment. The kernel operation of the collector is the evacuation judgment shown in Figures 9–11:

$$\langle r_I, i_I\rangle;\ S_I;\ \langle r_O, i_O\rangle;\ S_O;\ F\ \Downarrow^{\tau}_{\mathrm{C}}\ i'_I;\ i'_O;\ S'_O;\ F'.$$

Intuitively, this judgment copies the value reachable from the source address $\langle r_I, i_I\rangle$ in the source store S_I into the destination store S_O, starting at index i_O, producing an updated destination store S'_O. The source cursor i'_I records how many tokens were consumed when reading the source value, while the destination cursor i'_O records how many tokens were produced.

The forwarding map F records which source addresses have already been evacuated. Beyond its usual role of preventing duplicate copying, F drives the splice/keep decision: when a wrapper's target is already in F, the collector keeps the wrapper; otherwise it splices.

Textbook cases. Primitives and the empty tuple copy tokens directly. Tuples and constructor applications proceed structurally, recursively evacuating payloads and back-patching header fields, as in the duplication judgment.[1] For a raw location-based address $\langle r_T, i_T\rangle$, the collector checks the forwarding map: if the address is already present it emits an updated pointer to the forwarded destination; otherwise it evacuates the target into a fresh region and records the forwarding entry.

Adaptive linearization via wrapping. The non-standard cases concern wrapped objects $K^{\widetilde{\tau}*}\ \langle r_T, i_T\rangle$, which mark points where the mutator deferred a layout decision. The collector applies the splice/-keep policy introduced above:

Splice ($\langle r_T, i_T\rangle \notin \mathrm{dom}(F)$). The collector evacuates the target value directly into the current destination position, eliminating the indirection, and records forwarding entries for both the target address and the wrapper's own address.

Keep ($\langle r_T, i_T\rangle \in \mathrm{dom}(F)$). Splicing would duplicate a shared value, so the collector preserves the wrapper, emitting a wrapper constructor and an updated address to the forwarded destination.

4.5 Correctness

Materialization at an address. A location-based address $\langle r, i\rangle$ does not denote a value directly; instead it designates a start index of a τ-segment inside region r of the store. We therefore introduce a derived observation judgment, *materialization* at type τ,

$$\langle r, i\rangle; S\ \Downarrow^{\tau}_{\mathrm{M}}\ V; i',$$

[1]The left-to-right ordering in the rules is chosen for definiteness; other evaluation orders, including parallel evacuation of tuple fields, are admissible.

$$\text{Forwarding maps} \quad \ni \quad F \quad ::= \{\langle r_1, i_1\rangle \mapsto \langle r_1', i_1'\rangle, \ldots\}$$

$$\frac{S_I[r_I][i_I] = u \quad \mathrm{Ins}(S_O, r_O, i_O, u) = S_O'}{\langle r_I, i_I\rangle; S_I; \langle r_O, i_O\rangle; S_O; F \Downarrow_{\mathrm{C}}^{\mathbf{uint}}\ i_I{+}1;\ i_O{+}1;\ S_O';\ F}\ \text{EV-Uint} \qquad \frac{}{\langle r_I, i_I\rangle; S_I; \langle r_O, i_O\rangle; S_O; F \Downarrow_{\mathrm{C}}^{()}\ i_I;\ i_O;\ S_O;\ F}\ \text{EV-Emp}$$

$$\frac{\begin{array}{c} i_{I_1} = i_I + |B| \quad i_{O_1} = i_O + |B| \quad \forall_{j\in\{1,\ldots,n\}}.\ \langle r_I, i_{I_j}\rangle; S_I; \langle r_O, i_{O_j}\rangle; S_{O_j}; F_j \Downarrow_{\mathrm{C}}^{\tau_j}\ i_{I_{j+1}};\ i_{O_{j+1}};\ S_{O_{j+1}};\ F_{j+1} \\ F_1 = F \quad S_{O_1} = S_O \quad H_{\mathrm{hdr}} = [\, i_{I_j} \mid j \in B \text{ in ascending order }] \quad \overline{\mathrm{Ins}}(S_{O_{n+1}}, r_O, i_O, H_{\mathrm{hdr}}) = S_O^{\mathrm{final}} \end{array}}{\langle r_I, i_I\rangle; S_I; \langle r_O, i_O\rangle; S_O; F \Downarrow_{\mathrm{C}}^{(\tau_1\otimes_1\cdots\otimes_{n-1}\tau_n)}\ i_{I_{n+1}};\ i_{O_{n+1}};\ S_O^{\mathrm{final}};\ F_{n+1}}\ \text{EV-Tuple}$$

$$\frac{\begin{array}{c} \mathrm{Alias}(\widetilde{\tau}) \quad \mathrm{Ctr}(\widetilde{\tau}) = [\, K\,\sigma\,] \\ \langle r_I, i_I\rangle; S_I; \langle r_O, i_O\rangle; S_O; F \Downarrow_{\mathrm{C}}^{\sigma}\ i_I';\ i_O';\ S_O';\ F_1 \\ F' = F_1 \cup \{\langle r_I, i_I\rangle \mapsto \langle r_O, i_O\rangle\} \end{array}}{\langle r_I, i_I\rangle; S_I; \langle r_O, i_O\rangle; S_O; F \Downarrow_{\mathrm{C}}^{\widetilde{\tau}}\ i_I';\ i_O';\ S_O';\ F'}\ \text{EV-Alias} \qquad \frac{\begin{array}{c} \mathrm{Bdy}(\widetilde{\tau}) \quad S_I[r_I][i_I] = K \quad (K\,\tau) \in \mathrm{Ctr}(\widetilde{\tau}) \\ \mathrm{Ins}(S_O, r_O, i_O, K) = S_O^{\mathrm{tag}} \quad K \neq K^{\widetilde{\tau}*} \\ \langle r_I, i_I{+}1\rangle; S_I; \langle r_O, i_O{+}1\rangle; S_O^{\mathrm{tag}}; F \Downarrow_{\mathrm{C}}^{\tau}\ i_I';\ i_O';\ S_O';\ F_1 \\ F' = F_1 \cup \{\langle r_I, i_I\rangle \mapsto \langle r_O, i_O\rangle\} \end{array}}{\langle r_I, i_I\rangle; S_I; \langle r_O, i_O\rangle; S_O; F \Downarrow_{\mathrm{C}}^{\widetilde{\tau}}\ i_I';\ i_O';\ S_O';\ F'}\ \text{EV-Bdy}$$

Fig. 9. Evacuation base cases: primitives, tuples, and data types ($B = \{j \mid 2 \le j \le n \ \wedge\ \otimes_{j-1} = \times\}$, the random-access fields).

$$\text{EV-PtrCopy}: \quad \frac{\begin{array}{c} S_I[r_I][i_I] = \langle r_T, i_T\rangle \notin \mathrm{dom}(F) \quad r_N \text{ fresh} \\ \langle r_T, i_T\rangle; S_I; \langle r_N, 0\rangle; S_O; F \Downarrow_{\mathrm{C}}^{\tau}\ i_T';\ j';\ S_O';\ F_1 \\ F' = F_1 \cup \{\langle r_T, i_T\rangle \mapsto \langle r_N, 0\rangle\} \\ \mathrm{Ins}(S_O', r_O, i_O, \langle r_N, 0\rangle) = S_O'' \end{array}}{\langle r_I, i_I\rangle; S_I; \langle r_O, i_O\rangle; S_O; F \Downarrow_{\mathrm{C}}^{\mathbf{laptr}\ \tau}\ i_I{+}1;\ i_O{+}1;\ S_O'';\ F'}$$

$$\text{EV-PtrShare}: \quad \frac{\begin{array}{c} S_I[r_I][i_I] = \langle r_T, i_T\rangle \quad F[\langle r_T, i_T\rangle] = \langle r_D, j_D\rangle \\ \mathrm{Ins}(S_O, r_O, i_O, \langle r_D, j_D\rangle) = S_O' \end{array}}{\langle r_I, i_I\rangle; S_I; \langle r_O, i_O\rangle; S_O; F \Downarrow_{\mathrm{C}}^{\mathbf{laptr}\ \tau}\ i_I{+}1;\ i_O{+}1;\ S_O';\ F}$$

Fig. 10. Evacuation: location-pointer cases (forward on miss, reuse on hit).

$$\text{EV-WrapSplice}: \quad \frac{\begin{array}{c} S_I[r_I][i_I] = K^{\widetilde{\tau}*} \quad S_I[r_I][i_I{+}1] = \langle r_T, i_T\rangle \notin \mathrm{dom}(F) \\ \langle r_T, i_T\rangle; S_I; \langle r_O, i_O\rangle; S_O; F \Downarrow_{\mathrm{C}}^{\widetilde{\tau}}\ i_T';\ i_O';\ S_O';\ F_1 \\ F' = F_1 \cup \{\, \langle r_I, i_I\rangle \mapsto \langle r_O, i_O\rangle, \langle r_T, i_T\rangle \mapsto \langle r_O, i_O\rangle \,\} \end{array}}{\langle r_I, i_I\rangle; S_I; \langle r_O, i_O\rangle; S_O; F \Downarrow_{\mathrm{C}}^{\widetilde{\tau}}\ i_I{+}2;\ i_O';\ S_O';\ F'}$$

$$\text{EV-WrapKeep}: \quad \frac{\begin{array}{c} S_I[r_I][i_I] = K^{\widetilde{\tau}*} \quad S_I[r_I][i_I{+}1] = \langle r_T, i_T\rangle \\ F[\langle r_T, i_T\rangle] = \langle r_D, j_D\rangle \quad \mathrm{Ins}(S_O, r_O, i_O, K^{\widetilde{\tau}*}) = S_O^{\mathrm{tag}} \\ \mathrm{Ins}(S_O^{\mathrm{tag}}, r_O, i_O{+}1, \langle r_D, j_D\rangle) = S_O'' \\ F' = F \cup \{\, \langle r_I, i_I\rangle \mapsto \langle r_O, i_O\rangle \,\} \end{array}}{\langle r_I, i_I\rangle; S_I; \langle r_O, i_O\rangle; S_O; F \Downarrow_{\mathrm{C}}^{\widetilde{\tau}}\ i_I{+}2;\ i_O{+}2;\ S_O'';\ F'}$$

Fig. 11. Evacuation: wrapper cases—splice on first sight, keep otherwise.

which reads the region $S[r]$ beginning at index i and returns the tree-shaped payload V together with end witness i'. Materialization is read-only: it allocates nothing and does not mutate S. The freestanding structural rules in Figure 12 mirror the treeification rules of Figure 3 but recurse through materialization, enabling the judgment to traverse wrapper and indirection constructors that lie outside treeification's domain.

