LoCal: A Language for Programs Operating on Serialized Data*

Extended version

Michael Vollmer
Computer Science
Indiana University
Bloomington, Indiana, United States
vollmerm@indiana.edu

Chaitanya Koparkar
Computer Science
Indiana University
Bloomington, Indiana, United States
ckoparka@indiana.edu

Mike Rainey
Computer Science
Indiana University
Bloomington, Indiana, United States
me@mike-rainey.site

Laith Sakka
Electrical and Computer Engineering
Purdue University
West Lafayette, Indiana, United States
lsakka@purdue.edu

Milind Kulkarni
Electrical and Computer Engineering
Purdue University
West Lafayette, Indiana, United States
milind@purdue.edu

Ryan R. Newton
Computer Science
Indiana University
Bloomington, Indiana, United States
rrnewton@indiana.edu

Abstract

In a typical data-processing program, the representation of data *in memory* is distinct from its representation in a *serialized* form on disk. The former has pointers and arbitrary, sparse layout, facilitating easy manipulation by a program, while the latter is packed contiguously, facilitating easy I/O. We propose a language, LoCal, to unify in-memory and serialized formats. LoCal extends a region calculus into a *location calculus*, employing a type system that tracks the byte-addressed layout of all heap values. We formalize LoCal and prove type safety, and show how LoCal programs can be inferred from unannotated source terms.

We transform the existing Gibbon compiler to use LoCal as an *intermediate language*, with the goal of achieving a balance between code speed and data compactness by introducing *just enough* indirection into heap layouts, preserving the asymptotic complexity of traditional representations, but working with mostly or completely serialized data. We show that our approach yields significant performance improvement over prior approaches to operating on packed data, without abandoning idiomatic programming with recursive functions.

Keywords Region Calculus, Compiler Optimization, Data Encoding, Tree Traversal

1 Introduction

Virtually all programs running today use heap object representations *fixed* by the language runtime system. For instance, the Java or Haskell runtimes dictate an object layout, and the compiler must stick to it for all programs. And yet when humans optimize a program, one of their primary *levers on performance* is changing data representation. For example, an HPC programmer knows how to pack a regular tree into a byte array for more efficient access [8, 14, 16].

Whenever a program receives data from the network or disk, rigid insistence on a particular heap layout causes an impedance mismatch we know as *deserialization*. Yet, the alternative would seem to be writing low-level code to deal directly with specialized or serialized data layouts. This is error-prone, making it a "hacky" way to achieve performance optimization at the expense of safety and readability.

To ameliorate this tension we propose to reify *data layout* as an explicit part of the program. We introduce a language, LoCal (which stands for *location calculus*), whose type system directly encodes a byte-level layout for algebraic datatypes manipulated by the language. A well-typed program consists of functions, data definitions, *and* data representation choices, which can then be tailored to an application. This means that programs can operate over *densely* encoded (serialized) data in a type-safe way.

If data resides on disk in a LoCal-compatible format, it becomes possible to *bring the program to the data* rather than the traditional approach of bending the data to the code: deserializing it to match the rigid heap format of the language runtime. This effort contrasts with earlier work on persistent languages [2, 13] and object databases [10], which sought to expand the mutable heap to encompass disk as well memory, translating (swizzling) between persistent pointers and in-memory

*Extended version of "LoCal: A Language for Programs Operating on Serialized Data," Vollmer et al., PLDI 2019[29].

pointers. Instead, the emphasis here is on processing immutable data, and eschewing pointers entirely wherever possible.

The layout of a LoCal data constructor by default takes only one (unaligned) byte in memory and fields may be referred to *either* by pointer indirections or unboxed into the parent object (serialized). We can thus interpolate between fully serialized and fully pointer-based representations. LoCal can thus serve as a flexible intermediate representation for compilers or synthesis tools.

This paper makes four contributions:

- We introduce LoCal, the first formal language where well-typed terms prescribe the byte-addressed data-layout of the recursive datatypes they manipulate (§3). We formalize the core of the language and prove type safety (progress and preservation).
- We present an implementation strategy and compiler for LoCal (§4). By judicious use of indirections, it represents the first technique for compiling recursive functions on (mostly) serialized data which is *work efficient*, not compromising asymptotic complexity compared to a traditional language implementation.
- We present a strategy for synthesizing LoCal programs from a first-order, purely-functional input language, *HiCal* (the front-end for our Gibbon compiler), and redesign our compiler around LoCal (§5).
- We evaluate our compiler pipeline against approaches for working with serialized binary data, including *Compact Normal Form* [33], Cap'N Proto, and prior Gibbon [30] (§7). Our pipeline achieves 3.2×, 9.4×, and 202× geomean speedups respectively (Table 2), including asymptotic advantages, and substantial speedups in IO-intensive experiments.

2 Background

Consider a simple tree data structure, written in a language that supports algebraic datatypes:

```
data Tree = Leaf Int | Node Tree Tree
```

In memory, each node in this tree is either a `Leaf` node, typically consisting of a header word (denoting that it is a `Leaf`) and another word holding the integer data, or an interior `Node`, consisting of a header word and *two* double words (on a 64-bit system) holding pointers to its children. A tree with 2 internal nodes and 3 leaf nodes, then, occupies 64 bytes of space (20 bytes per internal node and 8 bytes per leaf node), even though it contains only 12 bytes of "useful" data. Storing the pointers that maintain the internal structure of the tree represents a significant storage overhead.

```
int sumPacked (byte * &ptr) {
  int ret = 0
  if (* ptr == LEAF) {
    ptr++; //skip past tag
    ret = * (int*)ptr; //retrieve integer
    ptr += sizeof(int); //skip past integer
  } else { //tag is INTERNAL
    ptr++; //skip past tag
    ret += sumPacked(ptr);
    ret += sumPacked(ptr);
  }
  return ret;
}
```

Figure 1. A low-level traversal of serialized tree data.

When relying on the usual pointer-based representation, this data can be readily traversed using standard idioms to perform computations such as summing all the leaf values:

```
sum t = case t of
          Leaf n   → n
          Node x y → (sum x) + (sum y)
```

But when represented on disk or sent over the wire, the same tree structure would not preserve pointers from a node to its children. Instead, the tree would be *serialized*, with the `Node`s and `Leaf`s of the tree laid out in a buffer in some sequential order. For example, the tree could be linearized in a left-to-right preorder, containing *tags* to mark data constructors and atomic fields such as integers, but ditching the pointers. Because it contains no pointers, this serialized representation is significantly more compact. But without this structural information, in most settings the pre-order serialization would be deserialized prior to processing, requiring more code than the simple `sum` function above.

However, this deserialization is not necessary—it is perfectly possible to write code that performs the same `sum` operation directly on the serialized representation. All that is necessary is for the code to visit every node in the tree, skipping over tags and `Node` data, and accumulating leaves into the `sum`. This traversal can be accomplished in existing languages, writing low-level buffer-processing code as in the C++ code shown in Fig. 1.

Essentially, this code operates as follows: `ptr` scans along the packed data structure. For each node type it encounters, it continues scanning through the node, retrieving the data it needs from the packed representation (in the case of `Leaf`s, the integer, in the case of `Node`s, nothing) and performing the necessary computation. Because this serialized representation is already in left-to-right preorder, no pointer-like accesses are necessary: scanning sequentially through the buffer suffices to access all the nodes of the tree. Note that the `sumPacked` function is

still recursive; the program stack helps capture the tree structure of the data.

There are several advantages to working directly on serialized data: the serialized representation can take many times fewer bytes to represent than a normal pointer-based representation; data can be traversed faster once in memory due to predictable memory accesses; *and* data can be read from disk without deserialization (e.g. via `mmap`).

However, working directly with serialized data is not always easy. First, programs written with typical pointer-based representations benefit from standard techniques, such as type checking, to help programmers avoid errors while constructing traversals of their data structures (so, e.g., type checking can prevent a programmer from reading an integer value out of an interior node of the tree, or from visiting the children of a leaf node). But operations on serialized representations provide no such protection: all of the data in the tree is packed into a flat buffer that is traversed using cursors (`ptr`, in Fig. 1). Cursors need to be manipulated carefully to visit the necessary portions of the buffer—skipping over the sections that are not needed—and read out the appropriate data, all without the safety net of a type checker. Hence, writing code to work directly on the serialized data can be tedious and error-prone.

We propose instead to write the above example in a language, *LoCal*, expressly designed to use dense serializations for its values. The LoCal `sum` function extends the simple functional one above with region and location annotations:

```
sum : ∀ l^r . Tree @ l^r → Int
sum [l^r] t = case t of
                Leaf (n : Int @ l_n^r ) → n
                Node (a : Tree @ l_a^r) (b : Tree @ l_b^r)
                  → (sum [l_a^r] a) + (sum [l_b^r] b)
```

This code operates on serialized data, taking locations of that data (input and output) as additional function arguments. It is a *region-polymorphic* function that performs a traversal within region r that contains serialized data. Well-typedness ensures that it only reads memory in a type-safe way. Location variables (l^r) have lexical scopes and are introduced as function arguments and pattern matches. For instance, in the above program, we cannot access child node locations ($l_a{}^r$,$l_b{}^r$) until we correctly parse the input data at l^r and ascertain that represents an intermediate node. Conversely, as we will see, to *construct* data the type system must enforce that adjacent fields be serialized consecutively.

Efficiency Using a type-safe language for serialized data manipulation eliminates *correctness* risks, but *efficiency* risks remain. As a simple example, consider taking the rightmost leaf of the same tree datatype:

```
rightmost : ∀ l^r . Tree @ l^r → Int
rightmost [l^r] tr =
   case tr of
      Leaf (n : Int @ l_n^r ) → n
      Node (a : Tree @ l_a^r) (b : Tree @ l_b^r) →
        rightmost [l_b^r] b
```

Here, operating on the *fully* serialized data representation is *more* expensive than operating on the pointer-based representation (linear instead of logarithmic). The reason is, in the fully serialized representation, the only way to access a particular field in a structure is to scan past all of the data that has been serialized before it.

Indirections and Random-access: One solution to this problem is to preserve some amount of *indirection* information (such as the size, in bytes, of the left subtree of each interior node). There has been substantial research in producing efficient layouts that preserve pointer information to allow easy traversal of recursive structures but still retain the locality benefits of linearized representations [5, 6, 11, 27] and such layouts are common in high-performance computing settings [8, 14, 16, 21, 22]. Unfortunately keeping pointer or indirection information in the linearized layout sacrifices some of the benefits of the serialized representation, and may not always be necessary. Indeed, it seems that there are two options in the design space: a fully-packed layout that eschews pointers entirely, but is specialized to specific traversal patterns, or a linearized order that pays the overhead of preserving pointer information in order to support arbitrary access patterns. But is there a way to interpolate between the two points in the design space? With LoCal, the answer is "yes" and random access can be restored on a per-data-constructor, per-field basis, without incurring any global cost of fixed representation choices.

3 From a Region- to a Location-Calculus

LoCal follows in the tradition of typed assembly language [18], region calculi [25], and Cyclone [9] in that it uses types to both expose and make safe low-level implementation mechanisms. The basic idea of LoCal is to first establish what data *share* which logical memory regions (essentially, buffers), and in what *order* those data reside, abstracting the details of computing exact addresses. For example, data constructor applications, such as `Leaf 3`, take an extra location argument in LoCal, specifying where the data constructor should place the resulting value in memory: `Leaf` l `3`. This location becomes part of type of the value: $Tree@l$. Every location resides in a region, and when we want to name that region, we write l^r.

Locations represent information about where values are in a store, but are less flexible than pointers. They

are introduced relative to other locations. A location variable is either *after* another variable, or it is at the beginning of a region, thus specifying a serial order. If location l_2 is declared as l_2 `= after(`$Tree@{l_1}^r$`)`, then l_2 is *after* every element of the tree rooted at l_1.

Regions in LoCal represent the memory buffers containing serialized data structures. Unlike some other region calculi, in LoCal, values in a region *may* escape the static scope which binds and allocates that region. In fact, an extension introduced later in §3.4 specifically relies on inter-region pointers and coarse-grained garbage collection of regions.

LoCal is a first-order, call-by-value functional language with algebraic datatypes and pattern matching. Programs consist of a series of datatype definitions, function definitions, and a main expression. LoCal programs can be written directly by hand, and LoCal also serves as a practical *intermediate language* for other tools or front-ends that want to convert computations to run on serialized data (essentially fusing a consuming recursion with the deserialization loop). We return to this use-case in §5.

Allocating to output regions Now that we have seen how data constructor applications are parameterized by locations, let us look at a more complex example than those of the prior section. Consider `buildtree`, which constructs the same trees consumed by `sum` and `rightmost` above. First, in the source language without locations:

```
buildtree : Int → Tree
buildtree n = if n == 0
              then Leaf 1
              else Node (buildtree (n - 1))
                        (buildtree (n - 1))
```

Then in LoCal, where the type scheme binds an output rather than input location:

```
buildtree : ∀ l^r . Int → Tree @ l^r
buildtree [l^r] n =
  if n == 0 then (Leaf l^r 1) -- write tag + int to
output
  else -- skip past tag:
       letloc l_a^r = l^r + 1 in
       -- build left in place:
       let left : Tree @ l_a^r =
           buildtree [l_a^r] (n - 1) in
       -- find start of right:
       letloc l_b^r = after(Tree @ l_a^r) in
       -- build right in place:
       let right : Tree @ l_b^r =
           buildtree [l_b^r] (n - 1) in
       -- write datacon tag, connecting things together:
       (Node l^r left right)
```

Here, we see that LoCal must represent locations that have *not yet been written*, i.e., they are output destinations. Nevertheless, in the recursive calls of `buildtree` this location is passed as an argument: a form of destination-passing style [23]. The type system guarantees that memory will be initialized and written exactly once. The output location is threaded through the recursion to build the left subtree, and then offset to compute the starting location of the right subtree. It might appear that computing `after(`$Tree@{l_a}^r$`)` could be quite expensive, if there is a large tree at that location. This does not need to be the case. In §4 we will present different techniques for efficiently compiling LoCal programs without requiring linear walks through serialized data.

One of the goals of LoCal is to support several compilation strategies. One extreme is compiling programs to work with a representation of data structures that do not include *any* pointers or indirections at run-time—within such a representation, the size of a value can be observed by threading through "end witnesses" while consuming packed values: for example, `buildtree` above would *return* ${l_b}^r$, rather than computing it with an `after` operation. (The end-witness strategy was already at use in *Gibbon* [30], which previously compiled functions on fully serialized data, while not preserving asymptotic complexity.) Next, we will present a formalized core subset of LoCal, its type system (§3.2), and operational semantics (§3.3), before moving on to implementation (§4, §5) and evaluation (§7).

3.1 Formal Language and Grammar

Fig. 2 gives the grammar for a formalized core of LoCal. We use the notation $\overrightarrow{x}$ to denote a vector $[x_1, \ldots, x_n]$, and $\overrightarrow{x_i}$ the item at position i. To simplify presentation, the language supports algebraic datatypes without any base primitive types, but could be extended in a straightforward manner to represent primitives such as an `Int` type or tuples. The expression language is based on the first-order lambda calculus, using A-normal form. The use of A-normal form simplifies our formalism and proofs without loss of generality.

Like previous work on region-based memory [26], LoCal has a special binding form for introducing region variables, written as `letregion`. Location variables are similarly introduced by `letloc`. The pattern-matching form `case` binds variables to serialized values, as well as binding the location for each variable. We require that each bound location in a source program is unique.

