What if serialized data were the heap?

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

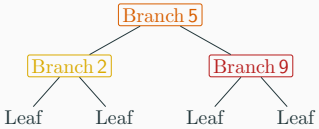
Michael Rainey[1] Michael H. Borkowski[2] Michael Vollmer[3]
Chaitanya S. Koparkar[4] Mikah Kainen[2] Vidush Singhal[2]

[1]Carnegie Mellon University [2]Purdue University [3]University of Kent [4]The MathWorks

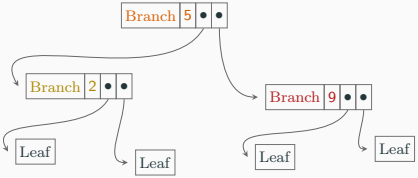
ICFP 2026 • Indianapolis • August 2026

A conventional heap makes us pay three times

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



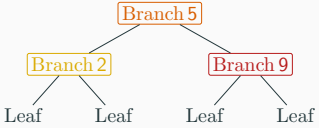
the value



what conventional runtimes build

- Metadata: pointers and headers can outweigh the payload.
- Traversal: the CPU follows links, not a contiguous value.
- Boundaries: crossing a process boundary means *encode*, then *decode*.

Heap form is wire form: crossing is just a copy



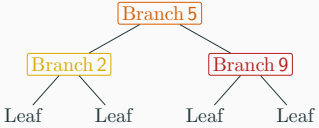
the value

Branch	8	5	Branch	7	2	Leaf	Leaf	Branch	12	9	Leaf	Leaf

the same value, contiguous in memory

$\longrightarrow$ traversal proceeds left to right

Heap form is wire form: crossing is just a copy



the value

Branch	8	5	Branch	7	2	Leaf	Leaf	Branch	12	9	Leaf	Leaf

the same value, contiguous in memory

$\longrightarrow$ traversal proceeds left to right

plain copy

Branch	8	5	Branch	7	2	Leaf	Leaf	Branch	12	9	Leaf	Leaf

$\rightsquigarrow$

Branch	8	5	Branch	7	2	Leaf	Leaf	Branch	12	9	Leaf	Leaf

network, disk, `mmap`

The heap bytes are the wire bytes.

But can a flat heap share?

Sharing prevents a completely flat heap

Build one new node above an existing value:

$$\mathrm{Branch}\big(5,\ \underbrace{t}_{\text{one billion nodes}},\ \mathrm{Leaf}()\big)$$

Inline t

Copy one billion constructors

$\Theta(n)$

Share t

Store one pointer

$O(1)$

Dense *except* where sharing demands a pointer:

mostly serialized memory

serialized bytes as the native heap, with selected indirections

The type decides where sharing may interrupt the stream

A decade of Gibbon built the compiler, representation, and collector.

What was missing: a specification for the adaptive serialization underneath.

boundary datatype

a type where the runtime may choose an inline value or a pointer wrapper

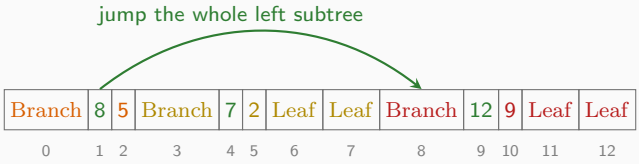
$$\mathrm{Branch}\big(\textbf{uint} \times \mathrm{itree} \times \mathrm{itree}\big)$$

At an itree boundary: keep the subtree inline, or store $\mathrm{Ptr}_{\mathrm{itree}}$.

We give this choice a precise semantics and a machine-checked metatheory.

The format stores only what the type cannot know

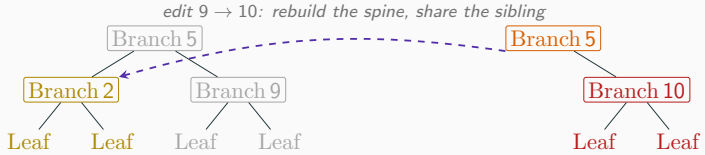
$$\textbf{data}\ \text{itree} \;=\; \text{Leaf}\ () \;\mid\; \text{Branch}\ (\textbf{uint} \boxtimes \text{itree} \times \text{itree})$$



tag	*type*	skip index
which constructor	fields in order	where to jump

A pointer is an optional constructor

Sharing needs a pointer. Where can it live without making every itree indirect?