Deep equality (store-sensitive equality). Pointers and wrapping constructors introduce representation choices that should not affect the *observed* value. We therefore define a store-indexed, deep

$$\frac{S[r][i] = u}{\langle r, i\rangle; S \;\Downarrow_{\mathrm{M}}^{\mathbf{uint}}\; u; i{+}1}\ \text{M-Uint} \qquad \frac{}{\langle r, i\rangle; S \;\Downarrow_{\mathrm{M}}^{()}\; (); i}\ \text{M-Emp}$$

$$\frac{i_1 = i + |B| \quad \forall_{j\in\{1,\ldots,n\}}.\ \langle r, i_j\rangle; S \;\Downarrow_{\mathrm{M}}^{\tau_j}\; V_j; i_{j+1}}{\langle r, i\rangle; S \;\Downarrow_{\mathrm{M}}^{(\tau_1\otimes\cdots\otimes_{n-1}\tau_n)}\; (V_1, \ldots, V_n); i_{n+1}}\ \text{M-Tup}$$

$$\frac{\mathrm{Alias}(\widetilde{\tau}) \;\; \mathrm{Ctr}(\widetilde{\tau}) = [K\,\sigma] \quad \langle r, i\rangle; S \;\Downarrow_{\mathrm{M}}^{\sigma}\; V; i'}{\langle r, i\rangle; S \;\Downarrow_{\mathrm{M}}^{\widetilde{\tau}}\; K\,V; i'}\ \text{M-Alias}$$

$$\frac{\mathrm{Bdy}(\widetilde{\tau}) \;\; S[r][i] = K \quad (K\,\sigma) \in \mathrm{Ctr}(\widetilde{\tau}) \quad \langle r, i{+}1\rangle; S \;\Downarrow_{\mathrm{M}}^{\sigma}\; V; i'}{\langle r, i\rangle; S \;\Downarrow_{\mathrm{M}}^{\widetilde{\tau}}\; K\,V; i'}\ \text{M-Bdy}$$

$$\text{M-Laptr}: \frac{S[r][i] = \langle r', i'\rangle \quad \langle r', i'\rangle; S \;\Downarrow_{\mathrm{M}}^{\tau}\; V; i''}{\langle r, i\rangle; S \;\Downarrow_{\mathrm{M}}^{\mathbf{laptr}\ \tau}\; \langle r', i'\rangle; i+1} \qquad \text{M-Wrap}: \frac{S[r][i] = K^{\widetilde{\tau}*} \quad \langle r, i+1\rangle; S \;\Downarrow_{\mathrm{M}}^{\mathbf{laptr}\ \widetilde{\tau}}\; \langle r', i'\rangle; i+2}{\langle r, i\rangle; S \;\Downarrow_{\mathrm{M}}^{\widetilde{\tau}}\; K^{\widetilde{\tau}*}\langle r', i'\rangle; i+2}$$

Fig. 12. Materialization of a value at an address; B as in Figure 3. Rules M-Captr and M-Chunk extend this judgment in §5.4.

$$\frac{n = n'}{S \vdash n \equiv^{\mathbf{uint}} n'}\ \text{DE-Uint} \qquad \frac{\forall_k.\ S \vdash V_k \equiv^{\tau_k} W_k}{S \vdash (V_1, \ldots, V_n) \equiv^{\tau_1\otimes\cdots\otimes\tau_n} (W_1, \ldots, W_n)}\ \text{DE-Tup}$$

$$\frac{S \vdash V \equiv^{\tau} W \quad (K\,\tau) \in \mathrm{Ctr}(\widetilde{\tau})}{S \vdash K\,V \equiv^{\widetilde{\tau}} K\,W}\ \text{DE-Data} \qquad \frac{\langle r, i\rangle; S \Downarrow_{\mathrm{M}}^{\tau} V; i_1 \quad \langle r', i'\rangle; S \Downarrow_{\mathrm{M}}^{\tau} W; i_2 \quad S \vdash V \equiv^{\tau} W}{S \vdash \langle r, i\rangle \equiv^{\mathbf{laptr}\ \tau} \langle r', i'\rangle}\ \text{DE-Laptr}$$

$$\frac{\langle r, i\rangle; S \Downarrow_{\mathrm{M}}^{\widetilde{\tau}} V; i' \quad S \vdash V \equiv^{\widetilde{\tau}} W}{S \vdash K^{\widetilde{\tau}*}\ \langle r, i\rangle \equiv^{\widetilde{\tau}} W}\ \text{DE-WrapL} \qquad \frac{\langle r, i\rangle; S \Downarrow_{\mathrm{M}}^{\widetilde{\tau}} W; i' \quad S \vdash V \equiv^{\widetilde{\tau}} W}{S \vdash V \equiv^{\widetilde{\tau}} K^{\widetilde{\tau}*}\ \langle r, i\rangle}\ \text{DE-WrapR}$$

Fig. 13. Deep equality modulo wrapping and addresses. Rules in §5.4 extend this judgment.

equality relation

$$S \vdash V \equiv^{\tau} W,$$

given in Figure 13. Intuitively: primitives are compared by value; tuples are compared componentwise; and ordinary constructors are compared by tag and payload. Pointers are compared by materializing their targets and comparing the resulting payloads. Wrapped objects are equal to whatever their (materialized) target is equal to. Thus $\equiv^{\tau}$ is insensitive to whether a substructure is represented inline or indirectly via a wrapper/pointer.

Lemma 4.4 (Deep equality is an equivalence relation). *Fix a type τ and a store S.*

(1) (Reflexive, whenever well-typed) If $\langle r, i\rangle; S \Downarrow_{\mathrm{M}}^{\tau} V; i'$, then $S \vdash V \equiv^{\tau} V$.
(2) (Symmetric) If $S \vdash V \equiv^{\tau} W$, then $S \vdash W \equiv^{\tau} V$.
(3) (Transitive) If $S \vdash V \equiv^{\tau} W$ and $S \vdash W \equiv^{\tau} U$, then $S \vdash V \equiv^{\tau} U$.

4.5.1 Shallow treeification. We now state the fundamental semantic guarantee provided by shallow treeification (Figure 7): shallow treeification exposes only the outer structure of a store segment, but it is observationally equivalent to fully materializing that segment.

Lemma 4.5 (Shallow treeification preserves deep equality). *Suppose shallow treeification succeeds:*

$$\langle r, i\rangle;\ S\ \Downarrow^{\tau}_{\mathrm{ST}}\ W.$$

Assume further that the same source segment materializes at type τ:

$$\langle r, i\rangle;\ S\ \Downarrow^{\tau}_{\mathrm{M}}\ V;\ i'.$$

Then the shallow tree value is observationally equal to the fully materialized value:

$$S \vdash V \equiv^{\tau} W.$$

4.5.2 Bound on the space use of value duplication. In addition to semantic preservation above, we rely on a size guarantee: shallow duplication never writes more than the static upper bound $\mathrm{MaxDupSzT}^{\tau}$ tokens into the destination region. This property is what makes the duplication discipline compatible with our space/cost analyses.

Lemma 4.6 (Token bound correctness for value duplication). *Suppose*

$$\langle r_I, i_I\rangle;\ \langle r_O, i_O\rangle;\ S\ \Downarrow^{\tau}_{\mathrm{SD}}\ S';\ i'_O.$$

Then the number of tokens written at the destination is bounded by Figure 8:

$$i'_O - i_O\ \le\ \mathrm{MaxDupSzT}^{\tau}.$$

Equivalently, if $S[r_O] = H_O$ and $S'[r_O] = H'_O$, then

$$|H'_O| - |H_O|\ \le\ \mathrm{MaxDupSzT}^{\tau},$$

whenever the duplication derivation is invoked at the end of the destination heap (i.e., $|H_O| = i_O$).

4.5.3 Store evacuation. The evacuation judgment

$$\langle r_I, i_I\rangle;\ S_I;\ \langle r_O, i_O\rangle;\ S_O;\ F\ \Downarrow^{\tau}_{\mathrm{C}}\ i'_I;\ i'_O;\ S'_O;\ F'$$

moves (and adaptively linearizes) the value reachable from the source address $\langle r_I, i_I\rangle$ into the destination store S_O at index i_O. Correctness is phrased in terms of deep equality: the evacuated result, when materialized in the output store, is observationally equal to the source value, even though the collector may splice wrappers (improving contiguity) or preserve them (preserving sharing) according to the forwarding map.

Lemma 4.7 (Evacuation preserves deep equality). *Assume the evacuation judgment succeeds:*

$$\langle r_I, i_I\rangle;\ S_I;\ \langle r_O, i_O\rangle;\ S_O;\ F\ \Downarrow^{\tau}_{\mathrm{C}}\ i'_I;\ i'_O;\ S'_O;\ F'.$$

Assume further that the source address materializes at type τ:

$$\langle r_I, i_I\rangle; S_I\ \Downarrow^{\tau}_{\mathrm{M}}\ V; i''_I.$$

Assume moreover that F is sound for (S_I, S_O) with some type assignment Φ (as defined in Definition A.4). Then $i'_I = i''_I$, there exists W such that

$$\langle r_O, i_O\rangle; S'_O\ \Downarrow^{\tau}_{\mathrm{M}}\ W; i'_O\quad \text{and}\quad S'_O \vdash V \equiv^{\tau} W,$$

and F' is sound for (S_I, S'_O) with a type assignment $\Phi' \supseteq \Phi$.