The `letloc` expression binds locations in only three ways: a location is either the *start* of a region (meaning, the location corresponds to the very beginning of that region), is immediately after another location, or it occurs *after* the last position occupied by some previously allocated data constructor. For the last case, the location is written to exist at (`after` $\tau@l^r$), where l is already bound in a region, and has a value written to it.

$K \in$ Data Constructors, $\tau \in$ Type Constructors,
$x, y, f \in$ Variables, $l, l^r \in$ Symbolic Locations,
$r \in$ Regions, $i, j \in$ Region Indices,
$\langle r, i \rangle^l \in$ Concrete Locations

Top-Level Programs	top	$::= \overrightarrow{dd}\ ;\overrightarrow{fd}\ ; e$
Datatype Declarations	dd	$::= \texttt{data}\ \tau = \overrightarrow{K\ \overrightarrow{\tau}}$
Function Declarations	fd	$::= f : ts; f\ \overrightarrow{x} = e$
Located Types	$\hat{\tau}$	$::= \tau @ l^r$
Type Scheme	ts	$::= \forall_{\overrightarrow{l^r}}.\ \overrightarrow{\hat{\tau}} \to \hat{\tau}$
Values	v	$::= x \mid \langle r, i \rangle^{l^r}$
Expressions	e	$::= v$
		$\mid f\ [\overrightarrow{l^r}]\ \overrightarrow{v}$
		$\mid K\ l^r\ \overrightarrow{v}$
		$\mid \texttt{let}\ x : \hat{\tau} = e\ \texttt{in}\ e$
		$\mid \texttt{letloc}\ l^r = le\ \texttt{in}\ e$
		$\mid \texttt{letregion}\ r\ \texttt{in}\ e$
		$\mid \texttt{case}\ v\ \texttt{of}\ \overrightarrow{pat}$
Pattern	pat	$::= K\ \ (\overrightarrow{x : \hat{\tau}})\ \ \to\ \ e$
Location Expressions	le	$::= (\texttt{start}\ r)$
		$\mid (l^r + 1)$
		$\mid (\texttt{after}\ \hat{\tau})$

Figure 2. Grammar of LoCal

Values in LoCal are either (non-location) variables or *concrete locations*. In contrast to bound location variables, concrete locations do not occur in source programs; rather, they appear at runtime, created by the application of a data constructor, which has the effect of extending the store. Every application of a data constructor writes a *tag* to the store, and concrete locations allow the program to navigate through it. To distinguish between concrete locations and location variables in the formalism, we refer to the latter as *symbolic locations*. A concrete location is a tuple $\langle r, i \rangle^l$ consisting of a region, an index, and symbolic location corresponding to its binding site. The first two components are sufficient to fully describe an *address* in the store.

3.2 Static Semantics

In Fig. 3, we extend the grammar with some extra details necessary for describing the type system. The typing rules for expressions in LoCal are given in Fig. 4, where the rule form is as follows:

$$\Gamma; \Sigma; C; A; N \vdash A'; N'; e : \hat{\tau}$$

The five letters to the left of the turnstile are different environments. Γ is a standard typing environment. Σ is

Typing Env.	Γ	$::= \{\, x_1 \mapsto \hat{\tau}_1,\ \ldots\ , x_n \mapsto \hat{\tau}_n \,\}$
Store Typing	Σ	$::= \{\, l_1^{r_1} \mapsto \tau_1,\ \ldots\ , l_n^{r_n} \mapsto \tau_n \,\}$
Constraint Env.	C	$::= \{\, l_1^{r_1} \mapsto le_1,\ \ldots\ , l_n^{r_n} \mapsto le_n \,\}$
Allocation Pointers	A	$::= \{\, r_1 \mapsto ap_1,\ \ldots\ , r_n \mapsto ap_n \,\}$
		where $ap = l^r \mid \varnothing$
Nursery	N	$::= \{\, l_1^{r_1},\ \ldots\ , l_n^{r_n} \,\}$

Figure 3. Extended grammar of LoCal for static semantics

a store-typing environment, which maps all *materialized* symbolic locations to their types. That is, every location in Σ *has been written* and contains a value of type $\Sigma(l^r)$. C is a constraint environment, which keeps track of how symbolic locations relate to each other. A maps each region in scope to a location, and is used to symbolically track the allocation and incremental construction of data structures; A can be thought of as representing the *focus* within a region of the computation. N is a nursery of all symbolic locations that have been allocated, but not yet written to. Locations are removed from N upon being written to, as the purpose is to prevent multiple writes to a location. Both A and N are threaded through the typing rules, also occuring in the output (to the right of the turnstile).

The T-Var rule ensures that the variable is in scope, and the symbolic location of the variable has been written to. T-Concrete-Loc is very similar, and also just ensures that the symbolic location has been written to. T-Let is straightforward, but note that along with Γ, it also extends Σ to signify that the location l has materialized.

In T-LetRegion, extending A with an empty allocation pointer brings the region r in scope, and also indicates that a symbolic location has not yet been allocated in this region.

There are three rules for introducing locations (T-LetLoc-Start (see Fig. 13 in the appendix), T-LetLoc-Tag and T-LetLoc-After), corresponding to three ways of allocating a new location in a region. A new location is either: at the start of a region, one cell after an existing location, or after the data structure rooted at an existing location. Introducing locations in this fashion sets up an ordering on locations, and the typing rules must ensure that the locations are used in a way that is consistent with this intended ordering. To this end, each such rule extends the constraint environment C with a constraint that is based on how the location was introduced, and N is extended to indicate that the new location is in scope and unwritten.

[T-Var]

$$\frac{\Gamma(x) = \tau@l^r \qquad \Sigma(l^r) = \tau}{\Gamma; \Sigma; C; A; N \vdash A; N; x : \tau@l^r}$$

[T-Concrete-Loc]

$$\frac{\Sigma(l^r) = \tau}{\Gamma; \Sigma; C; A; N \vdash A; N; \langle r, i\rangle^l : \tau@l^r}$$

[T-Let]

$$\frac{\Gamma; \Sigma; C; A; N \vdash A'; N'; e_1 : \tau_1@{l_1}^{r_1} \quad {l_1}^{r_1} \in N \quad {l_1}^{r_1} \notin N' \quad \Gamma'; \Sigma'; C; A'; N' \vdash A''; N''; e_2 : \tau_2@{l_2}^{r_2} \quad {l_2}^{r_2} \in N}{\Gamma; \Sigma; C; A; N \vdash A''; N''; \texttt{let}\ x : \tau_1@{l_1}^{r_1} = e_1\ \texttt{in}\ e_2 : \tau_2@{l_2}^{r_2}}$$

where $\Gamma' = \Gamma \cup \{ x \mapsto \tau_1@{l_1}^{r_1} \}$; $\Sigma' = \Sigma \cup \{ {l_1}^{r_1} \mapsto \tau_1 \}$

[T-LetRegion]

$$\frac{\Gamma; \Sigma; C; A'; N \vdash A''; N'; e : \tau@{l'}^{r'} \quad {l'}^{r'} \in N}{\Gamma; \Sigma; C; A; N \vdash A''; N'; \texttt{letregion}\ r\ \texttt{in}\ e : \tau@{l'}^{r'}}$$

where $A' = A \cup \{ r \mapsto \varnothing \}$

[T-LetLoc-Tag]

$$\frac{A(r) = {l'}^{r} \qquad {l'}^{r}, {l''}^{r''} \in N \qquad {l}^{r} \notin N'' \qquad {l}^{r} \neq {l''}^{r''} \quad \Gamma; \Sigma; C'; A'; N' \vdash A''; N''; e : \tau''@{l''}^{r''}}{\Gamma; \Sigma; C; A; N \vdash A''; N''; \texttt{letloc}\ l^r = ({l'}^{r} + 1)\ \texttt{in}\ e : \tau''@{l''}^{r''}}$$

where $C' = C \cup \{ l^r \mapsto ({l'}^{r} + 1) \}$; $A' = A \cup \{ r \mapsto l^r \}$; $N' = N \cup \{ l^r \}$

[T-LetLoc-After]

$$\frac{A(r) = {l_1}^{r} \qquad \Sigma({l_1}^{r}) = \tau' \qquad {l_1}^{r} \notin N \qquad {l}^{r} \notin N'' \qquad {l}^{r} \neq {l'}^{r'} \quad \Gamma; \Sigma; C'; A'; N' \vdash A''; N''; e : \tau'@{l'}^{r'} \qquad {l'}^{r'} \in N}{\Gamma; \Sigma; C; A; N \vdash A''; N''; \texttt{letloc}\ l^r = (\texttt{after}\ \tau'@{l_1}^{r})\ \texttt{in}\ e : \tau'@{l'}^{r'}}$$

where $C' = C \cup \{ l^r \mapsto (\texttt{after}\ \tau'@{l_1}^{r}) \}$; $A' = A \cup \{ r \mapsto l^r \}$; $N' = N \cup \{ l^r \}$

[T-DataConstructor]

$$\frac{\mathit{TypeOfCon}(K) = \tau \qquad \mathit{TypeOfField}(K, i) = \overrightarrow{\tau_i'} \qquad l^r \in N \qquad A(r) = \overrightarrow{{l_n}^{r}}\ \text{if } n \neq 0\ \text{ else } l^r \qquad C(\overrightarrow{{l_1}^{r}}) = l^r + 1 \qquad C(\overrightarrow{{l_{j+1}}^{r}}) = (\texttt{after}\ (\overrightarrow{\tau_j'@l_j^{r}})) \quad \Gamma; \Sigma; C; A; N \vdash A; N; \overrightarrow{v_i} : \overrightarrow{\tau_i'@{l_i}^{r}}}{\Gamma; \Sigma; C; A; N \vdash A'; N'; K\ l^r\ \overrightarrow{v} : \tau@l^r}$$

where $A' = A \cup \{ r \mapsto l^r \}$; $N' = N - \{ l^r \}$; $n = |\overrightarrow{v}|$; $i \in I = \{ 1, \ldots, n \}$; $j \in I - \{ n \}$

Figure 4. Selected type-system rules for LoCal.

Additionally, the location-introduction rules use A to ensure that a program must introduce locations in a certain pattern (corresponding to the left-to-right allocation and computation of fields, as explained in §3.3). In A, each region is mapped to either the right-most allocated symbolic location in that region (if it is unwritten), or to the symbolic location of the most recently materialized data structure. This mapping in A is used by the typing rules to ensure that: (1) T-LetLoc-Start may only introduce a location at the start of a region once; (2) T-LetLoc-Tag may only introduce a location if an unwritten location has just been allocated in that region (to correspond to the tag of some soon-to-be-built data structure); and (3) T-LetLoc-After may only introduce a location if a data structure has just been materialized at the end of the region, and the programmer wants to allocate *after* it. To attempt, for example, to allocate the location of the right sub-tree of a binary tree *before* materializing the left sub-tree would be a type error. Each location-introduction rule also ensures that the introduced location must be written to at some point, by checking that it's absent from the nursery after evaluating the expression.

In order to type an application of a data constructor, T-DataConstructor starts by ensuring that the tag being written and all the fields have the correct type. Along with that, the locations of all the fields of the constructor must also match the expected constraints. That is, the location of the first field should be immediately after the constructor tag, and there should be appropriate *after* constraints for other fields in the location constraint environment. After the tag has been written, the location l is removed from the nursery to prevent multiple writes to a location. As mentioned earlier, LoCal uses destination-passing style. To guarantee destination-passing style, it suffices to ensure that a function returns its value in a location passed from its caller. The LoCal type system enforces this property by using constraints of the form $l' \neq l$ in the premises of the typing rules of the operations that introduce new locations

As demonstrated by T-DataConstructor, the type system enforces a particular ordering of writes to ensure the resulting tree is serialized in a certain order. Some interesting patterns are expressible with this restriction (for example, writing or reading multiple serialized trees in one function), and, as we will address shortly in §3.4, LoCal is flexible enough to admit extensions that soften this restriction and allow for programmers to make use of more complicated memory layouts.

A simple demonstration of the type system is shown in Table 1, which tracks how A, C, and N change after each line in a simple expression that builds a binary tree with leaf children. Introducing l at the top establishes that it is at the beginning of r, A maps r to l, and N contains l. The location for the left sub-tree, l_a, is defined to be +1 after it, which updates r to point to l_a in A and adds a constraint to C for l_a. Actually constructing the `Leaf`

Table 1. Step-by-step example of type checking a simple expression.

Code	A	C	N
`letloc` l^r = `start(r)`	$\{r \mapsto l^r\}$	$\varnothing$	$\{l^r\}$
`letloc` ${l_a}^r$ = l^r + 1	$\{r \mapsto {l_a}^r\}$	$\{{l_a}^r \mapsto l^r + 1\}$	$\{l^r, {l_a}^r\}$
`let x : T @` ${l_a}^r$ = `Leaf` ${l_a}^r$ `1`	$\{r \mapsto {l_a}^r\}$	$\{l_a \mapsto l^r + 1\}$	$\{l^r\}$
`letloc` ${l_b}^r$ = `after(T @` ${l_a}^r$`)`	$\{r \mapsto {l_b}^r\}$	$\{{l_a}^r \mapsto l^r + 1,$ ${l_b}^r \mapsto after(T@{l_a}^r)\}$	$\{l^r, {l_b}^r\}$
`let y : T @` ${l_b}^r$ = `Leaf` ${l_b}^r$ `2`	$\{r \mapsto {l_b}^r\}$	$\{{l_a}^r \mapsto l + 1,$ ${l_b}^r \mapsto after(T@{l_a}^r)\}$	$\{l^r\}$
`Node` l^r `x y`	$\{r \mapsto l^r\}$	$\{{l_a}^r \mapsto l^r + 1,$ ${l_b}^r \mapsto after(T@{l_a}^r)\}$	$\varnothing$

$$\begin{array}{rrl} \text{Store} & S & ::= \{\, r_1 \mapsto h_1,\ \ldots\ , r_n \mapsto h_n \,\} \\ \text{Heap} & h & ::= \{\, i_1 \mapsto K_1,\ \ldots\ , i_n \mapsto K_n \,\} \\ \text{Location Map} & M & ::= \{\, l_1^{r_1} \mapsto \langle r_1, i_1\rangle,\ \ldots\ , l_n^{r_n} \mapsto \langle r_n, i_n\rangle \,\} \end{array}$$

Figure 5. Extended grammar of LoCal for dynamic semantics

in the next line removes l_a to N, because it has been written to. Once l_a has been written, the next line can introduce a new location l_b *after* it, which updates the mapping in A and adds a new constraint to C. Once l_b has been written and removed from N in the next line, the final `Node` can be constructed, which expects the constraints to establish that l is before l_a, which is before l_b.

Additional rules (such as for function application, pattern matching) are conventional and are in the Appendix, D.2.

3.3 Dynamic Semantics

The dynamic semantics for expressions in LoCal are given in Fig. 6, where the transition rule is as follows.

$$S; M; e \Rightarrow S'; M'; e'$$

To model the behavior of reading and writing from an indexed memory, we introduce the *store*, S. The store is a map from regions to *heaps*, where each heap consists of an array of *cells*, which contain store values (data constructor tags). To bridge from symbolic to concrete locations, we use the *location map*, M, which is a map from symbolic to concrete locations.

Case expressions are treated by the D-Case rule. The objective of the rule is to load the tag of the constructor K located at $\langle r, i\rangle$ in the store and dispatch the corresponding case. The expression produced by the right-hand side of the rule is the body of the pattern, in which all pattern-bound variables are replaced by the concrete locations of the fields of the constructor K.