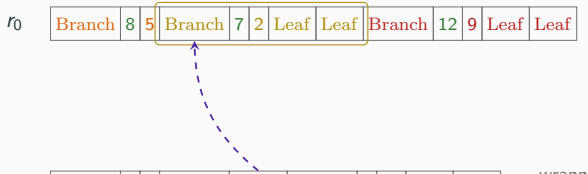


Not a pointer *type*: that commits every value to indirection. Instead, extend the datatype:

$$\textbf{data}\ \text{itree} \;=\; \text{Leaf}\ () \;\mid\; \text{Branch}\ (\textbf{uint} \boxtimes \text{itree} \times \text{itree}) \;\mid\; \text{Ptr}_{\text{itree}}\ (\textbf{ptr}\ \text{itree})$$

A pointer is an optional constructor

Sharing needs a pointer. Where can it live without making every itree indirect?



wrapper = tag + pointer:
a value-level boundary, undoable later

data itree $=$ Leaf () $\mid$ Branch (**uint** $\boxtimes$ itree $\times$ itree) $\mid$ $\text{Ptr}_{\text{itree}}$ (**ptr** itree)

Two tokens buy sharing — a tag and a pointer — and the dense run becomes fragmented

Both wrapper extremes fail

Wrapper placement is a policy. Try the extremes.

Wrap every type

Every value gains an alternative Ptr constructor. One-constructor records lose their free tag; every read checks for indirection.

The pointer heap, rebuilt.

Wrap no type

Sharing copies until the next boundary — but there is none. A list-tail share changes from $O(1)$ to $O(n)$.

Asymptotics lost.

Sharing cost = distance to the nearest boundary.

A nearby boundary keeps sharing cheap

Assign boundaries so every unfolding reaches one within a bounded number of steps.

boundary $\mathrm{itree} \rightsquigarrow \mathrm{Branch}\,(\mathbf{uint} \times \mathrm{itree}^{\bullet} \times \mathrm{itree}^{\bullet})$

every path meets a boundary ($\bullet$) in one step — a wrapper always fits

if it were not $\mathrm{ilist} \rightsquigarrow \mathrm{Cons}\,(\mathbf{uint} \times \mathrm{ilist}) \rightsquigarrow \mathrm{Cons}\,(\mathbf{uint} \times \mathrm{Cons}\,(\mathbf{uint} \times \mathrm{ilist})) \rightsquigarrow \cdots$

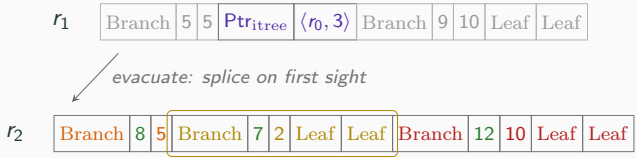
the Cons spine unfolds forever — no boundary, ever, so nothing on it may be shared

Our placement rule marks one-constructor, bounded-size types as boundaries.

Every shallow copy is statically bounded by MaxDupSzT(τ).

Type knowledge ends: size → skip index • constructor → tag • sharing → wrapper.

The collector repairs fragmentation



First sight

Follow the wrapper and splice the subtree inline.

Later sights

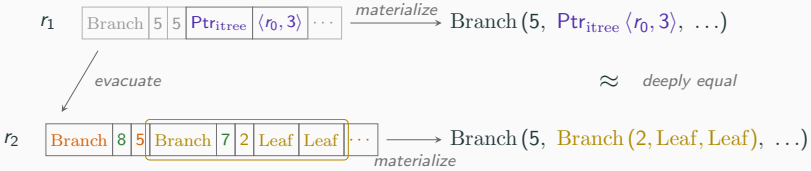
Keep the wrapper, so the collector preserves sharing.

Evacuation re-serializes while it copies.

The dense layout is reclaimed: adaptive serialization.

Evacuation preserves the value, not the representation

A wrong splice can still look like a valid heap.



Materialize

Observe what an address contains; leave wrappers visible.

Deep equality

Ignore inline vs. wrapped and physical location.

The square commutes: evacuation is the identity up to representation.

Proved in Rocq, with no admitted goals.

The representation travels; local pointers do not

Dense heap bytes cross by plain copy.

Local pointer

$$\textbf{ptr}\ \tau \ni \langle r, i \rangle$$

Offset i in one store r

Meaningless elsewhere

Content pointer

$$\textbf{captr}\ \tau \ni \langle d, i \rangle$$

Digest $d = H(U)$; offset i in U

Stable anywhere

One discipline, two memories

LAM for what lives in this process • CAM for what outlives it

Content names make equality cheap — but every digest costs

same value here $\longrightarrow$ canonical bytes $\longrightarrow H \longrightarrow d$

same value elsewhere $\longrightarrow$ canonical bytes $\longrightarrow H \longrightarrow d$

One digest for the value

Small metadata

An edit rehashes $O(n)$ constructors

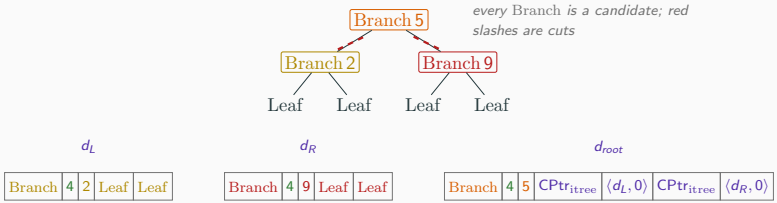
One digest per node

Local rehashing

But $O(n)$ digests

Where may it cut — and which candidates fire?