All of the proofs in § 3 and § 4 are mechanized in Rocq; see Appendix A.3 for details.

$$\begin{array}{lrcll}
\text{(a) Content addresses} & \text{Digests} & \ni & d & \\
\text{(b) Type System} & \text{Types} & \ni & \tau & ::= \ldots \mid \mathbf{captr}\ \tau \\
\text{(c) Structural} & \text{Tree-shaped values} & \ni & V & ::= \ldots \mid \langle d, i\rangle \\
\text{(c) Linearized} & \text{Tokens} & \ni & T & ::= \ldots \mid \langle d, i\rangle
\end{array}$$

$$\mathrm{MaxSzT}^{\mathbf{captr}\ \tau}(R) = 1 \quad \mathrm{MaxDupSzT}^{\mathbf{captr}\ \tau} = 1$$

$$\frac{H[i] = \langle d, i'\rangle}{i; H \Downarrow_{\mathrm{T}}^{\mathbf{captr}\ \tau} \langle d, i'\rangle;\ i{+}1}\ \text{T-Captr} \qquad \frac{H' = H +\!\!+ [\langle d, i'\rangle]}{\langle d, i'\rangle;\ i; H \Downarrow_{\mathrm{L}}^{\mathbf{captr}\ \tau} H';\ i{+}1}\ \text{L-Captr} \qquad \frac{S[r][i] = \langle d, i'\rangle}{\langle r, i\rangle; S \Downarrow_{\mathrm{ST}}^{\mathbf{captr}\ \tau} \langle d, i'\rangle}\ \text{ST-Captr}$$

$$\frac{T = S_I[r_I][i_I] \quad \mathrm{Ins}(S_O, r_O, i_O, T) = S'_O \quad F' = F \cup \{\langle r_I, i_I\rangle \mapsto \langle r_O, i_O\rangle\}}{\langle r_I, i_I\rangle; S_I; \langle r_O, i_O\rangle; S_O; F \Downarrow_{\mathrm{C}}^{\mathbf{captr}\ \tau} i_I{+}1;\ i_O{+}1;\ S'_O;\ F'}\ \text{EV-Captr}$$

Fig. 14. CAM extensions: type and value grammar, size bounds, and treeification/linearization/duplication/evacuation rules for **captr** τ.

5 Open-World Store: Canonical Bytes and Content-Addressed Memory

We now extend the LAM store model with content-addressable memory. At the type level, CAM adds a distinguished type **captr** τ whose values are *content pointers* $\langle d, i\rangle$—a digest d paired with an offset i into the corresponding byte sequence. Unlike location pointers, content pointers are stable, serializable, and freely communicable across machines. Figure 14 summarizes the type, value, and judgment extensions. We write U for a finite byte sequence of length $|U|$, i_B for a byte index with $0 \le i_B \le |U|$, and $U +\!\!+ V$ for concatenation.

Two operations connect LAM and CAM. *Commit*, $\mathrm{CAMCommit}(\tau, \langle r, i\rangle, S) = d$, linearizes the value at $\langle r, i\rangle$ into a canonical byte sequence, stores it in the external CAM, and returns its digest; it is observationally pure. Internally, commit factors into a traversal that copies the value into a work buffer H while computing a rolling hash h ($\Downarrow_{\mathrm{W}}^{\tau}$, detailed in §5.2), followed by canonical encoding and persistence via `CAMStore`:

$$\frac{\langle r, i\rangle; S; [\,] \Downarrow_{\mathrm{W}}^{\tau} i'; H; h \quad \texttt{CAMStore}(\tau, S, H) = d}{\mathrm{CAMCommit}(\tau, \langle r, i\rangle, S) = d} \qquad \frac{r \text{ fresh} \quad S' = S \cup \{r \mapsto H\} \quad \langle r, 0\rangle; S'; 0; \epsilon \Downarrow_{\mathrm{E}}^{\tau} U;\ i'_H;\ i'_B}{\texttt{CAMStore}(\tau, S, H) = \mathrm{put}(U)}$$

Here $\mathrm{put}(U)$ and $\mathrm{get}(d)$ are the external store interface: $\mathrm{put}(U)$ persists a byte sequence U and returns its content digest d; $\mathrm{get}(d)$ retrieves the byte sequence previously stored under d. *Dereference*, $\mathrm{CAMDeref}(\tau, d) = \langle r, S\rangle$, reverses commit: it fetches bytes by digest, decodes them into a fresh heap, and installs the result in a new region:

$$\frac{U = \mathrm{get}(d) \quad 0; U; 0; [\,] \Downarrow_{\mathrm{D}}^{\tau} H;\ i_H;\ i_B \quad r \text{ fresh} \quad S = \{\, r \mapsto H \,\}}{\mathrm{CAMDeref}(\tau, d) = \langle r, S\rangle}$$

5.1 Canonical byte encoding

A location pointer $\langle r, i\rangle$ is meaningful only within a particular store; allowing such pointers into serialized byte sequences would break portability. We enforce this boundary with a type-directed *openness predicate*: a type is *open* when its values contain no bare location pointers. For a type τ and a set R of datatype identifiers visited on the current descent, $\mathrm{Open}^{\tau}(R) \in \{\mathsf{true}, \mathsf{false}\}$ is defined by (writing $\mathrm{Open}^{\tau} \triangleq \mathrm{Open}^{\tau}(\emptyset)$):

$$\mathrm{Open}^{\mathbf{uint}}(R) = \mathsf{true} \qquad \mathrm{Open}^{()}(R) = \mathsf{true} \qquad \mathrm{Open}^{\mathbf{laptr}\ \sigma}(R) = \mathsf{false} \qquad \mathrm{Open}^{\mathbf{captr}\ \sigma}(R) = \mathrm{Open}^{\sigma}(R)$$

$$\begin{aligned}
\mathrm{Open}^{(\tau_1 \otimes_1 \cdots \otimes_{n-1} \tau_n)}(R) &= \textstyle\bigwedge_{i=1}^{n} \mathrm{Open}^{\tau_i}(R) \\
\mathrm{Open}^{\widetilde{\tau}}(R) &= \mathsf{true} \quad \text{if } \widetilde{\tau} \in R \\
\mathrm{Open}^{\widetilde{\tau}}(R) &= \textstyle\bigwedge_{K\,\sigma \in \mathrm{Ctr}(\widetilde{\tau})} \mathrm{Open}^{\sigma}(R \cup \{\widetilde{\tau}\}) \quad \text{otherwise}
\end{aligned}$$

Example 5.1 (Openness predicate).

$$\mathrm{Open}^{(\mathbf{uint} \otimes \mathbf{laptr\ uint})} = \mathrm{Open}^{\mathbf{uint}}(\emptyset) \ \wedge\ \mathrm{Open}^{\mathbf{laptr\ uint}}(\emptyset) \quad = \mathsf{true} \ \wedge\ \mathsf{false} = \mathsf{false}.$$

Although the openness predicate rejects bare location pointers, it permits wrappers—which themselves contain location pointers. This is intentional: wrappers introduced by adaptive serialization are *spliced* during encoding, so no location-based address appears in the byte stream.

Given an open type, a codec specifies a type-directed encoding judgment and its inverse:

$$\langle r, i\rangle; S; i_B; U \ \Downarrow^{\tau}_{\mathrm{E}}\ U';\ i';\ i'_B \qquad i_B; U;\ i; H \ \Downarrow^{\tau}_{\mathrm{D}}\ H';\ i'_H;\ i'_B$$

We assume a finite byte alphabet and a deterministic, type-directed mapping from values to byte sequences; concrete choices such as word size are left abstract. Appendix B.1 describes the codec in more detail.

5.2 Content-defined chunking

Figure 15 defines the internal CAM-introduction judgment $\langle r, i\rangle; S; H \ \Downarrow^{\tau}_{\mathrm{W}}\ i';\ H';\ h$: starting from $\langle r, i\rangle$ in store S, it copies the value of type τ into the work buffer H, producing H', and emits a rolling hash h. Primitives and content pointers copy tokens directly; tuples emit headers and recurse left-to-right; alias datatypes collapse through the single constructor; and LAM wrappers $K^{\widetilde{\tau}*}$ are spliced through, continuing from the wrapped location.

The key addition over treeification is the rule for boundary datatypes, CI-Bdy: at each application of a boundary constructor, the rule may cut the buffer, commit the suffix as a chunk via `CAMStore`, and replace it in the buffer with a $K^{\widetilde{\tau}\bullet}$ wrapper containing the resulting digest. We adapt *content-defined chunking* (CDC) [16, 31] — a byte-stream technique in which the data itself selects the chunk boundaries — to typed recursive structures. Boundary constructor applications serve as *potential chunk points* (PCPs); their spacing is governed by MaxDupSzT$^{\cdot}$, the same type-dependent measure that bounds shallow duplication cost.

At each PCP, two predicates decide whether to cut. The *rolling-hash predicate* $h' \bmod \kappa = 0$ fires whenever the incrementally maintained hash hits zero modulo κ, which makes chunk boundaries *data-defined*: a small edit perturbs only the chunks immediately around it, leaving distant boundaries unchanged—unlike fixed-size chunking, where a one-position shift cascades through every boundary downstream. The parameter κ controls expected chunk size, at roughly one cut per κ PCPs. The *chunk-size cap predicate* $|H'| - n > C$ fires whenever the per-chunk token count exceeds C, giving a deterministic upper bound on chunk size where the rolling hash alone provides only an expected-case bound. The two predicates address orthogonal concerns: the hash positions boundaries by content so that local edits stay local, while the cap supplies a worst-case token bound the hash alone cannot guarantee.