[D-DataConstructor]
$$S; M; K\ l^r\ \overrightarrow{v} \Rightarrow S'; M; \langle r, i\rangle^{l^r}$$
$$\text{where}\ \ S' = S \cup \{\, r \mapsto (i \mapsto K)\,\};\ \langle r, i\rangle = M(l^r)$$

[D-LetLoc-Start]
$$S; M; \texttt{letloc}\ l^r = (\texttt{start}\ r)\ \texttt{in}\ e \Rightarrow S; M'; e$$
$$\text{where}\ \ M' = M \cup \{\, l^r \mapsto \langle r, 0\rangle\,\}$$

[D-LetLoc-Tag]
$$S; M; \texttt{letloc}\ l^r = l'^r + 1\ \texttt{in}\ e \Rightarrow S; M'; e$$
$$\text{where}\ \ M' = M \cup \{\, l^r \mapsto \langle r, i+1\rangle\,\};\ \langle r, i\rangle = M(l'^r)$$

[D-LetLoc-After]
$$S; M; \texttt{letloc}\ l^r = (\texttt{after}\ \tau@{l_1}^r)\ \texttt{in}\ e \Rightarrow S; M'; e$$
$$\text{where}\ \ M' = M \cup \{\, l^r \mapsto \langle r, j\rangle\,\};\ \langle r, i\rangle = M({l_1}^r)$$
$$\tau; \langle r, i\rangle; S \vdash_{ew} \langle r, j\rangle$$

[D-Case]
$$S; M; \texttt{case}\ \langle r, i\rangle^{l^r}\ \texttt{of}\ [\ldots, K\ \ (\overrightarrow{x : \tau@l^r})\ \to\ e, \ldots] \Rightarrow$$
$$S; M'; e[\langle r, \overrightarrow{w}\rangle^{\overrightarrow{l^r}} / \overrightarrow{x}]$$
$$\begin{aligned} \text{where}\ \ & M' = M \cup \{\, \overrightarrow{l_1^r} \mapsto \langle r, i+1\rangle, \ldots, \overrightarrow{l_{j+1}^r} \mapsto \langle r, \overrightarrow{w_{j+1}}\rangle\,\} \\ & \overrightarrow{\tau_1}; \langle r, i+1\rangle; S \vdash_{ew} \langle r, \overrightarrow{w_1}\rangle \\ & \overrightarrow{\tau_{j+1}}; \langle r, \overrightarrow{w_j}\rangle; S \vdash_{ew} \langle r, \overrightarrow{w_{j+1}}\rangle \\ & K = S(r)(i);\ j \in \{\, 1, \ldots, n-1\,\};\ n = |\overrightarrow{x : \tau}| \end{aligned}$$

Figure 6. Selected dynamic-semantics rules for LoCal.

The concrete locations of the fields are obtained by the following process. If there is at least one field, then its starting address is the position one cell after the constructor tag. The starting addresses of subsequent fields depend on the sizes of the trees stored in previous fields.

A feature of LoCal is the flexibility it provides to pick the serialization layout. Our formalism uses our *end-witness rule* to abstract from different layout decisions. Given a type τ, a starting address $\langle r, i_s\rangle$, and store S, the rule below asserts that address of the end witness is $\langle r, i_e\rangle$.

$$\tau; \langle r, i_s\rangle; S \vdash_{ew} \langle r, i_e\rangle$$

Using this rule, the starting address of the second field is obtained from the end witness of the first, the starting address of the third from the end witness of the second, and so on.

The allocation and finalization of a new constructor is achieved by some sequence of transitions, starting with the D-LetLoc-Tag rule, then involving some number of transitions of the D-LetLoc-After rule, depending on the number of fields of the constructor, and finally

ending with the D-DataConstructor transition. The D-LetLoc-Tag rule allocates one cell for the tag of some new constructor of a yet-to-be determined type, leaving it to later to write to the new location. The resulting configuration binds its l to the address $\langle r, i+1\rangle$, that is, the address one cell past given location l' at $\langle r, i\rangle$. Fields that occur after the first are allocated by the D-LetLoc-After rule. Here, its l is bound to the address $\langle r, j\rangle$ one past the last cell of the constructor represented by its given symbolic location l_1. Like the D-Case rule, the required address is obtained by application of end-witness rule to the starting address of the given l_1 at the type of the corresponding field τ. The final step in creating a new data constructor instance is the D-DataConstructor rule. It writes the specified constructor tag K at the address in the store represented by the symbolic location l.

The D-LetLoc-Start rule for the `letloc` with `(start r)`. expression binds the location to the starting address in the region and starts running the body.

The remaining rules have conventional behavior, and are available in the Appendix, D.3. Also in the Appendix, there is a detailed explanation of the evaluation of a sample program (§D.3.1).

3.3.1 Type Safety

The key to proving type safety is our store-typing rule, given in full in the Appendix, D.5.

$$\Sigma; C; A; N \vdash_{wf} M; S$$

The store typing specifies three categories of invariants. The first enforces that allocations occur in the sequence specified by the constraint environment C. In particular, if there is some location l in the domain of C, then the location map and store have the expected allocations at the expected types. For instance, if $(l \mapsto (\texttt{after}\ \tau @ l'^{r})) \in C$, then l' maps to $\langle r, i_1\rangle$ and l to $\langle r, i_2\rangle$ in the location map, and i_2 is the end witness of i_1 at type τ in the store, at region r. The second category enforces that, for each symbolic location such that $(l \mapsto \tau) \in \Sigma$, there is some $\langle r, i_1\rangle$ for l in the location map and i_1 has some end witness i_2 at type τ. The final category enforces that each address in the store is written once. This property is asserted by insisting that, if $l \in N$, then there is some $\langle r, i\rangle$ for l in the location map, but there is no write to for i at r in the store. To support this property, there are two additional conditions which require that the most recently allocated location (tracked by to A, N) is at the end of its respective region.

The type safety of LoCal follows from the following result.

Theorem 3.1 (Type safety)

If $(\varnothing; \Sigma; C; A; N \vdash A'; N'; e : \hat{\tau}) \wedge (\Sigma; C; A; N \vdash_{wf} M; S)$
and $S; M; e \Rightarrow^{n} S'; M'; e'$
then $(e'\ \mathit{value}) \vee (\exists S'', M'', e''.\ S'; M'; e' \Rightarrow S''; M''; e'')$

Proof The type safety follows from an induction with the progress and preservation lemmas, shown in the Appendix, D.6.

3.4 Offsets and Indirections

As motivated in §2, it is sometimes desirable to be able to "jump over" part of a serialized tree. As presented so far, LoCal makes use of an end witness judgment to determine the end of a particular data structure in memory. The simplest computational interpretation of this technique is, however, a linear scan through the store. Luckily, extending the language to account for storing and making use of *offset* information for certain datatypes is straightforward, and does not add conceptual difficulty to neither the formalism nor type-safety proof.

Such an extension may use annotations on datatype declarations that identify which fields of a given constructor are provided *offsets* and to permit cells in the store to hold offset values. Because the offsets of a given constructor are known from its type, the D-LetLoc-Tag rule can allocate space for offsets when it allocates space for the tag. It is straightforward to fill the offset fields because D-DataConstructor rule already has in hand the required offsets, which are provided in the arguments of the constructor. Finally, the D-Case rule can use offsets instead of the end-witness rule.

Indirections permit fields of data constructors to point across regions, and thus require adding an annotation form (e.g., an annotation on the type of a constructor field to indicate an indirection) and extending the store to hold pointers. Fortunately, as discussed later, regions in LoCal are never collected; they are garbage collected in our implementation. Every time an indirection field is constructed, space for the pointer is allocated using a transition rule similar to the D-LetLoc-Tag rule. The D-DataConstructor rule receives the address of the indirection in the argument list, just like any other location and writes the indirection pointer to the address of the destination field.

To type check, the type system extends with two new typing rules and a new constraint form to indicate indirections. To maintain type safety in the presence of offsets and indirections, the store typing rule needs to be extended to include them. Because the programmer is not manually managing the creation or use of offsets or indirections (they are below the level of abstraction, indicated by annotating the datatype, but not changing the

code), the store-typing rule generalizes straightforwardly and the changes preserve type safety.

In datatype annotations each field can be marked to store its offset in the constructor *or* be represented by an indirection pointer (currently not both):

```
data T = K1 T (Ind T) | K2 T (Offset T) | K3 T
```

Type annotations would also be the place to express *permutations* on fields that should be serialized in a different order, (e.g., postorder). But it is equivalent to generating LoCal with reordered fields in the source program.

4 Compiling the Location Calculus

In this section we present a compiler for the LoCal language, which consists of the formalized core from §3, extended with various primitive types, tuples, convenience features, and a standard library. A well-typed LoCal program guarantees correct handling of regions, but the implementation still has substantial leeway to further modify datatypes and the functions that use them. By default, the compiler we present inserts enough indirection in datatypes to preserve the asymptotic complexity of the source functions (under the assumption of $O(1)$ field access), but we also provide a mode—activated globally or per-datatype—that leaves the data types *fixed* and instead introduces inefficient "dummy traversals" and copying code into compiled functions. (In this mode, our compiler produces a similar result to what Gibbon produced previously [30]—for comparison, we will distinguish between Gibbon2 (with LoCal) and Gibbon1 (without LoCal) in §7.)

Note that this "inflexible" mode—which doesn't allow the compiler to insert indirections—is also used when reading in external data. In our LoCal implementation, we provide a mechanism for any datatype to be read from a file (via `mmap`),whose contents are the pointer-free, full serialization. We use the same basic encoding as Haskell's `Data.Serialize` module derives by default, but plan to extend it in the future.

Ultimately, because LoCal is meant to be generated by tools as well as programmers, its goal is to add value in both safety and performance, but to leave *open* the design space of broader optimization questions to a front-end that targets LoCal. One example of such a front-end tool is in §5.

Compiler Structure We implement LoCal with a micropass, whole-program compiler that performs full region/location type checking between every pair of passes on the LoCal intermediate representation (IR). After a series of LoCal →LoCal passes, we lower to a second IR, *NoCal*. As shown in Fig. 7, NoCal is not a calculus at all, but a low-level language where memory operations are made explicit. NoCal functions closely resemble the

$$
\begin{array}{rrll}
 & n \in \text{Integers} & & \\
\text{Types} & \tau & ::= \ldots \mid \texttt{Cursor} \mid \texttt{Int} & \\
\text{Pattern} & \mathit{spat} & ::= K\ (x : \texttt{Cursor}) \rightarrow e & \\
\text{Expressions} & e & ::= \ldots & \\
 & & \mid \texttt{switch}\ x\ \texttt{of}\ \mathit{spat} & \\
 & & \mid \texttt{readInt}\ x \mid \texttt{writeInt}\ x\ n & \\
 & & \mid \texttt{readTag}\ x \mid \texttt{writeTag}\ x\ K & \\
 & & \mid \texttt{readCursor}\ x & \\
 & & \mid \texttt{writeCursor}\ x\ \langle r, i \rangle^{l} &
\end{array}
$$

Figure 7. Grammar of NoCal (an extension of LoCal)

C code shown in Fig. 1. Code in this form manipulates pointers into regions we call *cursors* because of their (largely continuous) motion through regions. We represent NoCal internally as a distinct AST type, with high level (non-cursor) operations excluded.

Within this prototype compiler, tuples, and built-in scalar types like Int, Bool etc. are *unboxed* (never require indirections). In the following subsections, we describe the compiler in four stages. Similar to NoCal, our compiler represents programs at these stages with AST types that track changes in the grammar needed by each pass. After these four steps, the final backend is completely standard. It eliminates tuples in the *unariser*, performs simple optimizations, and generates C code. Because of inter-region indirections, a small LoCal runtime system is necessary to support the generated code, as described in Section §4.5.

4.1 Compiler (1/4): Finding Traversals

Pattern matches in LoCal bind all constructor fields, including those that occur at non-constant offsets, later in memory. The compiler must determine which fields are reachable based on either (1) constant offsets, (2) stored offsets/indirections present in the datatype, or (3) by leveraging traversals already present in the code that scan past the relevant data. The third case corresponds to determining end witnesses in the formal semantics. Likewise, this compiler pass identifies data reached by the work the program already performs.

To this end, we use a variation of a technique we previously proposed [30]. Specifically, we assign *traversal effects* to functions. A function is said to *traverse* it's input location if it touches every part of it. In LoCal, a case expression is the only way to induce a traversal effect. If all clauses of the case expression in turn traverse the packed elements of the corresponding data constructors, the expression traverses to the end of the scrutinee. Traversing a location means witnessing the size of the value encoded at that location, and thus computing the

address of the *next* value in memory. After this pass, the type schemes of top-level function definitions reflect their traversal effects.

```
maplike : ∀ l1^r1 l2^r2. Tree @ l1^r1 -{l1,l2}-> Tree @ l2^r2
rightmost : ∀ l^r . Tree @ l^r -{}-> Int
```

4.2 Compiler (2/4): Implementing Random Access

Once we know what fields are traversed, we can also determine which fields are used but *not* naturally reachable by the program: e.g. the right subtree read by `rightmost`. In later stages of the compiler, we eliminate all direct references to pattern-matched fields *after* the first variable-sized one. This is where space/time optimization choices must be made: bytes for offsets v.s. cycles for unnecessary traversals.

To activate random-access for a particular field within a data constructor, we add additional information following the tag. Specifically, for a constructor `K T1 T2`, if we need immediate access to `T2`, we include a 4-byte relative-offset after the constructor.

Back-tracking Unfortunately, when we modify datatypes to add offsets, we invalidate previously computed location information. Thus the compiler *backtracks*, rewinding in time to before find-traversals (and inserting extra `letloc` expressions to skip-over the offset bytes themselves). Adding random access to one datatype never *increases* the set of constructors needing random-access to maintain work-efficiency, so in fact we only backtrack at most once.

In the default (offset-adding) mode, any function that demands random access to a field will determine the representation for all functions using the datatype. Our current LoCal compiler does *not* automate choices such as duplicating datatypes to achieve multiple encodings of the same data—that is left to the programmer or upstream tools.

If the LoCal compiler is passed a flag to *not* automatically change datatypes, then it must use the same approach we previously used in Gibbon [30]: insert dummy traversals that scan across earlier fields to reach later ones. Regardless of whether the offset or dummy-traversal strategy is used, at the end of this compiler phase, we blank non-first fields in each pattern match to ensure they are not referenced directly. So a pattern match in our tree examples becomes "`Node a _` → ⋯" or "`Node offset a _` → ⋯".

4.3 Compiler (3/4) Routing End Witnesses

Each of the traversal effects previously inferred proves we logically reach a point in memory, but to realize it in the program we add an additional return value to the function, witnessing the end-location for traversed values (as described in Vollmer et al. [30]). We extend the syntax to allow additional location-return values, equivalent to returning tuples. The `buildtree` example becomes:

```
buildtree : ∀ l^r. Int -{l}-> [after(Tree@l^r)] Tree@l^r
buildtree [l^r] n =
  if n == 0 then return [l^r + 9] (Leaf l^r 1)
  else letloc l_a^r = l^r + 1 in
       let [l_b^r] left = buildtree [l_a^r] (n - 1) in
       let [l_c^r] right = buildtree [l_b^r] (n - 1) in
       return [l_c^r] (Node l^r left right)
```

The `letloc` form for the location of the right subtree is gone, because the first recursive call to `buildtree` returned $l_b{}^r$ as an end-witness, bound here with an extended `let` form. Similarly, the final return statement returns the end-witness of the right subtree, $l_c{}^r$, using a new `return` form in the IR.