Boundary types supply candidates; the policy decides

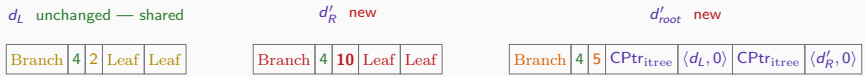


data itree $=$ Leaf () $\mid$ Branch $(\cdots)$ $\mid$ Ptr$_{\text{itree}}$ (**ptr** itree) $\mid$ CPtr$_{\text{itree}}$ (**captr** itree)

One analyzed policy: at a candidate, cut if $h \bmod \kappa = 0$ or residual $> C$.

Same subvalue $\Longrightarrow$ same cuts $\Longrightarrow$ same chunks.

A small edit changes $O(1)$ chunks in expectation



affected = chunks on the edit path; everything else deduplicates

Changed chunks

Store and hash their new content.

Spillover

Untouched constructors re-traversed because they share a chunk with the edit path.

Balanced tree, $\kappa = C = \log n$:

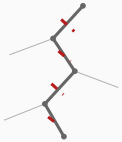
$O(1)$ expected chunks $\quad O(\log^2 n)$ expected constructors

At $n = 2^{20}$: expected changed chunks ≤ 5; affected constructors ≤ 400.

Proved and mechanized in Lean 4 + Mathlib.

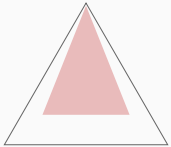
Hash and cap cover different worst cases

no hash



cap-cuts follow tree shape: up to $\Theta(d)$ chunks on an adversarial path

no cap



hash gaps can leave a supercritical residual: $\Theta(n^{0.93})$ at κ=20, n=10^6

Together: deterministic size + expected locality.

The cap bounds every chunk; the hash spreads cuts independently of tree shape in expectation.

Serialized data can be the heap — if the type controls indirection

Local

$\tau \mid \mathrm{Ptr}_\tau \langle r, i \rangle$

Cheap structural sharing

Collection restores density

Global

$\tau \mid \mathrm{CPtr}_\tau \langle d, i \rangle$

Typed, automatic chunking

Locality proved and mechanized

One discipline, two memories.

Proofs mechanized in Rocq and Lean; evaluated artifact available.

Backup: Gibbon, in four papers

Gibbon compiles a strict, purely functional Haskell subset to C — whole-program.

ECOOP'17	the compiler
PLDI'19	LoCal: the *intermediate representation*, formalized
ICFP'21	parallelism
ISMM'24	a generational collector

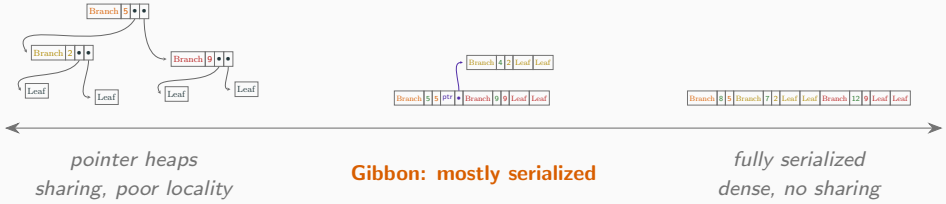
forwarding pointers too small for a slot; root-set sorting; half-written objects

What none of them pinned down: what the runtime does to *keep* the layout dense — wrapping, splicing, duplicating, evacuating.

Backup: why types, not a program analysis?

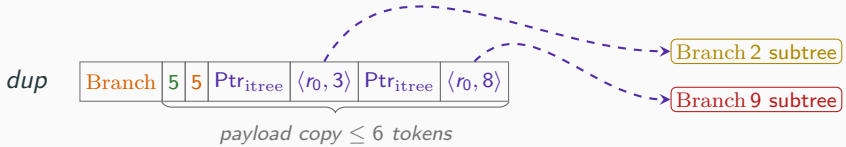
An analysis would have to prove, for every call site, which layout each value has — and stay sound under separate compilation and higher-order code. The type already carries that information, and the runtime needs it *at run time* to decode at all: an operational contract, not a static overlay.

Backup: the spectrum



Dense *except* where sharing demands a pointer. The right end is usually an *interchange* story (fixed byte layouts as a contract with the outside world); Gibbon makes the dense form the native heap representation.