Both predicates depend only on the subtree rooted at the PCP: the rolling hash receives no input from outside the subtree, and the residual $|H'| - n$ measures only the tokens this constructor's traversal contributed. Consequently, two structurally identical subvalues—whether at different positions in the same value, in different versions sharing a substructure, or on different machines committing the same value to a shared store—produce identical chunk decompositions and identical digests. This *content-defined purity* is what makes CAM useful as a shared deduplicated store—dedup is automatic across versions and across machines. It is also the precondition for the locality theorem of §5.3: it lets the analysis reason about a path edit through the rolling-hash residues

$$\frac{S[r][i] = u}{\langle r, i\rangle; S; H \;\Downarrow_{\mathrm{W}}^{\mathbf{uint}}\; i{+}1;\; H +\!\!+ [\,v\,];\; \mathrm{Hash}(u)}\;\text{CI-UINT} \qquad \frac{}{\langle r, i\rangle; S; H \;\Downarrow_{\mathrm{W}}^{()}\; i;\; H;\; \mathrm{Hash}(0)}\;\text{CI-UNIT}$$

$$\frac{\begin{array}{c} i_1 = i + |B| \quad H_{\mathrm{hdr}} = [\,i_j \mid j \in B \text{ ascending }\,] \quad H_1 = H +\!\!+ H_{\mathrm{hdr}} \\ \forall_{j\in\{1,\ldots,n\}}.\ \langle r, i_j\rangle; S; H_j \Downarrow_{\mathrm{W}}^{\tau_j} i_{j+1};\; H_{j+1};\; h_{j+1} \\ h = \mathrm{combine}(h_1, \ldots, h_n, \otimes_1, \ldots, \otimes_{n-1}) \end{array}}{\langle r, i\rangle; S; H \;\Downarrow_{\mathrm{W}}^{(\tau_1 \otimes_1 \cdots \otimes_{n-1} \tau_n)}\; i_{n+1};\; H_{n+1};\; h}\;\text{CI-TUPLE}$$

$$\frac{\mathrm{Alias}(\widetilde{\tau}) \quad \mathrm{Ctr}(\widetilde{\tau}) = [\,K\ \sigma\,] \quad \langle r, i\rangle; S; H \Downarrow_{\mathrm{W}}^{\sigma} i';\; H';\; h}{\langle r, i\rangle; S; H \;\Downarrow_{\mathrm{W}}^{\widetilde{\tau}}\; i';\; H';\; h}\;\text{CI-ALIAS}$$

$$\frac{\begin{array}{c} \mathrm{Bdy}(\widetilde{\tau}) \quad S[r][i] = K \quad K \neq K^{\widetilde{\tau}*} \quad (K\ \sigma) \in \mathrm{Ctr}(\widetilde{\tau}) \quad n = |H| \\ H_0 = H +\!\!+ [\,K\,] \quad \langle r, i{+}1\rangle; S; H_0 \Downarrow_{\mathrm{W}}^{\sigma} i';\; H';\; h \quad h' = \mathrm{combine}(K, h) \\ \begin{cases} h' \bmod \kappa = 0 \;\vee\; |H'| - n > C & : d = \mathtt{CAMStore}(\widetilde{\tau}, S, H'[n..|H'|)) \\ & \quad H'' = H'[0..n) +\!\!+ [\,K^{\widetilde{\tau}\bullet}, \langle d, 0\rangle\,] \\ \text{otherwise} & : H'' = H' \end{cases} \end{array}}{\langle r, i\rangle; S; H \;\Downarrow_{\mathrm{W}}^{\widetilde{\tau}}\; i';\; H'';\; h'}\;\text{CI-BDY}$$

$$\text{CI-WRAPSPLICE}: \frac{S[r][i] = K^{\widetilde{\tau}*} \quad S[r][i+1] = \langle r', j\rangle \quad \langle r', j\rangle; S; H \Downarrow_{\mathrm{W}}^{\widetilde{\tau}} j';\; H';\; h}{\langle r, i\rangle; S; H \;\Downarrow_{\mathrm{W}}^{\widetilde{\tau}}\; i{+}2;\; H';\; h}$$

$$\text{CI-CAPTR}: \frac{S[r][i] = \langle d, j\rangle}{\langle r, i\rangle; S; H \;\Downarrow_{\mathrm{W}}^{\mathbf{captr}\ \tau}\; i{+}1;\; H +\!\!+ [\langle d, j\rangle];\; \mathrm{Hash}(d, j)}$$

Fig. 15. Internal CAM-introduction rules. The CI-BDY rule cuts a chunk when either the rolling-hash predicate $h' \bmod \kappa = 0$ fires (hash-triggered) or the per-chunk token count $|H'| - n$ exceeds the cap C (cap-triggered); CI-WRAPSPLICE splices through LAM wrappers.

at the rebuilt constructors alone, without modeling a value-level transformation. The proof is by structural induction on the traversed type and is mechanized in Rocq; see Appendix B.2.

5.3 Locality of chunk decomposition

We designed our CDC approach so that small edits to a value perturb only a small number of chunks. This section states the locality guarantee precisely and illustrates its implications; details, supporting lemmas, and a summary of the Lean mechanization appear in Appendix B.2. Our analysis considers *path edits*. In a purely functional setting, modifying a value at depth d requires rebuilding the d constructors on the path from root to edit point, while all other substructures are shared via location pointers. We call this a *path edit of depth d*. For a tree with branching factor b, a depth-d edit leaves $s = (b-1) \cdot d$ unchanged sibling subtrees hanging off the edit path.

Two distinct costs are relevant when measuring the impact of a path edit on the chunk structure. First is *chunk count*: the number of chunks whose content differs between the original and updated values. This determines how many new chunks must be stored and how many digests must be computed. Second is *spillover*: the amount of content from unchanged sibling subtrees that must be retraversed because it falls within an affected chunk. This determines the computational work per update beyond the d rebuilt constructors themselves.

Under a standard hash-independence assumption—that the boundary event $h(p) \bmod \kappa = 0$ occurs with probability $1/\kappa$ at each PCP p, and that boundary decisions at distinct PCPs are mutually independent (Assumption B.2)—we establish the following result. Write $\mathcal{B}(\tau, V, S)$ for the multiset of content digests produced by the recursive calls to `CAMStore` during the evaluation

of CAMCommit$(\tau, \langle r, i\rangle, S)$, where $\langle r, i\rangle; S \Downarrow^{\tau}_{\mathrm{M}} V;\ i'$. Write $A \triangle B$ for the multiset symmetric difference of A and B.

Theorem 5.2 (Locality of chunk decomposition). *Assume* Open^{τ}*, and a chunk-size cap* $C = c\kappa$ *for a constant* $c \geq 1$*. Let* V *be a value of type* τ *in store* S*, and let* V' *in* $S' \supseteq S$ *be obtained from* V *by a path edit of depth* d *with* s *unchanged sibling subtrees. Then:*

(i) ***Chunk count.*** $\mathbb{E}\big[\,|\mathcal{B}(\tau, V, S) \triangle \mathcal{B}(\tau, V', S')|\,\big] \leq 2(d/\kappa + 1) + \lceil d/(c\kappa)\rceil = O(d/\kappa + 1)$.

(ii) ***Spillover cost.*** $\mathbb{E}[\,\text{spillover}\,] \leq \sum_{j=1}^{s} \min(c\kappa,\ \kappa - 1,\ h_{\tau_{R_j}}) \leq s \cdot (\kappa - 1)$*, where* $h_{\tau_{R_j}}$ *is the static leftmost-path PCP bound for type* τ_{R_j} *of the* j*th sibling (Lemma B.4).*

(iii) ***Total affected constructors.*** $\mathbb{E}[\,\text{affected constructors}\,] \leq d + s \cdot (\kappa - 1)$.

Corollary 5.3 (Balanced-binary-tree locality). *Let* V *be a value whose materialized shape is a balanced binary tree of* n *constructor applications. A path edit at any leaf has depth* $d = \lfloor \log_2 n \rfloor$ *and yields* $s = d$ *unchanged sibling subtrees, each of leftmost-path PCP bound at most* d *(Definition B.1). Choosing* $\kappa = \lfloor \log_2 n \rfloor$ *and* $C = \kappa$ *(so* $c = 1$*) in Theorem 5.2 yields, in expectation,* $O(1)$ *affected chunks and a total of* $d + s \cdot (\kappa - 1) \leq d \cdot \kappa = O(\log^2 n)$ *affected constructors per edit.*

Example 5.4 (Balanced binary tree, $n \approx 10^6$). *Instantiating Corollary 5.3 at* $n = 2^{20} \approx 10^6$ *constructors gives* $d = 20$*,* $s = 20$*, and recommended* $\kappa = C = 20$*. Plugging into Theorem 5.2: each of the* $s = 20$ *unchanged sibling subtrees contributes at most* $\kappa - 1 = 19$ *to expected spillover, giving total expected spillover* ≤ 380 *constructors and total affected content* $\leq d + s(\kappa - 1) = 400$ *constructors per edit; expected chunk count is* $\leq 2(20/20 + 1) + \lceil 20/20\rceil = 5$*. Reducing* κ *below the recommended value cuts spillover at the cost of more chunks (e.g.* $\kappa = 16$ *gives spillover* ≤ 300 *but chunk count* ≤ 6.5*); raising it well above inflates spillover linearly (at* $\kappa = 512$*, spillover rises to* $20 \cdot 511 \approx 10^4$ *while chunk count only drops to* ≈ 3*). By comparison, whole-value hashing requires* $O(n)$ *work on any edit; per-node hashing [15] achieves* $O(\log n)$ *work but stores* $O(n)$ *digests. Our decomposition stores* $O(n/\log n)$ *digests and requires* $O(\log^2 n)$ *work per edit (Corollary 5.3), trading a modest increase in update cost for a* $\log n$*-fold reduction in space overhead.*

Both predicates are essential, each addressing a failure the other cannot:

- **Without the hash**, chunks are bounded at C tokens by the cap alone, but cap-cuts cluster at structural positions: for a balanced binary tree, every $\Theta(\log_2 C)$ levels, so a path edit of depth d crosses $\Theta(d/\log_2 C)$ chunks rather than the hash's shape-independent $O(d/\kappa + 1)$ — $\Theta(\log n/\log\log n)$ vs $O(1)$ at $\kappa = C = \log n$. Tree shape also matters: path-like values give $\Theta(d/C)$ (better than balanced), while adversarial shapes can force $\Theta(d)$.
- **Without the cap**, residuals grow supercritically: a recurrence with growth factor $r = 2(1-1/\kappa) > 1$ for $\kappa > 2$ gives $\Theta(n^{\log_2(2-2/\kappa)})$ at the root of a balanced binary subtree — roughly $\Theta(n^{0.93})$ at $\kappa = 20$ and $n = 10^6$, a substantial fraction of the value.