4.4 Compiler (4/4): Converting to NoCal

In this stage, we convert from LoCal into NoCal, switching to imperative cursor manipulation. At this stage, location arguments and return values turn into first-class cursor values (pointers into memory buffers representing regions). The primitive operations on cursors read or write one atomic value, and advance the cursor to the next spot. We drop much of the type information at this phase, and `rightmost` becomes:

```
rightmost : Cursor → Int
rightmost cin =    -- take a pointer as input
  switch cin of    -- read one byte
    Leaf(cin1)  →
      let (cin2,n) = readInt(cin1) in n
    Node(cin1) →   -- only get a pointer to the 1st field
      let (cin2,ran) = readCursor(cin1) in
      rightmost ran
```

Here the `switch` construct is simpler than `case`, reading a one byte tag, switching on it, and binding a cursor to the very next byte in the stream (`cin1 == cin + sizeof(tag)` == cin+1).

The key takeaway here is that, because the relationship between location variables and normal variables representing packed data are made explicit in the types and syntax of LoCal, this pass does not require any complicated analysis. Also, in NoCal we can finally reorder writes to more often be *in order* in memory, which aids prefetching and caching, because writes are ordered only by data-dependencies for computing *locations*, with no ordering needed on the side-effects themselves.

4.5 LoCal Runtime System & Allocator

The LoCal runtime system is responsible for region-based memory management. A detailed description of the memory management strategy is available in the Appendix, C.

In brief, we use region-level reference counts. Each region is implemented as linked list of contiguous memory chunks, doubling in size. This memory is write-once, and immutability allows us to track reference counts *only* at the region level. Exiting a `letregion` decrements the region's count, and it is freed when no other regions point into it.

5 High-Level Programs: HiCal to LoCal

LoCal captures a notion of computation over (mostly) serialized data, *exposing* choices about representation. It provides the levers needed by a human or another tool to explore the design space of optimization trade-offs above this level, i.e., for the human or tool to answer the question *"how do we globally optimize a pipeline of functions on serialized data?"*.

First, if multiple functions use the same datatype, do they standardize on one representation? Or does that datatype take different encodings at different points in the pipeline (implemented by cloning the datatype and presenting it to LoCal with different annotations)? Second, when up against the constraint of *already-serialized* data on disk, the compiler can't change the existing representation, if the external data *lacks offsets*, is it better to *force* the first consuming function to use that representation, or to insert an extra *reserialization* step to convert[1]? Third, can the compiler permute fields to improve performance or reduce the stored offsets needed?

This large space of future work is beyond the scope of this paper, but we nevertheless illustrate the process of integrating LoCal into Gibbon. The front-end language for Gibbon, HiCal, is a vanilla purely functional language without any region or location annotations. It hides data-layout from the programmer (and the low-level control that comes with it). It also facilitates comparison with mature compilers, as HiCal runs standard functional programs: for example, the *unannotated* examples we've seen in this paper.

The syntax for HiCal is a subset of Haskell syntax, supporting algebraic data types and top-level function definitions. It is a monomorphic, strict functional programming language, and for simplicity it is first order, like LoCal. In future-work, we plan to add support for a higher-order, polymorphic front-end language through standard monomorphization and defunctionalization. (An interesting consequence of this will be that closures become regular datatypes, such that a list of closures could be serialized in a dense representation.)

[1]Still faster than traditional deserialization: no object graph allocation.

Implementing HiCal The compiler must perform a variant of *region inference* [25, 26], but differs from previous approaches in some key ways. The inference procedure uses a combination of standard techniques from the literature and specialized approach for satisfying LoCal's particular needs[2]. Because the inference must determine not only what region a value belongs to, but *where* in that region it will be, the inference procedure returns a set of constraints for an expression similar to the constraint environment used in the typing rules in Fig. 4, which are used to determine placement of symbolic location bindings. Additionally, certain locations are marked as *fixed* (function parameters, data constructor arguments), and when two fixed locations attempt to unify it signals the need for an indirection, and the program must be transformed accordingly.

Our current implementation adds an extra variant to every data type[3] representing a single indirection (called `I`). For example, a binary tree `T` becomes

```
data T = Leaf | Node T T | I (Ind T)
```

The identity function `id x = x`, when compiled to LoCal, is `id x = I x`. Likewise, sharing demands indirections, and `let x = _ in Node x x` becomes `let x = _ in Node x (I x)`.

6 Related Work

Many libraries exist for working with serialized data, and a few make it easier to use serialized data as in-memory data, or to export the host-language's pre-existing in-memory format as external data. Cap'N Proto[4], is designed to eliminate encoding/decoding by standardizing on a new binary format for use in memory as well on disk/network. *Compact Normal Forms* (CNF) [33] is a feature provided by the Glasgow Haskell Compiler since release 8.2. The idea is that any purely functional value, once fully evaluated, can be *compacted* into its own region of the heap — capturing a transitive closure of its reachable heap. After compaction, the CNF can be stored externally and loaded back into the heap later. Persistent languages tackle the problem of automatically moving data between disk and in-memory representations [1, 2, 13], and can swizzle pointers as part of this translation to create more efficient representations. However, like CNF, these representations still maintain pointers, so cannot realize the full advantage of our system.

If we look instead at compiler support for computing with data in dense or external forms, there are many compilers for stream processing languages [20, 24]—or

[2]The *Directed Inference Engine for region Types*, if you will.
[3]These indirections do double-duty in allowing the memory manager to use non-contiguous memory slabs for a region C.
[4]An "insanely fast data interchange format," https://capnproto.org/, [28]

restricted languages such as XPath [19]—that generate efficient computations over data streams. These are somewhat related, but LoCal differs in targeting general purpose recursive functions over algebraic datatypes. In this category, the main published approach is our prior work on Gibbon [30], which compiles idiomatic programs to operate on serialized data. However, the Gibbon approach described previously can only handle fully serialized data and thus introduces asymptotic slowdowns as we'll see in the next section. (At the time, we considered adding indirections as future work but they were not part of the compiler.) Also, the prior Gibbon compiler had no analogue to our location calculus: no way for a type-system to enforce correct handling of regions and locations within serialized data—which provides a much stronger foundation for building such compilers.

The problem of computing *without* deserializing can be viewed as a ***fusion/deforestation*** problem: to fuse the compute loop with the deserialize loop. But traditional deforestation approaches [31], don't rise to being able to handle a full deserializer, and popular approaches based on more restrictive *combinator libraries* [7] are less expressive than HiCal and LoCal.

Ornaments are a body of theory regarding connections between related data structure that differ based on additions or reorganization [15]. Indeed, LoCal's addition of offset fields to data *is* ornamentation. Practical implementations of ornaments [32] provide support for lifting functions across types related by ornaments, transforming the code. However, the isomorphism between a datatype and its serialized form is not an ornament, and thus lifting functions across that isomorphism is not supported.

Finally, LoCal relates to a broader literature on optimizing tree-traversing programs and heap representations. For the interested reader, this is detailed in the appendix, B. LoCal does not allow construction of cyclic values (only DAGs), so it is less related to graph-processing systems.

7 Evaluation

In this section we evaluate our implementation by looking both at benchmarks that operate on data already in memory, as well as programs that process serialized data stored on disk. One of the main results of this work is categorical rather than quantitative — that a compilation approach based on LoCal/HiCal can lift functions over (mostly) serialized data without changing their asymptotic complexity.

We compare performance against prior systems for computing with serialized data. The approach we used previously in Gibbon (which we denote "Gibbon1" here) provides one point of comparison; it achieves constant-factor speedups over pointer-based representations (even discounting [de]serialization time itself) [30], except when it generates code with an asymptotic slowdown. We also compare against Cap'N Proto (v0.6.1) and CNF (GHC 8.2.2), described in §6.

The evaluations in this section are performed on a 2-socket Intel Xeon E5-2630, with 125GB of memory, running Ubuntu 16.04, GCC 5.4.0. Measurements reported are the median of 9 trials, where an individual *trial* is defined as a separate run of the program that takes at least one second of real time, discounting setup time. (For short benchmarks, we iterate them a sufficient number of times in an inner loop to reach the 1s threshold, then divide to compute average per-iteration time.)

7.1 Serialized→Serialized Benchmark Programs

We consider the following set of tree programs, which provide a panel of litmus tests for which kinds of tree operations are well-supported by which frameworks: **id**, **buildTree**, **rightmost**, **sumTree**, **copyTree**. These programs have been either shown before or are self-explanatory, but for reference we include a description of each in Appendix A.1. This first batch of experiments use the simple binary tree datatype with integer leaves (first introduced in §2). In each case the goal is to get from a serialized input in a buffer (as it would have come from the network or disk) to another serialized output ready to send. In general, we allow *appending to* but not destroying the input message.

We additionally consider operations on search-trees in the below benchmarks. These search trees store integer keys on all intermediate nodes, where keys on left subtrees are smaller than the root, and right subtrees are larger. The data type is: `data STr = Null | Leaf Int | Node Int STr STr` These benchmarks—**buildSearchTree**, **treeContains**, **treeInsert**, **findMax**, **propagateConstant**—are described in §A.1, except for one that is not self-explanatory:

repMax: a variant of the "repmin" [4, 17] program, which returns a new search tree of the same shape, with every value replaced by the *maximum* value of the original tree. It performs two passes over the tree – a bottom-up pass to compute the largest element (i.e. findMax), and a top-down pass to propagate this value in the tree (i.e. propagateConst).

7.2 Discussion of Results

Table 2 shows the results. The column labeled "Gibbon2" shows performance of HiCal programs (low-level LoCal control was not needed for any of these) using indirections and offsets, automatically. "Gibbon1" shows the

approach described in Vollmer et al. [30]. There are two major sources of overhead for our new approach versus Gibbon1:

1. Growable regions: In each case, our compiler starts with smaller, growable regions[5], which we require to create small output regions as in **id** or **treeInsert**, but we suffer the overhead of bounds-checking. On the other hand, Gibbon always stores fully serialized data in huge regions.
2. Likewise, we have found that the backend C compiler is sensitive to the number of cases in switch statements on data constructor tags (for instance, triggering the jump table heuristic). By including the possibility that each tag we read may be a tagged indirection, we increase code size and increase the number of cases in our switch statements.

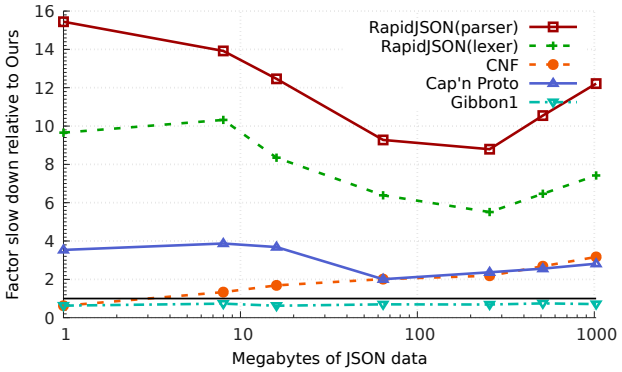
However, the benchmarks where indirections and random-access offsets are important (id, rightmost, treeInsert, findMax) show a huge difference between Gibbon1 and Gibbon2, as we would expect based on Gibbon1 requiring additional traversals to compile those functions.

Versus pointer-based representations The "Non-Packed" approach is LoCal configured to always insert indirections and thus emulate a traditional object representation. In this case, we are being overly friendly to this pointer-based representation by allowing it to read its input (for example, the input tree to **treeInsert**) in an already-deserialized, pointer-based form. A full apples-to-apples comparison would force the pointer-based version to deserialize the input message and reserialize the output; but we omit that here to focus only on the cost of the tree traversals themselves.

Versus competing libraries The biggest differences in Table 2 are due asymptotic complexity. However, for constant factor performance, we see the expected relationship—that our approach and Gibbon are faster than CNF and Cap'N Proto, sometimes by an order of magnitude, e.g.,**add1Leaves**.

CNF and Cap'N Proto encode some metadata in their serialization, to support the GHC runtime, and protocol evolution, respectively. On the other hand, our compiler only uses offsets and tagged indirections when needed, and the size ratio of the encodings depends on how much these features are used. For example, **rightmost** uses a data-encoding that includes random-access offsets, and **treeInsert** creates an output with a logarithmic number of tagged indirections. Thus while our size advantage over CNF is 4× smaller on **buildTree**, it is only 2.22× for **rightmost**.

[5]starting at 64K bytes



(a) Count "cats" hashtags from disk. Relative slowdown (resp. speedup) of approaches, normalized to our compiler.

	Gibbon2	CNF	Cap'n Proto
Size	257MB	2117MB	735MB

(b) Sizes for 1000MB of Twitter JSON data, translated to other formats.

Figure 8. Twitter data IO experiment

Composing traversals For offset-insertion, we allow the whole-program compiler to select the data representation based on what consuming functions are applied to it. In the presence of multiple functions traversing a single data structure, any function demanding random access changes the representation for all of them. **repMax** is one such example: `repMax t = propagateConst (findMax t) t`. Here **findMax** only requires a partial scan (random access), but propagating that value requires a full traversal. In this case, the compiler would add offsets to the datatype to ensure that 'findMax' remains logarithmic. However, this causes the subsequent traversal (propagateConst) to slow down, as it now has to unnecessarily skip over some extra bytes. Likewise, if we do not include findMax in the whole program, the data remains fully serialized, which is why **propagateConst** and **findMax** run separately take less than 440ms, but run together take 480ms. Yet the latter time is still 6× and 9× faster, respectively, than CNF and Cap'N Proto!

7.3 IO-intensive Benchmarks

The previous section examined benchmarks on data already in memory, but ultimately we want to minimize end-to-end latency to process data from disk or the network. Thus, in this section, we compare the cost of processing serialized data stored on disk, as well as the serialized space usage on disk. For a real data set, we use Twitter metadata consisting of user ID's and hashtags for all tweets posted in 1 month, and count the occurrences of the hashtag "cats" in this dataset. Here we seek to replicate and extend the CNF experiment reported by Yang et al. [33].

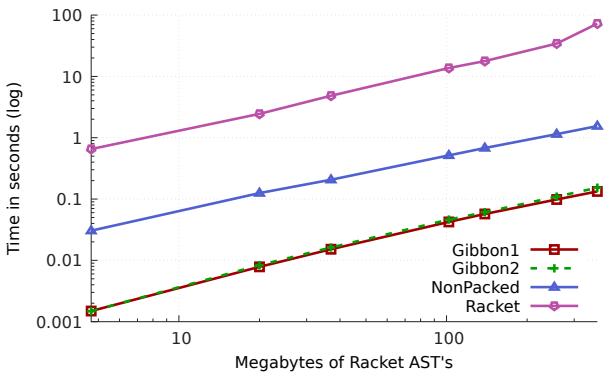


Figure 9. Count the nodes in a Racket AST

The dataset is stored on disk in JSON format, and we use RapidJSON v1.1.0 (http://rapidjson.org/) as a performance baseline: a widely recognized fast C++ JSON library. In Fig. 8a, we vary the amount of data processed, up to 1GB. (For each data-point, taking the median of 9 trials ensures the data is already in the Linux disk cache.) For fairness, all versions read the data via a single `mmap` call, plus demand paging.