Backup: shallow duplication, priced



$$\text{MaxDupSzT}((\textbf{uint} \boxtimes \text{itree} \times \text{itree})) = \underbrace{1}_{\text{skip}} + \underbrace{1}_{\text{label}} + \underbrace{2}_{\text{itree}} + \underbrace{2}_{\text{itree}} = 6$$

Each $\times$ costs one skip index; each boundary field is two tokens (a wrapper).

$\text{MaxDupSzT}(\text{itree}) = 2$.

Backup: content addressing as a type

$\lambda^\bullet$ (Miller et al.) introduced the authenticated type constructor $\bullet\tau$; our **captr** τ is its direct descendant. The remaining question is where to put that constructor:

$\mathrm{Branch}\ (\mathbf{uint} \boxtimes \mathbf{captr}\ \mathrm{itree} \times \mathbf{captr}\ \mathrm{itree})$	(a) digest at every node
$\mathrm{Branch}\ (\mathbf{uint} \boxtimes \mathbf{captr}\ (\mathrm{itree} \times \mathrm{itree}))$	(b) one digest per branch
$\mathrm{Branch}\ (\ldots) \mid \mathrm{Branch}_{\mathrm{ca}}\ (\ldots \mathbf{captr} \ldots)$	(c) manual chunking

(a) is $\lambda^\bullet$'s regime ($O(n)$ digests, forced by authentication); (b)/(c) bake the grain into the datatype — changing it rewrites the type and every function over it.

Backup: full locality bounds

For an edit at depth d, with s sibling subtrees on the rebuilt path and target chunk size κ:

$$\mathbb{E}[\text{changed chunks}] = O(d/\kappa + 1) \qquad \mathbb{E}[\text{affected constructors}] \leq d + s(\kappa - 1).$$

	digest space	work per balanced-tree edit
Whole-value hashing	1	$O(n)$
$\lambda^\bullet$	$O(n)$	$O(\log n)$
LoCalMem	$O(n/\log n)$	$O(\log^2 n)$

A logarithmic factor of update time buys a $\log n$-fold reduction in digest space.

Backup: which hash functions satisfy the assumption?

Two roles, two requirements. Locality needs mixing; four regimes: proved (tabulation hashing — discharges the assumption as a theorem), random-oracle (SHA-256, BLAKE3), SMHasher-implied (Gear, FNV-1a, CityHash), assumed-only (ad-hoc combiners). Identity — equal names imply equal values — needs collision resistance: cryptographic digests, the same standard assumption Git and IPFS already make.

Backup: what would you build with CAM?

Honestly: the paper claims a mechanism, not a killer app; no single canonical example yet. The natural first deployments:

- **Versioned values** — Git-for-data-structures: structural dedup across versions, integrity for free.
- **Shared structured state** across services or replicas: equal names mean nothing is re-transferred or re-checked.
- **Structural memoization**: the content name is the cache key.

Versus PGAS, pointer swizzling, or wholesale heap copy: those keep *location* names alive across machines; a content name survives the process and deduplicates. And dereference is a key–value get on an exact digest — naming values by content does not make lookup a search problem.

Backup: why Rocq and Lean — and why not Iris?

The split follows the mathematics. Rocq holds the core metatheory: the typed store, wrapping/splicing as inverses, the GC specification, commit/deref soundness. The Lean 4 half holds the CAM time/space bounds, because Mathlib supplies the expectation machinery the locality theorem needs.

Why not separation logic: the model is sequential and first-order — no ownership transfer, no concurrency — so plain operational semantics and induction carry every proof. Iris would buy generality this model does not yet spend.

Backup: is κ free?

No: locality saturates around $\kappa \approx \log n$; digest amortization bounds it below; and in any distributed deployment κ is a protocol constant — changing it invalidates every content identifier.

Backup: what “region” means here

Not Tofte–Talpin. There is no `letregion`, no lexical scope, and no deallocation at scope exit. In the model a store maps region names to heaps, and a location pointer $\langle r, i \rangle$ names offset i in region r; pointers cross regions freely, and that is exactly what makes sharing possible. In Gibbon they are growable buffers of unrestricted lifetime, kept alive by their incoming pointers — reference counted in PLDI’19, collected in ISMM’24. A region is where bump allocation happens, not a lifetime discipline.

Backup: LAM + CAM — why one paper?

Same boundary positions govern wrapper placement and chunk placement; the same measure MaxDupSzT$(\cdot)$ bounds duplication cost and chunk spacing. The unification emerged from analyzing the two together.

Backup: how close is this to real Gibbon?

Specification-first: we distill Gibbon's layout and adaptive serialization into a model and prove *it* sound. Verifying a concrete optimized runtime against this specification is future work.

Backup: artifact and next steps

Functional & Available

Zenodo `10.5281/zenodo.20315616`

Every proof re-checks offline in approximately 22 minutes.

Next

Verify a concrete Gibbon-style runtime against the specification.

Validate the mechanisms empirically.