Although κ is a tunable parameter in the theorem, it is not truly unconstrained in practice. The locality bound itself saturates around $\kappa \approx \log n$ for tree-structured data (Corollary 5.3), and chunks of a few hundred payload bytes already amortize a 32-byte digest below 1% overhead, so the practical range is bounded above and below by the analysis itself. Concrete deployments inherit further constraints from their storage substrate: an IPFS-backed CAM, for instance, has plenty of room at the upper end (Bitswap blocks up to 2 MiB [9] exceed the few-KiB chunks $\kappa \approx \log n$ produces) but operating below IPFS's 256 KiB UnixFS default [10] incurs modest per-block overhead from internals tuned for that size. One constraint is distinctive to content-addressed storage: κ determines the chunk decomposition (and hence the digests), so any distributed deployment must agree on κ as a protocol constant, since changing it invalidates all previously computed content identifiers.

5.4 Correctness: round trips between CAM and LAM

The central correctness property of our CAM extensions is that committing a value and immediately loading it back recovers a deeply equal value (Lemma 5.7). This rests on two intermediate results: the codec round-trip (Lemma 5.5) and the soundness of the internal CAM-introduction judgment (Lemma 5.8). We have mechanized these three lemmas in Rocq; details of the mechanization appear in Appendix B.

We begin with the codec. Encoding a well-typed, open value and then decoding the resulting bytes into a fresh heap recovers a deeply equal value:

Lemma 5.5 (Encode–decode round-trip). *Let τ be a type with* Open^{τ}*. Suppose the source address materializes at type τ:*

$$\langle r, i\rangle; S \;\Downarrow_{\mathrm{M}}^{\tau}\; V; i'$$

and that encoding succeeds from the same address, with the byte cursor at the end of the input buffer:

$$\langle r, i\rangle; S; i_B; U \;\Downarrow_{\mathrm{E}}^{\tau}\; U';\; i'';\; i'_B \qquad \textit{where} \qquad i_B = |U|.$$

Then for every heap H, heap index $i_H = |H|$, and region name $r_O \notin \mathrm{dom}(S)$, there exist a heap H', a heap index i'_H, and a value W such that:

(1) Decoding succeeds. $i_B;\ U';\ i_H;\ H \;\Downarrow_{\mathrm{D}}^{\tau}\; H';\ i'_H;\ i'_B$.
(2) The decoded segment materializes. $\langle r_O, i_H\rangle;\ \{r_O \mapsto H'\} \;\Downarrow_{\mathrm{M}}^{\tau}\; W;\ i'_H$.
(3) Deep equality. $S \cup \{r_O \mapsto H'\} \vdash V \equiv^{\tau} W$.
(4) End-witness agreement. $i'' = i'$.

The two cursor-at-end conditions $i_B = |U|$ and $i_H = |H|$ are the operational invariants the encoder and decoder maintain: encoding appends bytes to U starting at $|U|$, and decoding appends tokens to H starting at $|H|$. The proof is a straightforward induction on the type, with one subtlety: our encode and decode rules are deliberately undefined on bare LAM pointers, so the induction must establish that wrapper splicing correctly eliminates all location-based addresses before they reach the codec.

To state CAM correctness, we extend deep equality with two rules. The first compares two **captr** values by dereferencing both digests and comparing the resulting materialized values:

$$\frac{\begin{array}{c}\mathrm{CAMDeref}(\tau, d) = (r_1, S_1) \quad \mathrm{CAMDeref}(\tau, d') = (r_2, S_2) \\ \langle r_1, i\rangle; S_1 \Downarrow_{\mathrm{M}}^{\tau} V; i_1 \quad \langle r_2, i'\rangle; S_2 \Downarrow_{\mathrm{M}}^{\tau} W; i_2 \quad S \cup S_1 \cup S_2 \vdash V \equiv^{\tau} W\end{array}}{S \vdash \langle d, i\rangle \equiv^{\mathbf{captr}\ \tau} \langle d', i'\rangle}\ \text{DE-Captr}$$

The second and third handle $K^{\widetilde{\tau}\bullet}$ wrappers introduced by chunking, which must be transparent: a chunked value should be deeply equal to its unchunked counterpart (and vice versa).

$$\frac{\mathrm{CAMDeref}(\tau, d) = (r, S') \quad S'' = S \cup S' \quad \langle r, i\rangle; S'' \Downarrow_{\mathrm{M}}^{\widetilde{\tau}} V; i' \quad S'' \vdash V \equiv^{\widetilde{\tau}} W}{S \vdash K^{\widetilde{\tau}\bullet}\ \langle d, i\rangle \equiv^{\widetilde{\tau}} W}\ \text{DE-ChunkL}$$

$$\frac{\mathrm{CAMDeref}(\tau, d) = (r, S') \quad S'' = S \cup S' \quad \langle r, i\rangle; S'' \Downarrow_{\mathrm{M}}^{\widetilde{\tau}} W; i' \quad S'' \vdash V \equiv^{\widetilde{\tau}} W}{S \vdash V \equiv^{\widetilde{\tau}} K^{\widetilde{\tau}\bullet}\ \langle d, i\rangle}\ \text{DE-ChunkR}$$

This design choice allows CAM correctness to be expressed as *semantic transparency* statements.

Similarly we extend materialization with rules:

$$\text{M-Captr}: \frac{\begin{array}{c}S[r][i] = \langle d, i_1\rangle \quad \mathrm{CAMDeref}(\tau, d) = (r_1, S_1) \\ \langle r_1, i_1\rangle; S_1 \Downarrow_{\mathrm{M}}^{\tau} V; i'_1\end{array}}{\langle r, i\rangle; S \Downarrow_{\mathrm{M}}^{\mathbf{captr}\ \tau} \langle d, i_1\rangle; i+1}$$

$$\text{M-Chunk}: \frac{S[r][i] = K^{\widetilde{\tau}\bullet} \quad \langle r, i+1\rangle; S \;\Downarrow_{\mathrm{M}}^{\mathbf{captr}\ \widetilde{\tau}}\; \langle d, i_1\rangle; i+2}{\langle r, i\rangle; S \;\Downarrow_{\mathrm{M}}^{\widetilde{\tau}}\; K^{\widetilde{\tau}\bullet}\langle d, i_1\rangle; i+2}$$

The extended relation remains well-founded because content-addressed digests form a DAG: the digest of a value is computed from its serialized bytes, so d cannot appear among the sub-values of the datum it names. Each dereference therefore descends strictly in DAG depth, and the overall induction is well-founded by lexicographic order on (DAG depth, type structure).

The correctness argument assumes only an abstract external store:

Assumption 5.6 (External-store laws). *There is an abstract map $\mathcal{E}$ from digests to byte sequences such that* $\mathrm{put}(U)$ *returns a digest d and extends $\mathcal{E}$ with $\mathcal{E}(d) = U$, and* $\mathrm{get}(d)$ *returns $\mathcal{E}(d)$ for any $d \in \mathrm{dom}(\mathcal{E})$. Furthermore, the canonical heap obtained by decoding any byte sequence in $\mathcal{E}$ is Loc-free (no* **laptr** *tokens): every byte sequence ever deposited in $\mathcal{E}$ was produced by the encoder, and by inspection of the encoding rules the encoder never emits* **laptr** *bytes.*

The main correctness statement follows. Beyond CAMCommit succeeding (which implicitly requires that all wrapper targets reachable from the source are present in the store), the lemma takes two well-formedness preconditions on the caller's state: a closed-world-fresh work region r_H and a source materialization for the value being committed. Both are operational invariants of any well-formed runtime; we surface them explicitly because they are what the proof transports through.

Lemma 5.7 (Commit–load round-trip). *Assume* Open^τ. *Suppose:*

(1) CAM commit succeeds: $\mathrm{CAMCommit}(\tau, \langle r, i\rangle, S) = d$.
(2) There is a region r_H that is fresh in S (i.e., $r_H \notin \mathrm{dom}(S)$) and is also closed-world fresh: no region of S contains a **laptr** *token pointing into r_H.*
(3) The source materializes: $\langle r, i\rangle;\ S \ \Downarrow^\tau_{\mathrm{M}}\ V;\ i'$.

Let $\mathrm{CAMDeref}(\tau, d) = (r', S')$. *Then the loaded root is materializable:*

$$\langle r', 0\rangle;\ S' \ \Downarrow^\tau_{\mathrm{M}}\ W;\ i_W$$

for some W and i_W, and

$$S \cup S' \vdash V \equiv^\tau W.$$

The proof follows from the codec round trip (Lemma 5.5) together with the soundness lemma for the internal CAM-introduction judgment (Lemma 5.8), with one additional step: deep equality from Lemma 5.8 is stated in the work-region store S_H, and must be transported to $S \cup S'$. The transport is valid because the CAM-introduction heap is LAM-pointer-free (an invariant of the CAMCommit judgment, since by inspection of its rules the only way an **laptr** would enter the work buffer is via the CI-WrapSplice rule, which instead follows the wrapper to its target) and r_H is closed-world fresh (item 2 above); these together ensure that neither the deep-equality derivation nor the materializations it invokes ever read from r_H. The proof of Lemma 5.8 is an induction on the CAM-introduction derivation, parametric in the hash family (hashfn, combine) and chunking modulus κ: soundness asks only that the chunk-cut decision is a deterministic function of the hash, and is therefore independent of any probability model over the hash's outputs.

Lemma 5.8 (CAM introduction preserves deep equality). *Assume* Open^τ *and suppose the internal CAM-introduction judgment derives:*

$$\langle r, i\rangle;\ S;\ H_0 \ \Downarrow^\tau_W\ i';\ H;\ h,$$

with the starting work buffer Loc-free (no **laptr** *tags or location pointers in H_0). Let r_H be a work region fresh for S (i.e., $r_H \notin \mathrm{dom}(S)$), fresh in the additional closed-world sense that no region of S contains a* **laptr** *token pointing into r_H; and let $n = |H_0|$ and $S_H = S \cup \{r_H \mapsto H\}$. Then:*

(1) Materializability. *The suffix of H contributed by the derivation can be materialized:*

$$\langle r_H, n\rangle;\ S_H \ \Downarrow^{\tau}_{\mathrm{M}}\ V_H;\ |H|$$

for some value V_H.