There are two RapidJSON versions. The "lexer" version never constructs an object representing a parsed tweet, rather, it is a state-machine that is able to count "cats" while tokenizing, *without parsing*. It is optimized to be as fast as possible for this particular JSON schema, with no error detection (a non-compliant input would give silent failures and wrong answers). The "parser" version represents a more traditional and idiomatic situation use of the library: calling the `.Parse()` method to produce a DOM object, and then accessing its fields. We have structured this benchmark to maximally advantage this parsing approach: the 9,111,741 tweets processed in the rightmost data points of Fig. 8a are stored as one JSON object each, on each line of the input file. Thus the data only needs to be read into memory once, and in a single pass the RapidJSON benchmark reads, parses, discards, and repeats. Conversely, if the tweets were instead stored as a single JSON array, filling the entire input file, then RapidJSON would have to parse the entire file (writing the DOM tree out to memory, overflowing last level cache), then read that same tree back into memory in a second pass to count hashtags. Nevertheless, in spite of this single-pass advantage, our compiler achieves 6× and 12× speedup over RapidJSON lexer/parser. We process the 9.1M tweets in 0.39s.

For non-JSON implementations (Gibbon1, Gibbon2, CNF, and Cap'n Proto) we store the serialized data on disk in its respective binary format, without indirections. Of course, if the data originates in JSON or another format, a conversion and caching layer will be needed to convert it (once) to the efficient format. As shown in Fig. 8b there is a significant difference in the sizes of the serialized data, when converting 1GB of JSON data. Our approach is smaller and more than twice as fast at processing all tweets than either CNF or Cap'N Proto. Gibbon1 is slightly faster due to its huge regions, lack of bounds checking, and fewer switch cases (especially given the small number of switch cases in this data schema).

Table 2. Tree-processing functions operating on serialized data. Each cell contains time, complexity, and input bytes [1] or output bytes [2], where appropriate. The fastest variant is highlighted, as well as any that are an order of magnitude or more *slower* than the fastest. We are 202 / 2.6 / 3.2 / 17.9 × geomean faster than Gibbon/NonPacked/CNF/CapNP, and 0.96 / 2.6 / 3.2 / 9.8 × faster for only apples-to-apples asymptotics.

Benchmark	Gibbon2	Gibbon1	NoPacked	CNF	CapProto
id	2.1ns O(1)	0.32s O(N)	**0.93ns** **O(1)**	2.1ns O(1)	204ns O(1)
leftmost [1]	**17ns** **O(log(N))** **335MB**	18ns O(log(N)) 335MB	26ns O(log(N)) 335MB	44ns O(log(N)) 1.34GB	420ns O(log(N)) 1.61GB
rightmost [1]	175ns O(log(N)) 603MB	56ms O(N) 335MB	**19ns** **O(log(N))** **335MB**	47ns O(log(N)) 1.34GB	561ns O(log(N)) 1.61GB
buildTree [2]	0.27s O(N) 335MB	**0.24s** **O(N)** **335MB**	2.7s O(N) 1.34GB	4.5s O(N) 1.34GB	2.66s O(N) 1.61GB
add1leaves	0.25s O(N)	**0.24s** **O(N)**	3.1s O(N)	2.7s O(N)	4.88s O(N)
sumTree	95ms O(N)	**67ms** **O(N)**	0.81s O(N)	0.27s O(N)	1.22s (ON)
copyTree	**0.2s** **O(N)**	0.24s O(N)	3.5s O(N)	1.1s O(N)	2.53s O(N)
srchTree [2]	0.5s O(N) 603MB	**0.49s** **O(N)** **603MB**	2.96s O(N) 1.61GB	4.27s O(N) 1.61GB	3.94s O(N) 1.61GB
treeContains	**0.69µs** **O(log(N))**	0.1s O(N)	0.92µs O(log(N))	1µs O(log(N))	1.25µs O(log(N)
treeInsert [3]	**0.87µs** **O(log(N))** **677 bytes**	0.38s O(N) 603MB	2.5µs O(log(N)) 856 bytes	3.5µs O(log(N) 848 bytes	2.24s O(N) 1.61GB
insertDestr	NA	NA	NA	NA	**1.72µs** **O(log(N))**
findMax	206ns O(log(N))	88ms O(N)	**41ns** **O(log(N))**	75ns O(log(N))	3.94µs O(log(N))
propConst	0.43s O(N)	**0.42s** **O(N)**	3.3s O(N)	4.2s O(N)	4.83s O(N)
repMax	**0.48s** **O(N)**	0.51s O(N)	3.2s O(N)	4.3s O(N)	4.88s O(N)

7.4 IO + large datatype: Traversing Racket ASTs

Finally, as a second IO-intensive benchmark, we consider reading a tree with a more complicated type (9 mutually recursive types with 36 data constructors). Fig. 9 shows the result of reading the full ASTs used internally in the Racket compiler, and then counting AST nodes.

Using our approach, or Gibbon, or our approach with full indirections ("NonPacked") is vastly faster than the native and idiomatic way a Racket programmer would ingest this data. Racket's optimized `read` function (implemented in C inside the runtime) takes 71.1 seconds to parse a 366MB S-expression file, and then 0.94s to traverse it in memory. Our compiler reads and processes the data in its binary format (100MB) in 131ms. Thus it is 543× faster than the idiomatic method of data processing in the original language, and 7.1× faster even if data deserialization and IO are completely discounted. Also, here Gibbon loses its small advantage over our approach due to the already large size of the switch statements produced (36 constructor variants).

8 Conclusions & Future Work

When the purpose of a program is to process external data, we should consider alternate implementation approaches that **transform the code to bring it closer to the data**, rather than the other way around. This can speed IO by eschewing parsing/deserialization, speed traversals once in memory, and, with care, can optimize fast cases without introducing unpredictably (asymptotically) slow cases.

We believe that this work opens up significant follow-on possibilities. First, a LoCal implementation can become more representation-flexible to directly support appropriate external data formats such as Apache Avro or CBOR.

Second there is plenty of room to grow the size of the functional language supported by a $HiCal \rightarrow LoCal$ compiler, for example, into a much larger subset of Haskell. Integrating task-parallelism one direction, and the *mostly* serialized representations supported by LoCal suggest a connection to parallel- vs sequential-processing in a granularity-management strategy. Mutation is another frontier, where a traditional approach to mutable data (as found in imperative languages or `IO` monads) would appear to clash with the needs of dense representations. We plan to employ the Linear Haskell extensions [3] to introduce a limited capability for mutable data, exposing a more context-sensitive notion of where and when mutation may occur within a tree, and also striving to retain (deterministic) parallelism.

Acknowledgments

This work was supported in part by National Science Foundation awards CCF-1725672 and CCF-1725679, and by Department of Energy award DE-SC0010295. We would like to thank our shepherd, Ilya Sergey, as well as the anonymous reviewers for their suggestions and comments.

References

[1] Malcolm Atkinson and Ronald Morrison. 1995. Orthogonally Persistent Object Systems. *The VLDB Journal* 4, 3 (July 1995), 319–402. http://dl.acm.org/citation.cfm?id=615224.615226

[2] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. 1996. An Orthogonally Persistent Java. *SIGMOD Rec.* 25, 4 (Dec. 1996), 68–75. https://doi.org/10.1145/245882.245905

[3] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-order Polymorphic Language. *Proc. ACM Program. Lang.* 2, POPL, Article 5 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158093

[4] R. S. Bird. 1984. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Inf.* 21, 3 (Oct. 1984), 239–250. https://doi.org/10.1007/BF00264249

[5] TM Chilimbi, MD Hill, and JR Larus. 1999. Cache-conscious structure layout. *ACM SIGPLAN Notices* (1999). http://dl.acm.org/citation.cfm?id=301633

[6] Trishul M. Chilimbi and James R. Larus. 1999. Using generational garbage collection to implement cache-conscious data placement. , 37–48 pages. https://doi.org/10.1145/301589.286865

[7] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream fusion: from lists to streams to nothing at all. In *ICFP: International Conference on Functional Programming*. ACM.

[8] Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. 2013. General transformations for GPU execution of tree traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing) (SC '13)*.

[9] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based memory management in Cyclone. In *PLDI*. http://dl.acm.org/citation.cfm?id=512563

[10] Antony L. Hosking and J. Eliot B. Moss. 1993. Object Fault Handling for Persistent Programming Languages: A Performance Evaluation. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '93)*. ACM, New York, NY, USA, 288–303. https://doi.org/10.1145/165854.165907

[11] Chris Lattner and Vikram Adve. 2005. Automatic pool allocation: improving performance by controlling data structure layout in the heap. *ACM SIGPLAN Notices* 40 (2005), 129–142. https://doi.org/10.1145/1065010.1065027

[12] Chris Lattner and Vikram S. Adve. 2005. Transparent Pointer Compression for Linked Data Structures. In *Proceedings of the 2005 Workshop on Memory System Performance (MSP '05)*. ACM, New York, NY, USA, 24–35. https://doi.org/10.1145/1111583.1111587

[13] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. 1996. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD '96)*. ACM, New York, NY, USA, 318–329. https://doi.org/10.1145/233269.233346

[14] Junichiro Makino. 1990. Vectorization of a treecode. *J. Comput. Phys.* 87 (March 1990), 148–160. https://doi.org/10.1016/0021-9991(90)90231-O

[15] Conor McBride. 2010. Ornamental algebras, algebraic ornaments. *Journal of functional programming* (2010).

[16] Leo A. Meyerovich, Todd Mytkowicz, and Wolfram Schulte. 2011. Data Parallel Programming for Irregular Tree Computations, In HotPAR. https://www.microsoft.com/en-us/research/publication/data-parallel-programming-for-irregular-tree-computations/

[17] Leo A. Meyerovich, Todd Mytkowicz, and Wolfram Schulte. 2011. Data Parallel Programming for Irregular Tree Computations, In HotPAR. https://www.microsoft.com/en-us/research/publication/data-parallel-programming-for-irregular-tree-computations/

[18] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1998. From System F to Typed Assembly Language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*. ACM, New York, NY, USA, 85–97. https://doi.org/10.1145/268946.268954

[19] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. 2012. High-performance Complex Event Processing over XML Streams. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 253–264. https://doi.org/10.1145/2213836.2213866

[20] Ryan R. Newton, Sivan Toledo, Lewis Girod, Hari Balakrishnan, and Samuel Madden. 2009. Wishbone: Profile-based partitioning for sensornet applications. In *Symposium on Networked Systems Design and Implementation (NSDI'09)*. USENIX Association, 395–408.

[21] Bin Ren, Gagan Agrawal, James R. Larus, Todd Mytkowicz, Tomi Poutanen, and Wolfram Schulte. 2013. SIMD parallelization of applications that traverse irregular data structures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*. IEEE Computer Society, 20:1–20:10. https://doi.org/10.1109/CGO.2013.6494989

[22] Bin Ren, Todd Mytkowicz, and Gagan Agrawal. 2014. A Portable Optimization Engine for Accelerating Irregular Data-Traversal Applications on SIMD Architectures. *TACO* 11, 2 (2014), 16:1–16:31. https://doi.org/10.1145/2632215

[23] Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. Destination-passing Style for Efficient Memory Management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing (FHPC 2017)*. ACM, New York, NY, USA, 12–23. https://doi.org/10.1145/3122948.3122949

[24] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction*. Springer-Verlag.

[25] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher Order Symbol. Comput.* 17, 3 (Sept. 2004), 245–265. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

[26] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Inf. Comput.* 132, 2 (Feb. 1997), 109–176. https://doi.org/10.1006/inco.1996.2613

[27] D. N. Truong, F. Bodin, and A. Seznec. 1998. Improving Cache Behavior of Dynamically Allocated Data Structures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*. IEEE Computer Society, Washington, DC, USA, 322–. http://portal.acm.org/citation.cfm?id=522344.825680

[28] Kenton Varda. 2015. Cap'n Proto. https://capnproto.org/

[29] Michael Vollmer, Chaitanya Koparkar, , Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. 2019. *LoCal: A Language for Programs Operating on Serialized Data*. Technical Report. Indiana University. https://help.sice.indiana.edu/techreports/TRNNN.cgi?trnum=TR741.

[30] Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan R. Newton. 2017. Compiling Tree Transforms to Operate on Packed Representations. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 26:1–26:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.26

[31] P. Wadler. 1988. Deforestation: Transforming Programs to Eliminate Trees. In *European Symposium on Programming*. Berlin: Springer-Verlag, 344–358. citeseer.ist.psu.edu/wadler90deforestation.html

[32] Thomas Williams and Didier Rémy. 2017. A Principled Approach to Ornamentation in ML. *Proc. ACM Program. Lang.* 2, POPL, Article 21 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158109

[33] Edward Z. Yang, Giovanni Campagna, Ömer S. Ağacan, Ahmed El-Hassany, Abhishek Kulkarni, and Ryan R. Newton. 2015. Efficient Communication and Collection with Compact Normal Forms. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 362–374. https://doi.org/10.1145/2784731.2784735

A Evaluation Details

A.1 Benchmark Descriptions

In §7.1, we mentioned a series of benchmark programs that we would use to evaluate our implementation, most of which have self-evident names (like **sumTree**), but for full transparency we will enumerate each benchmark and give any relevant details about their different implementations in differing systems we compare.

- **id**: the simplest test, an identity function ($\lambda x.\, x$). For this benchmark, `id` must be called as a function, not inlined, and return a valid message. In a general-purpose language, we simply return the input message in $O(1)$ time, but Gibbon can only compile this function as a deep copy! We are also limited by our type system, forced to keep output region distinct from input, but we can write a tagged indirection, pointing back to the input message.
- **buildTree**: build a binary tree of given height. We use height 25, creating trees with 2^{25} leaves and $2^{25} - 1$ interior nodes (likewise for the rest of the benchmarks in this group).
- **rightmost**: return the rightmost leaf in the binary tree.
- **add1Leaves**: map an add1 function over the tree, returning a fresh tree in the output message.
- **sumTree**: sum the leaves of the tree.
- **copyTree**: copy the input message to a fresh buffer. For this benchmark we require no sharing of data with the input message, but we allow whatever form of copying performs best for the given system. For example, it is permitted to simply `memcpy` the message. Lacking such a low-level primitive, the Gibbon approach and ours require a tree-walk (similar to **add1Leaves**). CNF allows a memcpy style operation, but it uses absolute pointers, and the "pointer fixup" pass makes it slightly slower than just doing a tree-walk, so we use a tree-walk. Cap'N Proto does *not* provide low-level access to the memory segments holding the message (the API instead wants you to send the message to a file descriptor), but it does provide its own notion of an internal deep-copy (tree walk), which we use.

And the search tree benchmarks:

- **buildSearchTree**: build a balanced search tree in the output message, containing 2^{26} total nodes and max height 26 (including null nodes as separate nodes).
- **treeContains**: take the previous search tree as input and check if it contains a random key.
- **treeInsert**: take the same search tree as input and insert one new element at a random key, rebuilding a spine of the tree. That is, it may write a new piece of memory (contiguous with the input or not) containing only the $log(N)$ newly created nodes, but pointing back to the unchanged part of the tree, if that is supported. Because Cap'N Proto does not support sharing, it must copy the whole tree to insert into it. However, it will allow direct mutation of the tree inside the input message buffer, so we test this as a separate variant, **treeInsertDestructive**.
- **findMax**: return the largest element in the search tree.
- **propagateConst**: map over the tree and replace all values with a constant integer.
- **repMax**: a variant of the "repmin"[4, 17] program, which returns a new search tree of the same shape, with every value replaced by the *maximum* value of the original tree. It performs two passes over the tree – a bottom-up pass to compute the largest element (i.e. findMax), and a top-down pass to propagate this value in the tree (i.e. propagateConst).