(2) Source materializability. *The source address is materializable in S with the same end witness:*

$$\langle r, i\rangle;\ S \ \Downarrow^{\tau}_{\mathrm{M}}\ V;\ i'$$

for some value V.

(3) Deep equality in the work-region store. *Moreover,*

$$S_H \vdash V \equiv^{\tau} V_H.$$

This statement matches what the induction on the CAM-introduction derivation directly yields: deep equality in the *work-region store* S_H. The stronger fact that deep equality also holds in S alone follows by an independent transport step, which we use in the commit–load round trip (Lemma 5.7): because the CAM-introduction heap is LAM-pointer-free (by the same invariant of the CAMCommit rules noted in the discussion of Lemma 5.7), V_H contains no **laptr** or K^* sub-values, the deep-equality derivation never fires a wrapper rule on the V_H side, and the closed-world freshness of r_H in S ensures the V side never reaches r_H either. Thus the r_H-region content of S_H is dead weight and may be dropped.

The closed-world freshness condition on r_H (no **laptr** in S points into r_H) is an operational well-formedness invariant on the source store: location pointers are only ever created referencing previously allocated regions, so a freshly chosen r_H is automatically free of incoming pointers. We make it explicit because the transport step described above requires it.

6 Related work

Typed memory representations. The Gibbon project [13, 14, 23, 28, 29] computes directly on packed, serialized algebraic data. The published formalizations in this line cover progressively more of the real implementation: LoCal [28] gave an operational semantics for the fully inlined representation; the parallel extension [13] formalized limited indirections at thread boundaries; the GC integration [14] extended this to dense heaps with addressable sub-parts. In each case the proofs are pen-and-paper, and the mixed representation that real Gibbon runtimes use — wrappers, location pointers, and a GC that traverses them — is left outside the core calculus.

We extend this line on three fronts, each a first for this body of work. First, we give a formal account of automatic serialization as a runtime operation: adaptive linearization between mixed representations, with wrapper constructors, shallow duplication, and moving GC, generalizing parallel Gibbon's thread-boundary indirections. Second, we integrate the boundary discipline with content-addressable persistence (CAM), via content-defined chunking and typed commit/dereference operations that mediate between local heaps and a shared content-addressed store. Third, all the resulting metatheory is mechanized in Rocq and Lean.

Baudon et al. [1] develop compilation techniques for custom, type-driven memory representations of algebraic data types, supporting low-level encoding optimizations such as tag compression and bit stealing while preserving pattern-matching semantics. Rather than statically choosing a custom layout to exploit spare bits, we dynamically adjust layout via adaptive linearization, splicing and wrapping to trade locality against sharing as the heap evolves, and extend the same boundary discipline to content-addressable persistence.

Delaware et al. [7] derive correct-by-construction encoders and decoders from typed binary format specifications. While their work establishes correctness of translation between structured values and byte streams, we treat serialized layout as the runtime representation itself. Rather than

generating encoders and decoders around a fixed layout, we formalize a mixed representation in which values may be inline, location-addressed, or content-addressed, and prove that the runtime preserves the semantics of recursive data structures across these representations.

Parallels in systems and language runtimes. The pattern of introducing structural boundaries to manage cost and locality recurs across systems and language runtimes; in each case the boundary mechanism is hand-coded for the specific data structure or storage layer. B-trees [2] insert page-split boundaries when a node overflows, trading internal density for bounded fan-out. The Log-Structured File System [21] fixes per-segment boundaries up front and relies on a cleaner to reclaim fragmented space. Log-Structured Merge trees [18] (the foundation for LevelDB and RocksDB) compact at fixed level boundaries to amortize write amplification. Region-based memory management [24] statically bundles allocations into nested regions freed at scope exit.

Each of these mechanisms is specialized to one data structure or storage layer. Our contribution is generic: a single type-directed discipline—boundary datatypes parameterized by MaxDupSzT˙—supplies the analogous boundary mechanism uniformly for any algebraic data type, deriving the placement and cost from the type declaration rather than from per-structure encoding.

Content-addressed programming. Miller et al. [15] present $\lambda^\bullet$ (lambda-auth), a typed functional language with an authenticated type constructor $\bullet\tau$ for cryptographically verifiable values. Our **captr** τ type constructor is directly inspired by theirs, sharing the model of content-addressing as a first-class type feature rather than a library or external system. The two systems differ in goal and in granularity: lambda-auth targets authentication in untrusted-server settings, where verifying individual operations forces hashing at fine granularity ($O(n)$ hashes for n nodes in their internal-nodes-only scheme); we target deduplication and locality for persistent data structures in a trusted runtime, where content-defined chunking reduces overhead to $O(n/\kappa)$ for chunks of κ constructors (§5.2).

Content-addressable storage systems. IPFS [3, 6] is a distributed content-addressed storage system based on Merkle DAGs; it could serve as the external block store for our CAM layer. Git's object store [25], by contrast, names objects by cryptographic hash but uses fixed structural boundaries (one object per file or tree node), so a single-byte change in a large file produces an entirely new blob and destroys locality; our content-defined chunking gives fine-grained locality within large values while preserving the benefits of content addressing.

Content addressing is widely deployed at the byte or file level (Git, IPFS, blockchains [12, 17, 30]), but integration into a typed functional language has been attempted in only two prior systems: $\lambda^\bullet$ for authentication (discussed above) and Unison [26] for code identity. Neither addresses runtime data representation: deriving block boundaries from the type structure for incremental update with provable locality.

Content-defined chunking. Existing content-defined chunking (CDC) schemes operate on unstructured byte streams: chunks are arbitrary byte ranges that require parsing to recover higher-level structure. LBFS [16] introduced the technique for low-bandwidth file synchronization; FastCDC [31] is its modern performance-tuned successor; Chonkers [4] gives deterministic $O(\log^* n)$ worst-case locality bounds where traditional randomized CDC offers only expected-case guarantees. Our content-defined block decomposition adapts CDC to typed recursive structures, evaluating the boundary predicate at boundary-datatype constructors rather than at every byte; the resulting chunks are well-typed sub-values, directly usable without re-parsing, and our locality bounds are in expectation. Prolly Trees [22] apply CDC at a higher abstraction level, building probabilistically balanced B-trees over key-value stores via a rolling hash on keys: their boundaries fall between keys rather than between datatype constructors.

7 Conclusion

We have presented the first formalization of adaptive serialization for typed functional data, giving operational semantics and soundness proofs for a unified memory model with two realizations: location-addressable memory for local computation and content-addressable memory for persistence and distributed deduplication. The key abstraction in both models is the boundary datatype: in LAM, boundary constructors govern wrapper placement and bound shallow-duplication cost via $\mathrm{MaxDupSzT}^{\cdot}$; in CAM, the same constructors serve as potential chunk points for content-defined chunking, with locality guarantees tied to the same static measure. Our model treats CAM commit and dereference as observationally pure: committing a value and dereferencing its digest recovers a deeply equal value, regardless of how the digest interface is realized in practice. This lets content addressing fit into the typed functional setting as a value-level operation rather than an IO interface, while leaving an implementation free to back the digest with any storage layer it likes.

Limitations and future work. A datatype's classification as a boundary or alias type decides where wrapper constructors—and hence sharing points—may appear, and it bounds the cost of shallow duplication through $\mathrm{MaxDupSzT}^{\cdot}$. We classify a datatype as an alias when it has a single constructor and a statically bounded size. This criterion is a sound sufficient condition—every alias can be materialized inline with a finite, data-independent duplication cost—but it is conservative in two ways.

The first is a matter of cost. A wide alias type, such as a large tuple, has no internal boundary, so duplicating any structure that contains it must copy the type in full, and the constant in $\mathrm{MaxDupSzT}^{\cdot}$ grows with the type's width. A *threshold-based* policy would make a datatype a boundary type whenever $\mathrm{MaxDupSzT}^{\cdot}$ would exceed a fixed cap, bounding this per-update copy cost by a chosen constant—at the price of an extra constructor tag and read barrier.

The second is a matter of precision. The single-constructor requirement rejects multi-constructor types that are themselves cheap to materialize—for instance a sum of fixed-width payloads, whose $\mathrm{MaxSzT}^{\cdot}$ is bounded—forcing them to be boundary types that carry a wrapper they do not need. Admitting such types as aliases would remove the wrapper, at the cost of a data-dependent traversal: shallow duplication must read the constructor tag to determine the payload layout, and $\mathrm{MaxDupSzT}^{\cdot}$ rises from 2 to $1 + \max_K \mathrm{MaxDupSzT}^{\sigma_K}$ over the constructors $K\ \sigma_K$.

Both refinements preserve the type-directed framework, and we leave their development to future work. Conversely, some types with unbounded $\mathrm{MaxSzT}^{\cdot}$ may nevertheless admit only values of bounded size in practice; relaxing the static invariant to accommodate such cases is a natural extension. The CDC mechanism is parametric in the rolling-hash function; more sophisticated chunking schemes could be substituted without changing the type-directed framework. Finally, we leave the inclusion of mutability to future work.

The runtime model presented here is a specification that pins down what correct adaptive serialization and content-addressing should do, parameterized by the boundary discipline, the chunk-size cap, and the rolling hash family. Verifying that a concrete optimized runtime refines this specification is a separate, larger effort. A faithful refinement would have to track the optimizations that make the runtime practical: generational copying collection with a reference-counted old generation, root-set sorting for cache-coherent evacuation, bump-pointer allocation against contiguous nursery blocks. Empirical validation against a concrete runtime, measuring chunk-size distributions and re-traversal costs on workloads of interest, is a separately planned effort that we treat as orthogonal to the foundational results presented here.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-2115104 and CCF-2119352. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

We used large language models from Anthropic and OpenAI to assist with editing our prose, generating LaTeX for the formalism, and mechanizing the proofs. All ideas are our own, and the authors reviewed and take full responsibility for the accuracy and originality of the content.