B Related Work in Tree Layout and Allocation

There has been significant work in optimizing the layout and traversal patterns of tree data structures for performance reasons. The most closely related line of work is *cache-conscious structure layout* [5], which proposes a data layout scheme that lays out the nodes of a tree according to an order determined by a provided traversal function. Because this layout is determined by a specific traversal function, it serves a similar purpose to the linearization of data in our packed layout: when trees are traversed in the same manner as the layout order, spatial locality is improved. Note, however, that Chilimbi et al.'s approach does not change the internal structure of objects, nor the code that traverses those data structures. Hence, all pointers are preserved, and this approach does not offer the additional benefits of our packed layout such as denser accesses and avoidance of pointer indirection. Other spatial locality work [6, 11, 27] has similar effects and limitations to cache-conscious structure layout.

Lattner and Adve [11] propose *automatic pool allocation* that allocates disjoint data structures into disjoint partitions of memory. Leveraging this approach, Lattner and Adve [12] propose a compression optimization. Because a data structure is allocated into an isolated pool, "internal" pointers that connect nodes of that data structure definitely do not access arbitrary memory locations, and hence can use narrower bit widths to save space. Unlike the spatial locality work discusses in the previous paragraph, this compression optimization both shrinks the overall representation

of the data structure (as in our packed representation) and utilizes compiler rewrites to do so (as in our compiler transformations).

Tree linearization, and the attendant changes required to traversal code, are common in high-performance computing settings, especially for vectorization [8, 14, 16, 21, 22]. These approaches, by closely matching data structure layout to the traversal behavior of specific applications, can eliminate many pointer dereferences, compress data structures, and simplify traversal implementations. However, all of these approaches are programmer-directed: either *ad hoc*, application-specific implementations [8, 14, 16], or driven by library functions that the programmer must exploit [21, 22].

C LoCal Runtime System Details

In LoCal, locations track natural number positions within a region. Abstractly, a region is an unbounded, byte-indexed storage area that can be extended incrementally by requesting N additional bytes (equivalent to `malloc(N)`). Each region grows monotonically, never shrinks, and can be freed only as a whole. Practically speaking, there are at many reasonable implementation strategies. We always start by allocating a contiguous *chunk* of memory of bounded size. When that chunk is exhausted, we must choose whether to grow the region by **copying** (or changing memory-mapping), retaining a contiguous address range, or by linking a new, non-contiguous chunk.

We choose the latter and implement regions as a linked list of chunks: a constant sized initial chunk, with subsequent chunks doubling in size. The runtime representation of locations (and `Ptr T` values) is a direct pointer into the interior of a chunk. (We call the writable portion of the chunk that can carry data the *payload*.) Chunks linked together form regions as pictured in Fig. 10. Chunk metadata is stored at the *end* of the allocated area, in a footer data structure listed below:

```
struct footer {
  // Available bytes for serialized-data storage.
  int  size;
  // Shared reference count for this region (not chunk)
  int* refcount;
  // Set of regions we have outbound pointers into.
  ptrset  outset;
  // The chunk that follows this one (or NULL)
  footer* next;
}
```

We avoid additional indirection by combining this metadata struct with the payload, which is essentially an array of bytes, forming one heap object. The reason we store the metadata as a *footer*, at the end rather than the start, is so that the payload grows *towards* the struct. Thus the pointer to the region-chunk does double duty for bounds checking. When the payload space is full, we allocate a new chunk of double the size and point to it with `next`.

But what do we put in the serialized bytestream to mark that the stream continues in another chunk? Here we implicitly add a reserved tag to each packed data type, signaling *end of chunk* (EOC).[6] When the reader hits an EOC, they must use their pointer to the end of the current payload to access the footer, follow the `next` pointer, and resume reading at the head of the next chunk.

Garbage collection In most classic treatments, regions introduced with a `letregion`, are deallocated immediately upon the end of that `letregion`'s lexical scope. However, in this paper we choose to allow tagged indirection nodes to include **inter-region pointers**. Thus one can keep a region alive beyond the scope of the `letregion` that introduced it, by simply capturing a pointer to it within another region. This choice is critical to our ability to lift functions onto (mostly) serialized representations without changing their asymptotic complexity.

In our setting, pointers between regions are immutable, which simplifies the job of garbage collection. Rather than keeping a "remembered set" of inter-region pointers as in a generational collector, we can instead *coarsen* the dependencies to record only that "chunk A points to region B". The `outset` in the `footer` struct above tracks regions to which our chunk points[7].

[6] Of course, there are 256 possible one-byte tags, so adding indirections, random-access nodes, and EOC tags reduces the largest sum type supported (at least, without using an escape sequence to access additional tags).

[7] This set is optimized for zero or one elements. A null pointer denotes the empty set, and singleton is a direct (tagged) pointer to the element. Two or more elements introduce a heap data structure to store the out-set.

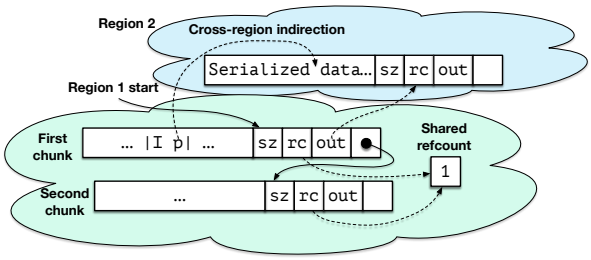


Figure 10. Example of multiple chunks making up a region, and of an inter-region indirection.

Both tracing or reference counting collectors would benefit from this coarsening. However, given that we already amortize the overheads of memory management through coarsening, we choose reference counting for our implementation to achieve prompt deallocation (and reuse) of chunks. Thus when a region is created with `letregion` its reference count is set to 1, and it is decremented on exit from the `letregion`. Reference counts are region-level rather than chunk-level, which is why the `footer` contains a pointer to the region-level reference count, rather than a reference count directly. When a region hits zero reference count, it is freed immediately via freeing its chunks one by one. When a chunk is deallocated, it decrements the reference count of any regions it points to.

Comparing against prior art's memory management Finally, in order to compare against other work, we also implement a technique for **huge regions**; these are allocated as a large slab containing many pages, and could be extended by mapping new virtual memory after hitting a guard page capping the region end. This was the approach used by Gibbon [30]. These huge regions avoid bounds checks when writing payloads and they are suitable for programs with a small number of large regions (especially a single input and single output region). But they are inappropriate for the more general case where programs may have small and short-lived regions.

The choice of allocation strategy can be informed by static information the compiler gathers about the lifetime and potential size range of the region. For example, the region-based MLKit compiler achieved significant speedups from statically classifying a majority of regions as constant-sized [25], in which case they are allocated inside the procedure stack frame. In our approach, we unbox constant-sized data types (e.g. numbers), and pack recursive data-types into growable regions, so we do not observe the same opportunity for constant-sized regions.

D Appendix: formalization and type-safety proof

This section contains:

- a complete version of the typing rules and dynamic semantics, along with an example to aid in the understanding of the dynamic semantics,
- metafunctions and correctness judgements for well formedness of the store
- and the complete proof of type safety.

Some of the rules from the type system and dynamic semantics that appear in this section do not appear in the main body of the paper, owing to space limitations. Moreover, to make the appendix somewhat self contained, we have in a few cases copies of rules given in the main body of the paper. In each case, we indicate what is new and what is copy.

D.1 Grammar for LoCal

The complete grammar of LoCal, along with specifications of environments and machine states used by the type system and dynamic semantics, appears in Figure 11.

Variables and Substitution We use the convention that all variables for binding values, locations, and regions are distinct, and maintain this invariant implicitly. The bindings sites of variables are summarized by the following:

- Variables for binding values x are bound by function definitions fd and pattern matches pat.
- Location variables l^r are bound by type schemes ts, pattern matches pat, and $letloc$ binders.
- Region variables r are bound by type schemes ts, pattern matches pat, and $letregion$ binders.

The use sites of variables are summarized by the following:

- Variables for binding varlues x are used by values v.
- Location variables l^r are used by concrete locations $\langle r, i\rangle^{l^r}$, the argument list of function applications $f\ [\overrightarrow{l^r}]\ \overrightarrow{v}$, the location argument of constructor applications $K\ l^r\ \overrightarrow{v}$, located types $\hat{\tau}$, and located expressions le.

$K \in$ Data Constructors, $\tau \in$ Type Constructors,
$x, y, f \in$ Variables, $l, l^r \in$ Symbolic Locations,
$r \in$ Regions, $i, j \in$ Region Indices,
$\langle r, i \rangle^l \in$ Concrete Locations

Top-Level Programs	top	$::= \overrightarrow{dd}\ ;\ \overrightarrow{fd}\ ;\ e$
Datatype Declarations	dd	$::= \texttt{data}\ \tau = \overrightarrow{K\ \overrightarrow{\tau}}$
Function Declarations	fd	$::= f : ts; f\ \overrightarrow{x} = e$
Located Types	$\hat{\tau}$	$::= \tau @ l^r$
Type Scheme	ts	$::= \forall_{\overrightarrow{l^r}}.\ \overrightarrow{\hat{\tau}} \to \hat{\tau}$
Values	v	$::= x \mid \langle r, i \rangle^{l^r}$
Expressions	e	$::= v$
		$\mid f\ [\overrightarrow{l^r}]\ \overrightarrow{v}$
		$\mid K\ l^r\ \overrightarrow{v}$
		$\mid \texttt{let}\ x : \hat{\tau} = e\ \texttt{in}\ e$
		$\mid \texttt{letloc}\ l^r = le\ \texttt{in}\ e$
		$\mid \texttt{letregion}\ r\ \texttt{in}\ e$
		$\mid \texttt{case}\ v\ \texttt{of}\ \overrightarrow{pat}$
Pattern	pat	$::= K\ \ (\overrightarrow{x : \hat{\tau}})\ \to\ e$
Location Expressions	le	$::= (\texttt{start}\ r)$
		$\mid (l^r + 1)$
		$\mid (\texttt{after}\ \hat{\tau})$

Typing Env.	Γ	$::= \{\ x_1 \mapsto \hat{\tau}_1,\ \ldots\ , x_n \mapsto \hat{\tau}_n\ \}$
Store Typing	Σ	$::= \{\ l_1^{r_1} \mapsto \tau_1,\ \ldots\ , l_n^{r_n} \mapsto \tau_n\ \}$
Constraint Env.	C	$::= \{\ l_1^{r_1} \mapsto le_1,\ \ldots\ , l_n^{r_n} \mapsto le_n\ \}$
Allocation Pointers	A	$::= \{\ r_1 \mapsto ap_1,\ \ldots\ , r_n \mapsto ap_n\ \}$
		where $ap = l^r \mid \varnothing$
Nursery	N	$::= \{\ l_1^{r_1},\ \ldots\ , l_n^{r_n}\ \}$

Store	S	$::= \{\ r_1 \mapsto h_1,\ \ldots\ , r_n \mapsto h_n\ \}$
Heap	h	$::= \{\ i_1 \mapsto K_1,\ \ldots\ , i_n \mapsto K_n\ \}$
Location Map	M	$::= \{\ l_1^{r_1} \mapsto \langle r_1, i_1 \rangle,\ \ldots\ , l_n^{r_n} \mapsto \langle r_n, i_n \rangle\ \}$

Figure 11. The grammar of LoCal. These definitions are copies of those shown in Figures 2, 3, and 5.

- Region varaibles r are used in the same places as location variables.

We use the following conventions for variable substitution:

- $e[v/x]$: Substitute v for x in e. We let the notation extend to vectors such that $e[\overrightarrow{v}/\overrightarrow{x}]$ denotes the iterated substitution $e[\overrightarrow{v_1}/\overrightarrow{x_1}] \ldots [\overrightarrow{v_n}/\overrightarrow{x_n}]$, where $n = |\overrightarrow{x}| = |\overrightarrow{v}|$.
- $e[{l_2}^{r_2}/{l_1}^{r_1}]$: Substitute location variable ${l_2}^{r_2}$ for ${l_1}^{r_1}$ in e. We extend this notation to vectors of locations in the same fashion, as described above.
- $e[r_2/r_1]$: Substitute region variable r_2 for r_1 in e. We extend this notation to vectors of locations in the same fashion, as described above.
- Finally, we extend the aforementioned notation so that substitution can act on environments C, A, and N, e.g., $C[{l_2}^{r_2}/{l_1}^{r_1}]$.

[T-Var]
$$\frac{\Gamma(x) = \tau@l^r \qquad \Sigma(l^r) = \tau}{\Gamma; \Sigma; C; A; N \vdash A; N; x : \tau@l^r}$$

[T-Concrete-Loc]
$$\frac{\Sigma(l^r) = \tau}{\Gamma; \Sigma; C; A; N \vdash A; N; \langle r, i\rangle^l : \tau@l^r}$$

[T-Let]
$$\frac{\Gamma; \Sigma; C; A; N \vdash A'; N'; e_1 : \tau_1@{l_1}^{r_1} \quad {l_1}^{r_1} \in N \quad {l_1}^{r_1} \notin N' \qquad \Gamma'; \Sigma'; C; A'; N' \vdash A''; N''; e_2 : \tau_2@{l_2}^{r_2} \quad {l_2}^{r_2} \in N}{\Gamma; \Sigma; C; A; N \vdash A''; N''; \texttt{let}\ x : \tau_1@{l_1}^{r_1} = e_1\ \texttt{in}\ e_2 : \tau_2@{l_2}^{r_2}}$$
where $\Gamma' = \Gamma \cup \{ x \mapsto \tau_1@{l_1}^{r_1} \}$; $\Sigma' = \Sigma \cup \{ {l_1}^{r_1} \mapsto \tau_1 \}$

[T-LetRegion]
$$\frac{\Gamma; \Sigma; C; A'; N \vdash A''; N'; e : \tau@{l'}^{r'} \quad {l'}^{r'} \in N}{\Gamma; \Sigma; C; A; N \vdash A''; N'; \texttt{letregion}\ r\ \texttt{in}\ e : \tau@{l'}^{r'}}$$
where $A' = A \cup \{ r \mapsto \varnothing \}$

[T-LetLoc-Tag]
$$\frac{A(r) = {l'}^{r} \qquad {l'}^{r}, {l''}^{r''} \in N \qquad l^r \notin N'' \qquad l^r \neq {l''}^{r''} \qquad \Gamma; \Sigma; C'; A'; N' \vdash A''; N''; e : \tau''@{l''}^{r''}}{\Gamma; \Sigma; C; A; N \vdash A''; N''; \texttt{letloc}\ l^r = ({l'}^{r} + 1)\ \texttt{in}\ e : \tau''@{l''}^{r''}}$$
where $C' = C \cup \{ l^r \mapsto ({l'}^{r} + 1) \}$; $A' = A \cup \{ r \mapsto l^r \}$; $N' = N \cup \{ l^r \}$

[T-LetLoc-After]
$$\frac{A(r) = {l_1}^{r} \qquad \Sigma({l_1}^{r}) = \tau' \qquad {l_1}^{r} \notin N \qquad l^r \notin N'' \qquad l^r \neq {l'}^{r'} \qquad \Gamma; \Sigma; C'; A'; N' \vdash A''; N''; e : \tau'@{l'}^{r'} \qquad {l'}^{r'} \in N}{\Gamma; \Sigma; C; A; N \vdash A''; N''; \texttt{letloc}\ l^r = (\texttt{after}\ \tau'@{l_1}^{r})\ \texttt{in}\ e : \tau'@{l'}^{r'}}$$
where $C' = C \cup \{ l^r \mapsto (\texttt{after}\ \tau'@{l_1}^{r}) \}$; $A' = A \cup \{ r \mapsto l^r \}$; $N' = N \cup \{ l^r \}$

[T-DataConstructor]
$$\frac{\mathit{TypeOfCon}(K) = \tau \qquad \mathit{TypeOfField}(K, i) = \overrightarrow{\tau'_i} \qquad l^r \in N \qquad A(r) = \overrightarrow{l_n^{\,r}} \ \text{if } n \neq 0 \ \text{ else } l^r \qquad C(\overrightarrow{l_1^{\,r}}) = l^r + 1 \qquad C(\overrightarrow{l_{j+1}^{\,r}}) = (\texttt{after}\ (\overrightarrow{\tau'_j@l_j^{\,r}})) \qquad \Gamma; \Sigma; C; A; N \vdash A; N; \overrightarrow{v_i} : \overrightarrow{\tau'_i@l_i^{\,r}}}{\Gamma; \Sigma; C; A; N \vdash A'; N'; K\ l^r\ \overrightarrow{v} : \tau@l^r}$$
where $A' = A \cup \{ r \mapsto l^r \}$; $N' = N - \{ l^r \}$
$n = |\overrightarrow{v}|$; $i \in I = \{ 1, \ldots, n \}$; $j \in I - \{ n \}$

Figure 12. Copy of typing rules in Figure 4

D.2 Typing rules for LoCal

The complete typing rules are contained in two figures. In Figure 12, there is a copy of the typing rules in Figure 4. The rules not shown in the main body, namely those for function application, pattern matching, function definition, and program top-level, are given in Figure 13.

Function application in T-App ensures the location of the result of the application is initially unwritten, and is considered written afterward. Types and locations for the function are pulled from the function signature. Pattern matching is handled by T-Case and T-Pattern, which are straightforward. Similarly, typing the top-level forms is straightforward.

To simplify the formalism and proofs, we restricted typing rules somewhat so that, in effect, the rules restrict well-typed expressions so that they can return only the the result of a freshly allocated constructor application. Consequently, it is not possible, for instance, to type the following expression, because the right-hand side is a value and, as such, does not allocate.

```
let x : T @ l^r = y in  ···
```

This restriction is enforced by there being an assertion of the form $l^r \in N$ in the premise of the typing rules of the non-value expressions, such that $\tau@l^r$ is the result type of the given expression. Lifting this restriction is conceptually straightforward, but would require either added complexity to the substitution lemma or could be achieved by using a different factoring of the grammar and typing rules. Similarly, our formalism and proofs could be extended to treat primitive types, such as ints, bools, tuples, etc., as well as with offsets and indirections in data constructors, with some conceptually straightforward extensions to the formalism.

[T-LetLoc-Start]

$$\frac{A(r) = \varnothing \qquad l^r \notin N'' \qquad l'^{r'} \neq l^r \qquad l'^{r'} \in N \qquad \Gamma; \Sigma; C'; A'; N' \vdash A''; N''; e : \tau'@l'^{r'}}{\Gamma; \Sigma; C; A; N \vdash A''; N''; \texttt{letloc}\ l^r = (\texttt{start}\ r)\ \texttt{in}\ e : \tau'@l'^{r'}}$$

$$\text{where}\ \ C' = C \cup \{\, l^r \mapsto (\texttt{start}\ r) \,\};\ A' = A \cup \{\, r \mapsto l^r \,\};\ N' = N \cup \{\, l^r \,\}$$

[T-App]

$$\frac{\begin{array}{c} |\overrightarrow{l'^{r'}}| = |\overrightarrow{l''^{r''}}| \quad |\overrightarrow{v}| = |\overrightarrow{x}| \\ \Gamma; \Sigma; C; A; N \vdash A; N; \overrightarrow{v_i} : \overrightarrow{\tau_i@l_i^{r_i}} \qquad l^r \in N \qquad A(r) = l^r \\ \forall_i.\exists_j.\overrightarrow{l_i'''^{r_i'''}} = \overrightarrow{l_j''^{r_j''}} \wedge \overrightarrow{l_i^{r_i}} = \overrightarrow{l_j'^{r_j'}} \quad \exists_j.l'''^{r'''} = \overrightarrow{l_j''^{r_j''}} \wedge l^r = \overrightarrow{l_j'^{r_j'}} \end{array}}{\Gamma; \Sigma; C; A; N \vdash A; N'; f\ [\overrightarrow{l'^{r'}}]\ \overrightarrow{v} : \tau@l^r}$$

$$\text{where}\ \ f : \forall_{\overrightarrow{l''^{r''}}}.\overrightarrow{\tau_i@l_i'''^{r_i'''}} \to \tau@l'''^{r'''};\ (f\ \overrightarrow{x} = e) = \mathit{Function}(f)$$
$$N' = N - \{\, l^r \,\};\ n = |\overrightarrow{v}|;\ i \in \{1, \ldots, n\}$$

[T-Pattern]

$$\frac{\begin{array}{c} \mathit{TypeOfCon}(K) = \tau'' \qquad \mathit{ArgTysOfConstructor}(K) = \overrightarrow{\tau'} \qquad \Sigma(l^r) = \tau \\ l^r \neq \overrightarrow{l_i'^{r'}} \qquad \Gamma'; \Sigma'; C; A; N \vdash A'; N'; e : \tau@l^r \end{array}}{\tau''; \Gamma; \Sigma; C; A; N \vdash_{pat} A'; N'; K\ \ \overrightarrow{(x : \tau'@l'^{r'})}\ \to\ e : \tau@l^r}$$

$$\text{where}\ \ \Gamma' = \Gamma \cup \{\, \overrightarrow{x_1} \mapsto \overrightarrow{\tau_1'@l_1'^{r'}},\ \ldots,\ \overrightarrow{x_n} \mapsto \overrightarrow{\tau_n'@l_n'^{r'}} \,\}$$
$$\Sigma' = \Sigma \cup \{\, \overrightarrow{l_1'^{r'}} \mapsto \overrightarrow{\tau_1'},\ \ldots,\ \overrightarrow{l_n'^{r'}} \mapsto \overrightarrow{\tau_n'} \,\}$$
$$i \in \{1, \ldots, n\};\ n = |\overrightarrow{\tau'}| = |\overrightarrow{x : \tau'@l'^{r}}|$$

[T-Case]

$$\frac{\begin{array}{c} \Gamma; \Sigma; C; A; N \vdash A; N; v : \tau'@l'^{r'} \qquad l^r \in N \\ \tau'; \Gamma; \Sigma; C; A; N \vdash_{pat} A'; N'; \overrightarrow{pat_i} : \tau@l^r \end{array}}{\Gamma; \Sigma; C; A; N \vdash A'; N'; \texttt{case}\ v\ \texttt{of}\ \overrightarrow{pat} : \tau@l^r}$$

$$\text{where}\ \ n = |\overrightarrow{pat}|;\ i \in \{1, \ldots, n\}$$

[T-Function-Definition]

$$\frac{\begin{array}{c} \Gamma; \Sigma; C; A; N \vdash A; N'; e : \tau@l^r \qquad l^r \notin N' \\ \forall_{i \in \{1,\ldots,n\}}.\exists_j.\overrightarrow{l_i^{r_i}} = \overrightarrow{l_j'^{r_j'}} \qquad \exists_j.l^r = \overrightarrow{l_j'^{r_j'}} \end{array}}{\vdash_{fun} f : \forall_{\overrightarrow{l'^{r'}}}.\overrightarrow{\tau@l^r} \to \tau@l^r;\ f\ \overrightarrow{x} = e}$$

$$\text{where}\ \ \Gamma = \{\, \overrightarrow{x_1} \mapsto \overrightarrow{\tau_1@l_1^{r_1}},\ \ldots,\ \overrightarrow{x_n} \mapsto \overrightarrow{\tau_n@l_n^{r_n}} \,\}$$
$$\Sigma = \{\, \overrightarrow{l_1^{r_1}} \mapsto \overrightarrow{\tau_1},\ \ldots,\ \overrightarrow{l_n^{r_n}} \mapsto \overrightarrow{\tau_n} \,\}$$
$$C = \varnothing;\ A = \{\, r \mapsto l^r \,\};\ N = \{\, l^r \,\};$$
$$n = |\overrightarrow{x}| = |\overrightarrow{\tau@l^r}|$$

[T-Program]

$$\frac{\vdash_{fun} \overrightarrow{fd} \qquad \Gamma; \Sigma; C; A; N \vdash A'; N'; e : \tau@l^r}{\vdash_{prog} A'; N'; \overrightarrow{dd}\ ; \overrightarrow{fd}\ ; e : \tau@l^r}$$

$$\text{where}\ \ \Gamma = \varnothing;\ \Sigma = \varnothing$$
$$C = \{\, l^r \mapsto (\texttt{start}\ r) \,\};\ A = \{\, r \mapsto l^r \,\};\ N = \{\, l^r \,\}$$

Figure 13. Remaining typing judgements for LoCal

D.3 Dynamic semantics rules for LoCal

Figures 14 and 15 complete dynamic semantics of LoCal. In the following, we explain the rules shown in Figure 15, which do not appear in the main body of the paper. The D-Let-Expr rule for let-expressions evaluates the let-bound expression to a value and the D-Let-Val rule substitutes the value for the let-bound variable in the body. The D-App rule for function applications looks up the function by name in the top-level environment and substitutes arguments for parameters in the function body, substitutes argument symbolic locations for parameter symbolic locations, then starts the resulting function body running. The D-LetRegion rule for the `letregion` expression binds the new region and starts running the body.

The driver which runs an LoCal program initially loads all data types, functions, type checks them, and if successful, then seeds the *Function*, *TypeOfCon*, and *TypeOfField* environments. Let e_0 be the main expression. If e_0 type checks with respect to the T-Program rule, then the main program is safe to run. The initial configuration for the

[D-DataConstructor]
$S; M; K\ l^r\ \overrightarrow{v} \Rightarrow S'; M; \langle r, i\rangle^{l^r}$
where $S' = S \cup \{\, r \mapsto (i \mapsto K)\,\};\ \langle r, i\rangle = M(l^r)$

[D-LetLoc-Start]
$S; M; \texttt{letloc}\ l^r = (\texttt{start}\ r)\ \texttt{in}\ e \Rightarrow S; M'; e$
where $M' = M \cup \{\, l^r \mapsto \langle r, 0\rangle\,\}$

[D-LetLoc-Tag]
$S; M; \texttt{letloc}\ l^r = l'^r + 1\ \texttt{in}\ e \Rightarrow S; M'; e$
where $M' = M \cup \{\, l^r \mapsto \langle r, i+1\rangle\,\};\ \langle r, i\rangle = M(l'^r)$

[D-LetLoc-After]
$S; M; \texttt{letloc}\ l^r = (\texttt{after}\ \tau@{l_1}^r)\ \texttt{in}\ e \Rightarrow S; M'; e$
where $M' = M \cup \{\, l^r \mapsto \langle r, j\rangle\,\};\ \langle r, i\rangle = M({l_1}^r)$
$\tau; \langle r, i\rangle; S \vdash_{ew} \langle r, j\rangle$

[D-Case]
$S; M; \texttt{case}\ \langle r, i\rangle^{l^r}\ \texttt{of}\ [\ldots, K\ \ (\overrightarrow{x : \tau@l^r})\ \rightarrow\ e, \ldots] \Rightarrow$
$S; M'; e[\langle r, \overrightarrow{w}\rangle^{\overrightarrow{l^r}} / \overrightarrow{x}]$
where $M' = M \cup \{\, \overrightarrow{l_1^r} \mapsto \langle r, i+1\rangle, \ldots, \overrightarrow{l_{j+1}^r} \mapsto \langle r, \overrightarrow{w_{j+1}}\rangle\,\}$
$\overrightarrow{\tau_1}; \langle r, i+1\rangle; S \vdash_{ew} \langle r, \overrightarrow{w_1}\rangle$
$\overrightarrow{\tau_{j+1}}; \langle r, \overrightarrow{w_j}\rangle; S \vdash_{ew} \langle r, \overrightarrow{w_{j+1}}\rangle$
$K = S(r)(i);\ j \in \{\, 1, \ldots, n-1\,\};\ n = |\overrightarrow{x : \hat{\tau}}|$

Figure 14. Copy of dynamic semantics shown in Figure 6.

[D-Let-Expr]
$$\frac{S; M; e_1 \Rightarrow S'; M'; e_1' \qquad e_1 \neq v}{S; M; \texttt{let}\ x : \hat{\tau} = e_1\ \texttt{in}\ e_2 \Rightarrow S'; M'; \texttt{let}\ x : \hat{\tau} = e_1'\ \texttt{in}\ e_2}$$

[D-Let-Val]
$S; M; \texttt{let}\ x : \hat{\tau} = v_1\ \texttt{in}\ e_2 \Rightarrow S; M; e_2[v_1/x]$

[D-App]
$S; M; f\ [\overrightarrow{l^r}]\ \overrightarrow{v} \Rightarrow S; M; e[\overrightarrow{v}/\overrightarrow{x}][\overrightarrow{l^r}/\overrightarrow{l'^{r'}}]$
where $fd = Function(f)$
$f : \forall_{\overrightarrow{l'^{r'}}}.\overrightarrow{\hat{\tau}_f} \rightarrow \hat{\tau}_f; (f\ \overrightarrow{x} = e) = Freshen(fd)$

[D-LetRegion]
$S; M; \texttt{letregion}\ r\ \texttt{in}\ e \Rightarrow S; M; e$

Figure 15. Remaining dynamic semantics for LoCal

machine with an empty store is

$$\varnothing; \{\, l \mapsto \langle r, 0\rangle\,\}; e_0,$$

which is, by itself, not particularly interesting or useful. It is, however, straightforward to construct a type-safe initial configuration whose store is nonempty, as long as the initial configuration has a store that is well formed, as described in §D.5. The program can start taking evaluation steps from this configuration.

D.3.1 Example: allocating a binary tree.

Consider this code snippet of LoCal.

```
letloc l1^r = l0^r + 1 in
let a : Tree @ l1^r = (Leaf l1^r) in
letloc l2^r = (after (Tree @ l1^r)) in
let b : Tree @ l2^r = (Leaf l2^r) in
Node l0^r a b
```

Assume that the store starts out with a fresh heap, $S = \{\, r \mapsto \varnothing\,\}$ and the location ${l_0}^r$ maps to $\langle r, 0\rangle$ in the location map. After stepping past the first line, the D-LetLoc-Tag step has allocated a cell for the tag of the interior node and bound the location ${l_1}^r$ to $\langle r, 1\rangle$. After the next line, the D-DataConstructor transition writes a leaf node to the store at the address represented by ${l_1}^r$: $S = \{\, r \mapsto \{\, 1 \mapsto \texttt{Leaf}\,\}\,\}$. The second `letloc` obtains the starting address for the second leaf node by using end witness of the previous leaf node. The write of the second leaf node appears in the store after the next line, leaving the following store: $S = \{\, r \mapsto \{\, 1 \mapsto \texttt{Leaf}, 2 \mapsto \texttt{Leaf}\,\}\,\}$. Finally, after the D-DataConstructor step taken for the last line, the store contains the finalized allocation: $S = \{\, r \mapsto \{\, 0 \mapsto \texttt{Node}, 1 \mapsto \texttt{Leaf}, 2 \mapsto \texttt{Leaf}\,\}\,\}$.

Table 3. Summary of judgements used to establish well formedness of the store.