Data-Availability Statement

The Rocq and Lean mechanizations supporting the results of this paper are archived on Zenodo [20]. The artifact contains the Rocq soundness development (LAM and CAM) and the Lean locality development, together with a self-contained virtual-machine image carrying the pinned toolchains (Rocq 9.0.1, Lean 4.27.0, and Mathlib4 v4.27.0) so the proofs can be re-checked entirely offline.

Appendix A Location-addressable memory

Appendix A.1 Typed Data (from § 3)

Definition A.1 (Well formedness of algebraic data types). *A finite collection Δ of declarations is* well formed *when:*

(1) all data-type identifiers in Δ are pairwise distinct;
(2) all constructor tags in Δ are pairwise distinct (we assume global uniqueness); and
(3) every data-type identifier appearing in any payload type τ_i is introduced by some declaration in Δ (mutual recursion is allowed).

Appendix A.2 Closed-World Store (from §4)

The following lemmas are used in the proof of Lemma 4.7, which establishes the GC evacuation invariant that the source and destination values are deeply equal.

Lemma A.2 (Materialization heap extension). *If $\langle r, i\rangle; S \Downarrow_{\mathrm{M}}^{\tau} V; i'$ and $S' = S[r \mapsto S[r] +\!\!+ H_{\mathrm{ext}}]$ (i.e., S' extends region r by appending tokens), then $\langle r, i\rangle; S' \Downarrow_{\mathrm{M}}^{\tau} V; i'$.*

Lemma A.3 (Store weakening for deep equality). *If $S_1 \vdash V \equiv^{\tau} W$ and $S_1 \subseteq S_2$, then $S_2 \vdash V \equiv^{\tau} W$.*

Definition A.4 (Forwarding soundness). *A forwarding map F equipped with a type assignment $\Phi : \mathrm{dom}(F) \to \mathrm{Type}$ is* sound for *(S_I, S_O) if for every $\langle r_S, i_S\rangle \mapsto \langle r_D, j_D\rangle \in F$ and every value V: if $\langle r_S, i_S\rangle; S_I \Downarrow_{\mathrm{M}}^{\Phi(\langle r_S, i_S\rangle)} V; i'$, then there exist W and j' such that $\langle r_D, j_D\rangle; S_O \Downarrow_{\mathrm{M}}^{\Phi(\langle r_S, i_S\rangle)} W; j'$ and $S_O \vdash V \equiv^{\Phi(\langle r_S, i_S\rangle)} W$.*

Appendix A.3 Rocq Mechanization

The metatheory of the closed-world store model (LAM) is mechanized in the Rocq prover (Coq 9.1.1) across two files totalling around 3,700 lines of code. In `Definitions.v`, we define all of the syntax and judgments of § 3 and § 4 including the additional syntactic forms and rules added in § 5. In `LAM.v` we machine-check that linearization and treeification are mutually inverse on a heap segment (Lemmas 3.4 and 3.5) and the size bound on linearization (Lemma 3.6). The statement of Lemma 3.6 only applies to wrapper-free inputs (the § 3 setting) because the size of a wrapped object (2 tokens) can be larger than MaxSzT^{τ} in the corner case of a simple boundary data type such as `bool`. We prove Lemmas 4.4-4.7 as well. The correctness of evacuation depends on helper lemmas allowing us to extend the heap of a materialization (Lemma A.2) and weaken the store in a deep equality judgment (Lemma A.3). All results are closed with `Qed`; the two files contain no `Admitted` goals and declare no axioms.

Three deliberate choices in the mechanization should be noted. *(i)* The content-pointer dereference CAMDeref occurs in materialization and deep equality rules, but we do not mechanize it as a stand-alone judgment; instead, we treat it as a meta-level function to prevent the LAM metatheory from depending on the CAM metatheory. Correspondingly, loc-freedom of CAM content, the property that chunk heaps contain no location tokens, is posited as a well-formedness hypothesis rather than derived as in § 5. *(ii)* The content-pointer rules of `DeepEquiv` recurse over the *ambient* store, instead of the explicit store union $S \cup S'$ or $S \cup S_1 \cup S_2$ used in the on-paper rules; the two formulations coincide under loc-freedom, and the simplification avoids complex store-restriction machinery the union form would require. *(iii)* The size metrics `maxSzT`/`maxDupSzT` are defined with an explicit fuel parameter and a visited-set guard to ensure structural termination over recursive datatype environments; their bounds are therefore stated relative to convergence of the fuel-bounded metric.

Appendix B Content-addressable memory

Appendix B.1 Byte codec and CAM round trips

This appendix presents the rules of the open-world byte codec introduced in §5.1, the assumptions on its primitive operations, and a summary of the Rocq mechanization of the round-trip lemmas of §5.4. The codec applies only to *open* types — those satisfying Open^{τ} (§5.1), which rules out bare location pointers in semantic values; its correctness lemmas play the same role for serialization that Lemma 4.7 plays for evacuation.

Primitive codec. The byte-level codec is abstracted as four encoder/decoder pairs, one per kind of primitive token that appears in encoded heaps: unsigned integer values, datatype tags, content digests, and 64-bit byte offsets. We treat these eight functions as opaque parameters; the proofs assume only three properties of each pair. *Fixed width* (eight bytes for integers, tags, and offsets; thirty-two for digests): every encoder produces a byte string of a declared, type-dependent length. *Round trip*: decoding immediately after encoding recovers the token, that is, $\mathrm{dec}(U +\!\!+ \mathrm{enc}(x) +\!\!+ E,\ |U|) = x$ for any prefix U and suffix E, so a decoder depends only on the bytes within its own token. *Locality*: extending the buffer beyond the read window does not change the decoded value, $\mathrm{dec}(U +\!\!+ E,\ i_B) = \mathrm{dec}(U,\ i_B)$ whenever $i_B + \mathit{width} \le |U|$. No further property of the codec is assumed.

Encode and decode. The encode and decode relations are mutually inductive judgments layered above the primitive codec. Encode reads typed tokens from the source store at $\langle r, i\rangle$ and appends bytes to a buffer; decode reads bytes at an offset and appends typed tokens to a fresh heap. Both are type-directed (one rule per type form; auxiliary mutual relations recurse over tuple-field lists). Three load-bearing details: variable-width tuples carry a leading byte-cursor header — one slot per variable-width field — so decode reconstructs field positions without knowing field widths *a priori*; encode at a boundary data type may follow a LAM wrapper transparently to its target via a dedicated WRAP-SPLICE rule, so the encoded bytes are the same regardless of how the source is shared; and no decode rule produces an **laptr** token, so any decoded segment is closed-world by construction (matching the decoded-heap loc-freedom clause of Assumption 5.6).

The three correctness lemmas (Lemmas 5.5, 5.8, and 5.7) are mechanized in Rocq. Each Rocq theorem matches the corresponding lemma above modulo notation: same hypotheses, same conclusions. The proof graph rooted at these three theorems reaches roughly two dozen helper lemmas, with a maximum chain depth of five. A handful of additional helpers — store weakening, materialization determinism, and the mutual-induction principles for the materialization and deep-equality judgments — are shared with the companion LAM mechanization. No goals are admitted.

Four families of axioms suffice. The primitive codec contributes its four encoder/decoder pairs with the three properties above. The external content-addressable store contributes Assumption 5.6 (get/put round trip plus decoded-heap loc-freedom). Two CAM-specific operational invariants are declared as parameters: the canonical chunk heap for a digest equals the bytes committed under that digest, and the commit operation is functional in the digest output. Finally, the hash family (combine, hashfn, modulus κ, per-chunk cap C) appears as opaque parameters: the soundness story above is parametric in the hash and uses no probability theory; the probabilistic content is mechanized separately in Lean and summarized in §B.2 below.

Appendix B.2 Locality of chunk decomposition

This subsection formalizes the path-edit notion and hash-independence assumption that Theorem 5.2 (§5.3) rests on, states the per-side and per-sibling building-block lemmas the theorem

composes, records the deterministic complement to its in-expectation bounds, and closes with a summary of the Lean mechanization.

Definition B.1 (Path edit). *A* path edit of depth d *transforms a value V of type τ in store S into a value V' in $S' \supseteq S$ by rebuilding the d boundary-datatype constructors on a single path from the root of V to the edit point. Every substructure of V not on this path is referenced from V' via a location pointer into S, forming an* unchanged sibling subtree. *For a tree of branching factor b the number of unchanged siblings is $s = (b-1)d$.*

Although the theorem statement is phrased in terms of V and V', the values themselves do not enter the probability calculation. By content-defined purity (Proposition 5.2), shared PCPs produce identical chunks on both sides of the edit and drop out of the symmetric difference; only the rolling-hash residues at the d rebuilt PCPs do. The random data the analysis quantifies over is therefore a finite tuple of d values in $\{0, \ldots, \kappa-1\}$, drawn independently and uniformly; the two-side comparison in clause (i) uses two such tuples drawn independently of each other. Both the lemmas below and the Lean mechanization work at this level: the proofs do not construct V or V', and the mechanization adopts the i.i.d. uniform model directly — uniform marginals with mutual independence, which is exactly Assumption B.2 — rather than carrying it as a separate hypothesis.

Assumption B.2 (Hash independence). *For every PCP p, the boundary event $h(p) \bmod \kappa = 0$ occurs with probability $1/\kappa$, and the boundary events at distinct PCPs are mutually independent.*

Justifying Assumption B.2. The assumption is parametric in combine and is realistic at several levels of rigor, which we match to the results that consume it. The boundary-count bounds (Lemma B.3 and clause (i) of Theorem 5.2) use only the $1/\kappa$ marginal, by linearity of expectation, so Carter–Wegman 2-universal families and tabulation hashing [19] discharge that much outright as a theorem. The geometric argument behind the $\kappa - 1$ spillover witness (Lemma B.4 and clauses (ii)–(iii)) consumes the mutual-independence clause in full; cryptographic hashes (SHA-256, BLAKE3) satisfy it under the random oracle model. Widely deployed non-cryptographic combiners (xxHash3, MurmurHash3, the Gear hash of FastCDC [31]) pass the SMHasher suite [27], an empirical certification of the avalanche and uniformity properties the assumption requires. A deployment selects the regime its threat model demands.

Lemma B.3 (Hash-boundary count on an edit path). *Under Assumption B.2, the expected number of hash-triggered boundary cuts among the d rebuilt PCPs of a single edit-path side is at most d/κ; the expected number of chunks containing rebuilt PCPs is therefore at most $d/\kappa + 1$ per side.*

Lemma B.4 (Spillover bound). *Under Assumption B.2, with chunk-size cap $C = c\kappa$ for $c \geq 1$, the spillover X_R^C from an unchanged sibling subtree R of type τ_R — the number of constructors traversed in R before the first cut node (hash- or cap-triggered) — satisfies*

$$\mathbb{E}[X_R^C] \;\leq\; \min(C,\; \kappa - 1,\; h_{\tau_R}),$$

where h_{τ_R} is the maximum leftmost-path PCP count over values of type τ_R: a static constant under Open^{τ_R}*, since any recursive or oversized sub-type appears only behind a* **captr** *wrapper.*

Deterministic complement. Two unconditional facts accompany the expected bounds above. First, the chunk-size cap fires whenever the post-order residual at a PCP would exceed C, so every chunk contains at most C tokens in every execution, regardless of the hash output. Second, the symmetric-difference block count of Theorem 5.2 is always between 2 and $2d$: at most d block boundaries can fall among the d rebuilt PCPs in any execution, so an adversary controlling combine cannot inflate the affected-block count beyond $O(d)$ without performing a deeper edit.

Mechanization. Lemmas B.3 and B.4, together with all three clauses of Theorem 5.2, are mechanized in Lean over Mathlib4's probability and integration libraries; no probability argument in the proofs is left informal.

Lemma B.4's bound is established as the minimum of three independent witnesses — a deterministic cap, a geometric argument on the rolling-hash residues, and a type-determined structural bound — combined in a one-line min. The three clauses of Theorem 5.2 reduce to these building-block lemmas by linearity of expectation:

- *Clause (i)*, chunk count, doubles the per-side bound of Lemma B.3 (one copy per side of the edit, the two sides drawn independently) and adds the deterministic cap-triggered contribution.
- *Clause (ii)*, expected spillover, sums the per-sibling bound of Lemma B.4 across the s unchanged sibling subtrees.
- *Clause (iii)*, total affected constructors, adds the d rebuilt constructors on the edit path to the clause-(ii) bound.

The static measure h_{τ_R} is encoded as a partial inductive predicate over types whose constructors mirror the leftmost-path case analysis (constants for primitives, the first-field rule for tuples, transparent aliases, and 1 + max over constructor payloads for non-alias data types). Partiality is deliberate: a derivation exists exactly when h_{τ_R} is finite, so the Open^{τ} side condition becomes "a derivation exists" rather than a separate well-formedness predicate.

Two items live outside the Lean tree. Cap-purity (Proposition 5.2) is a structural-induction result mechanized on the Rocq side; the Lean development holds the digest computation opaque precisely because the locality bounds do not depend on it. Corollary 5.3 (the balanced-binary-tree specialization) is left as a short calculation from the mechanized bounds.

References

[1] Thaïs Baudon, Gabriel Radanne, and Laure Gonnord. 2023. Bit-Stealing Made Legal: Compilation for Custom Memory Representations of Algebraic Data Types. *Proc. ACM Program. Lang.* 7, ICFP (2023). doi:10.1145/3607858

[2] R. Bayer and E. McCreight. 1972. Organization and maintenance of large ordered indexes. *Acta Informatica* 1, 3 (1972), 173–189. doi:10.1007/BF00288683 Introduces the B-tree, including the page-split mechanism that introduces a structural boundary when a node overflows..

[3] Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. doi:10.48550/arXiv.1407.3561 arXiv:1407.3561 [cs.NI]

[4] Benjamin Berger. 2025. The Chonkers Algorithm: Content-Defined Chunking with Provable Strict Guarantees on Size and Locality. doi:10.48550/arXiv.2509.11121 arXiv:2509.11121 [cs.DS]

[5] Daniele Bonetta, Júnior Löff, Matteo Basso, and Walter Binder. 2025. HeapBuffers: Why Not Just Using a Binary Serialization Format for Your Managed Memory? *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 397 (Oct. 2025), 25 pages. doi:10.1145/3763175

[6] Erik Daniel and Florian Tschorsch. 2022. IPFS and Friends: A Qualitative Comparison of Next Generation Peer-to-Peer Data Networks. *IEEE Communications Surveys & Tutorials* 24, 1 (2022), 31–52. doi:10.1109/COMST.2022.3143147

[7] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: Correct-by-Construction Derivation of Decoders and Encoders from Binary Formats. *Proc. ACM Program. Lang.* 3, ICFP, Article 82 (2019), 29 pages. doi:10.1145/3341686

[8] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. 2004. Nix: A Safe and Policy-Free System for Software Deployment. In *Proceedings of the 18th USENIX Conference on System Administration (LISA '04)*. USENIX Association, 79–92.

[9] IPFS Standards. 2024. Bitswap Protocol Specification. https://specs.ipfs.tech/bitswap-protocol/. Specifies that compliant implementations must support blocks up to 2 MiB, with larger blocks discouraged for interoperability..

[10] IPFS Standards. 2024. UnixFS Specification. https://specs.ipfs.tech/unixfs/. Recommends 256 KiB (legacy) or 1 MiB (modern) as the default chunk size for newly created blocks..

[11] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. 2021. A Hardware Accelerator for Protocol Buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 462–478. doi:10.1145/3466752.3480051

[12] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In *Advances in Cryptology – CRYPTO 2017 (Lecture Notes in Computer Science, Vol. 10401)*. Springer, 357–388. doi:10.1007/978-3-319-63688-7_12

[13] Chaitanya Koparkar, Mike Rainey, Michael Vollmer, Milind Kulkarni, and Ryan R. Newton. 2021. Efficient tree-traversals: reconciling parallelism and dense data representations. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. doi:10.1145/3473596

[14] Chaitanya S. Koparkar, Vidush Singhal, Aditya Gupta, Mike Rainey, Michael Vollmer, Artem Pelenitsyn, Sam Tobin-Hochstadt, Milind Kulkarni, and Ryan R. Newton. 2024. Garbage Collection for Mostly Serialized Heaps. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management* (Copenhagen, Denmark) *(ISMM 2024)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3652024.3665512

[15] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. 2014. Authenticated data structures, generically. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 411–423. doi:10.1145/2535838.2535851

[16] Athicha Muthitacharoen, Benjie Chen, and David Mazières. 2001. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*. ACM, Banff, Alberta, Canada, 174–187. doi:10.1145/502034.502052 Introduced content-defined chunking using Rabin fingerprinting for efficient file synchronization over low-bandwidth networks..

[17] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf

[18] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385. doi:10.1007/s002360050048 Original LSM-tree design with level-based compaction; foundation for LevelDB and RocksDB..

[19] Mihai Pătraşcu and Mikkel Thorup. 2012. The Power of Simple Tabulation Hashing. *J. ACM* 59, 3 (2012), 14:1–14:50. doi:10.1145/2220357.2220361 Shows that simple tabulation hashing is pairwise (in fact 3-wise) independent, providing provable random-like behavior that suffices for many hashing-based analyses without cryptographic assumptions..

[20] Michael Rainey, Michael H. Borkowski, Michael Vollmer, Chaitanya S. Koparkar, Mikah Kainen, and Vidush Singhal. 2026. LoCalMem: Type-Directed Adaptive Serialization for Location- and Content-Addressable Memory (ICFP 2026 Artifact). doi:10.5281/zenodo.20315616

[21] Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems* 10, 1 (1992), 26–52. doi:10.1145/146941.146943 The Log-Structured File System (LFS): treats the disk as a sequential log, fixing per-segment boundaries up front and relying on a cleaner to reclaim space..

[22] Tim Sehn. 2024. *Prolly Trees.* https://www.dolthub.com/blog/2024-03-03-prolly-trees/ Blog post, DoltHub, Inc..

[23] Vidush Singhal, Chaitanya Koparkar, Joseph Zullo, Artem Pelenitsyn, Michael Vollmer, Mike Rainey, Ryan Newton, and Milind Kulkarni. 2024. Optimizing Layout of Recursive Datatypes with Marmoset: Or, Algorithms + Data Layouts = Efficient Programs. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 38:1–38:28. doi:10.4230/LIPIcs.ECOOP.2024.38

[24] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and Computation* 132, 2 (1997), 109–176. doi:10.1006/inco.1996.2613 Region calculus for stack-discipline memory management: static analysis bundles allocations into nested regions freed at scope exit..

[25] Linus Torvalds and Junio Hamano. 2005. Git: Fast Version Control System. https://git-scm.com

[26] Unison. 2024. Unison. https://www.unison-lang.org/

[27] Reini Urban, Austin Appleby, and contributors. 2024. SMHasher: a test suite for hash functions. https://github.com/rurban/smhasher. Empirical test suite covering avalanche behavior, collision rates, output distribution, and statistical independence for non-cryptographic hash functions. Originally created by Austin Appleby and maintained by Reini Urban..

[28] Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. 2019. LoCal: a language for programs operating on serialized data. (2019), 48–62. doi:10.1145/3314221.3314631

[29] Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan Newton. 2017. Compiling Tree Transforms to Operate on Packed Representations. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 26:1–26:29. doi:10.4230/LIPICS.ECOOP.2017.26

[30] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Platform. https://ethereum.github.io/yellowpaper/paper.pdf Yellow Paper.

[31] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. 2016. FastCDC: A Fast and Efficient Content-Defined Chunking Approach for Data Deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC '16)*. USENIX Association, Denver, CO, 101–114. https://www.usenix.org/conference/atc16/technical-sessions/presentation/xia Achieves 3-10x speedup over Rabin fingerprinting using Gear hash and fast normalization techniques..