	Judgement form	Section	Summary
Store well formedness	$\Sigma; C; A; N \vdash_{wf} M; S$	D.5	The store S along with location map M are well formed with respect to typing environments Σ, C, and A.
End witness	$\tau; \langle r, i_s \rangle; S \vdash_{ew} \langle r, i_e \rangle$	D.5.1	The store address $\langle r, i_e \rangle$ is the position one after the last cell of the tree of type τ starting at $\langle r, i_s \rangle$ in store S.
Constructor-application well formedness	$C \vdash_{wf_{cfc}} M; S$	D.5.2	All in-flight data-constructor applications in store S along with location map M are well formed with respect to constructor-progress typing environment C.
Allocation well formedness	$A; N \vdash_{wf_{ca}} M; S$	D.5.3	Allocation in store S along with location map M is well formed with respect to allocation-typing environments A and N.

The end-witness judgement of the new data constructor is the following: $\texttt{Tree}; \langle r, 0 \rangle; S \vdash_{ew} \langle r, 3 \rangle$ The judgement applies, in part, because, as expected, the tag at the address $\langle r, 0 \rangle$ is a tag of type `Tree`. In addition, because the tag indicates an interior node with two subtrees for fields, the judgement obligation extends to recursively showing (1) that the end witness of the first leaf node (also at type `Tree`) at $\langle r, 1 \rangle$ has an end witness (which is $\langle r, 2 \rangle$), (2) that the second field has an end witness starting at the end witness of the first field, namely $\langle r, 2 \rangle$, and ending at some higher address (which in this case is $\langle r, 3 \rangle$), and (3) finally that the end witness of the second field is the end witness of the entire constructor, as is the case here.

D.4 Global environments and metafunctions

- $Function(f)$: An environment that maps a function f to its definition fd.
- $Freshen(fd)$: A metafunction that freshens all bound variables in function definition fd and returns the resulting function definition.
- $TypeOfCon(K)$: An environment that maps a data constructor to its type.
- $TypeOfField(K, i)$: A metafunction that returns the type of the `i`'th field of data constructor K.
- $ArgTysOfConstructor(K)$: An environment that maps a data constructor to its field types.
- $MaxIdx(r, S)$: $\max\{-1\} \cup \{ j \mid (r \mapsto (j \mapsto K)) \in S \}$.

D.5 Well-formedness of the Store

The well formedness of the store is defined by the top-level judgement

$$\Sigma; C; A; N \vdash_{wf} M; S$$

whose definition itself uses three other judgements. All of these judgements are summarized in Table 3.

Notation for references to well-formedness judgements Because there are many requirements specified inside the various well-formedness judgements, we introduce notation for referring to requirements individually. For example, the notation WF D.5.3;2 refers to the judgement

$$A; N \vdash_{wf_{ca}} M; S,$$

specified in Section D.5.3, and in that judgement, rule number 2.

The definition of store well formedness follows.

Judgement form $\Sigma; C; A; N \vdash_{wf} M; S$

The well-formedness judgement specifies the valid layouts of the store by using the location map and the various environments from the typing judgement. Rule 1 specifies that, for each location in the store-typing environment, there is a corresponding concrete location in the location map and that concrete location satisfies a corresponding end-witness judgement. Rules 2 and 3 reference the judgements for well formedness concerning in-flight constructor applications (§D.5.1) and correct allocation in regions (§D.5.3), respectively. Finally, Rule 4 specifies that the nursery and store-typing environments reference no common locations, which is a way of reflecting that each location is either in the process of being constructed and in the nursery, or allocated and in the store-typing environment, but never both.

Definition

1 $(l^r \mapsto \tau) \in \Sigma \Rightarrow$
$((l^r \mapsto \langle r, i_1 \rangle) \in M \wedge$
$\tau; \langle r, i_1 \rangle; S \vdash_{ew} \langle r, i_2 \rangle)$
2 $C \vdash_{wf_{cfc}} M; S$
3 $A; N \vdash_{wf_{ca}} M; S$
4 $dom(\Sigma) \cap N = \varnothing$

D.5.1 End-Witness judgement

Judgement form $\tau; \langle r, i_s \rangle; S \vdash_{ew} \langle r, i_e \rangle$

The end-witness judgement specifies the expected layout in the store of a fully allocated data constructor. Rule 1 requires that the first cell store a constructor tag of the appropriate type. Rule 3 specifies the address of the cell one past the tag. Rule 4 recursively specifies the positions of the constructor fields. Finally, Rule 2 specifies that the end witness of the overall constructor is the address one past the end of either the tag, if the constructor has zero fields, or the final field, otherwise.

Definition

1 $S(r)(i_s) = K'$ such that
$\texttt{data}\ \tau = \overrightarrow{K_1\ \overrightarrow{\tau}_1} \mid \ldots \mid K'\ \overrightarrow{\tau}' \mid \ldots \mid \overrightarrow{K_m\ \overrightarrow{\tau}_m}$
2 $\overrightarrow{w_0} = i_s + 1$
3 $\overrightarrow{\tau'_1}; \langle r, \overrightarrow{w_0} \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_1} \rangle \wedge$
$\overrightarrow{\tau'_{j+1}}; \langle r, \overrightarrow{w_j} \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_{j+1}} \rangle$
where $j \in \{1, \ldots, n-1\}; n = |\overrightarrow{\tau'}|$
4 $i_e = \overrightarrow{w_n}$

D.5.2 Well-formedness of constructor application

Judgement form $C \vdash_{wf_{cfc}} M; S$

The well-formedness judgement for constructor application specifies the various constraints that are necessary for ensuring correct formation of constructors, dealing with constructor application being an incremental process that spans multiple LoCal instructions. Rule 1 specifies that, if a location corresponding to the first address in a region is in the constraint environment, then there is a corresponding entry for that location in the location map. Rule 2 specifies that, if a location corresponding to the address one past a constructor tag is in the constraint environment, then there are corresponding locations for the address of the tag and the address after in the location map. Rule 3 specifies that, if a location corresponding to the address one past after a fully allocated constructor application is in the constraint environment, then there are corresponding locations for the address one past the constructor application and for the address of the start of that constructor application in the location map, as well as the existence of an end witness for that fully allocated location.

Definition

1 $(l^r \mapsto (\texttt{start}\ r)) \in C \Rightarrow$
$(l^r \mapsto \langle r, 0 \rangle) \in M$

2 $(l^r \mapsto (l'^r + 1)) \in C \Rightarrow$
$(l'^r \mapsto \langle r, i_l \rangle) \in M \wedge$
$(l^r \mapsto \langle r, i_l + 1 \rangle) \in M$
3 $(l^r \mapsto (\texttt{after}\ \tau @ l'^{r\,r})) \in C \Rightarrow$
$((l'^r \mapsto \langle r, i_1 \rangle) \in M \wedge$
$\tau; \langle r, i_1 \rangle; S \vdash_{ew} \langle r, i_2 \rangle \wedge$
$(l^r \mapsto \langle r, i_2 \rangle) \in M)$

D.5.3 Well-formedness concerning allocation

Judgement form $A; N \vdash_{wf_{ca}} M; S$

The well-formedness judgement for safe allocation specifies the various properties of the location map and store that enable continued safe allocation, avoiding in particular overwriting cells, which could, if possible, invalidate overall type safety. Rule 1 requires that, if a location is in both the allocation and nursery environments, i.e., that address represents an in-flight constructor application, then there is a corresponding location in the location map and the address of that location is the highest address in the store. Rule 2 requires that, if there is an address in the allocation environment and that address is fully allocated, then the address of that location is the highest address in the store. Rule 3 requires that, if there is an address in the nursery, then there is a corresponding location in the location map, but nothing at the corresponding address in the store. Finally, Rule 4 requires that, if there is a region that has been created but for which nothing has yet been allocated, then there can be no addresses for that region in the store.

Definition

1 $((r \mapsto l^r) \in A \wedge l^r \in N) \Rightarrow$
$((l^r \mapsto \langle r, i \rangle) \in M \wedge i > MaxIdx(r, S))$
2 $((r \mapsto l^r) \in A \wedge (l^r \mapsto \langle r, i_s \rangle) \in M \wedge l^r \notin N \wedge \tau; \langle r, i_s \rangle; S \vdash_{ew} \langle r, i_e \rangle) \Rightarrow$
$i_e > MaxIdx(r, S)$
3 $l^r \in N \Rightarrow$
$((l^r \mapsto \langle r, i \rangle) \in M \wedge$
$(r \mapsto (i \mapsto K)) \notin S)$
4 $(r \mapsto \varnothing) \in A \Rightarrow$
$r \notin dom(S)$

D.6 Technical lemmas

Lemma D.1 (Substitution lemma)

$$
\begin{aligned}
\textit{If} \quad & \Gamma \cup \{\, \overrightarrow{x_1} \mapsto \overrightarrow{\tau_1} @ \overrightarrow{l_1^{\,r_1}}, \ldots, \overrightarrow{x_n} \mapsto \overrightarrow{\tau_n} @ \overrightarrow{l_n^{\,r_n}} \,\}; \Sigma; C; A; N \vdash A'; N'; e : \tau @ l^r \\
\textit{and} \quad & \Gamma; \Sigma'; C'; A'; N' \vdash A'; N'; \overrightarrow{v_i} : \overrightarrow{\tau_i} @ \overrightarrow{l_i'^{\,r_i'}} \qquad i \in \{\, 1, \ldots, n \,\} \\
\textit{then} \quad & \Gamma; \Sigma'; C'; A'; N' \vdash A'''; N'''; e[\,\overrightarrow{v}/\overrightarrow{x}\,][\overrightarrow{l'^{\,r'}}/\overrightarrow{l^r}][l'^{\,r'}/l^r] : \tau @ l'^{\,r'} \\
\textit{where} \quad & \Sigma = \Sigma_0 \cup \{\, \overrightarrow{l_1^{\,r_1}} \mapsto \overrightarrow{\tau_1}, \ldots, \overrightarrow{l_n^{\,r_n}} \mapsto \overrightarrow{\tau_n} \,\} \\
\textit{and} \quad & \forall_{(x \mapsto \tau'' @ l''^{\,r''}) \in \Gamma} . (l''^{\,r''} \mapsto \tau'') \in \Sigma_0 \\
\textit{and} \quad & dom(\Sigma) \cap N = \varnothing \\
\textit{and} \quad & N = N_0 \cup l^r \\
\textit{and} \quad & \Sigma' = \Sigma \cup \{\, \overrightarrow{l_1'^{\,r_1'}} \mapsto \overrightarrow{\tau_1}, \ldots, \overrightarrow{l_n'^{\,r_n'}} \mapsto \overrightarrow{\tau_n} \,\} \\
\textit{and} \quad & C' = C[\overrightarrow{l'^{\,r'}}/\overrightarrow{l^r}][l'^{\,r'}/l^r] \\
\textit{and} \quad & A' = A[\overrightarrow{l'^{\,r'}}/\overrightarrow{l^r}][l'^{\,r'}/l^r][r'/r] \\
\textit{and} \quad & N' = N[l'^{\,r'}/l^r]
\end{aligned}
$$

Proof The proof is by rule induction on the given typing derivation.

Case T-Var, T-Concrete-Loc
These cases discharge vacuously because the corresponding typing judgements cannot establish that the expression e has type $\tau@l^r$, as required by the premise of the lemma. The reason is that the premise of the lemma also assumes that $l^r \in N$ and $dom(\Sigma) \cap N = \varnothing$, but by inversion on the respective typing judgements, it must be that $(l^r \mapsto \tau) \in \Sigma$, thereby resulting in a contradiction.

Case

$$\frac{\begin{array}{c}[\text{T-DataConstructor}] \\ TypeOfCon(K) = \tau \qquad TypeOfField(K, i) = \overrightarrow{\tau'_i} \\ l^r \in N \qquad A(r) = \overrightarrow{l_n{}^r} \quad \text{if } n \neq 0 \quad \text{else } l^r \\ C(\overrightarrow{l_1{}^r}) = l^r + 1 \qquad C(\overrightarrow{l_{j+1}{}^r}) = (\texttt{after}\ (\overrightarrow{\tau'_j}@\overrightarrow{l'_j{}^r})) \\ \Gamma; \Sigma; C; A; N \vdash A; N; \overrightarrow{v_i} : \overrightarrow{\tau'_i@l_i{}^r}\end{array}}{\Gamma; \Sigma; C; A; N \vdash A'; N'; K\ l^r\ \overrightarrow{v} : \tau@l^r}$$

where $A' = A \cup \{ r \mapsto l^r \};\ N' = N - \{ l^r \}$
$n = |\overrightarrow{v}|;\ i \in I = \{ 1, \ldots, n \};\ j \in I - \{ n \}$

By inversion on the typing judgement, there are three proof obligations for this case. The first one concerns the subtitution of location l^r, which changes the type of the term e from $\tau@l^r$ to $\tau@l'^{r'}$. The specific obligation is to establish that all uses of l^r in the typing judgement are properly substituted by $l'^{r'}$, thereby satisfying the corresponding parts of the typing judgement that need to reflect the change in the result location. The uses of l^r in the typing judgement are the first argument of the constructor application, the result type, the constraint environment C, and environments A, N, A', and N'. The corresponding updates are established by inspection of the various substitutions in the consequent of the lemma, which affect e and the typing environments. The second obligation concerns the locations used by the typing judgement in C, each of which is substituted as needed in the environment C'.

The third and final obligation is to establish typing judgements required by the premise of T-DataConstructor that concern the arguments of the constructor application. To distinguish the constructor arguments from the values $\overrightarrow{v}$ that are being substituted, let the constructor arguments be $\overrightarrow{v'}$, and $m = |\overrightarrow{v'}|$. Then the specific obligation is to establish the typing judgements

$$\Gamma; \Sigma'; C'; A'; N' \vdash A'; N'; \overrightarrow{v'_k}[\overrightarrow{v}/\overrightarrow{x}][\overrightarrow{l'^{r'}}/\overrightarrow{l^r}][l'^{r'}/l^r] : \overrightarrow{\tau'_k@l''_k{}^r},$$

for all $k \in \{1, \ldots, m\}$, and for some suitable corresponding locations $l''_k{}^r$. Each value $\overrightarrow{v'_k}$ is either a variable or a concrete location.

- Case $\overrightarrow{v'_k} = y$, for some variable y:
  - Case $y = \overrightarrow{x_j}$, for some j:
    Now, the obligation is to establish that the value resulting from the substitution of y, namely $\overrightarrow{v_j}$, has type $\overrightarrow{\tau'_k}@\overrightarrow{l'^r_j}$. From the premise of the lemma, we have that
    $$\Gamma; \Sigma'; C'; A'; N' \vdash A'; N'; \overrightarrow{v_j} : \overrightarrow{\tau_j}@\overrightarrow{l'^r_j},$$
    and, moreover, by inversion on T-DataConstructor, we can conclude that $\overrightarrow{\tau_j} = \overrightarrow{\tau'_k}$, thereby establishing that
    $$\Gamma; \Sigma'; C'; A'; N' \vdash A'; N'; \overrightarrow{v_j} : \overrightarrow{\tau'_k}@\overrightarrow{l'^r_j},$$
    and thus discharging this case.
  - Case $y \neq \overrightarrow{x_j}$, for all j:
    This case discharges immediately by implication of the typing judgement of the source term given in the premise of this lemma, and by inversion on T-Var.

- Case $\overrightarrow{v'_k} = \langle r, i'''\rangle^{l'''^r}$, for some location l'''^r, i'''
  - Case $l'''^r = \overrightarrow{l_j^r}$, for some j:
    The specific obligation is to establish the type of the concrete location affected by the substitution of the location l'''^r for $\overrightarrow{l_j'^r}$, that is,
    $$\Gamma; \Sigma'; C'; A'; N' \vdash A'; N'; \langle r, i'''\rangle^{\overrightarrow{l_j'^r}} : \overrightarrow{\tau'_k}@\overrightarrow{l_j'^r}.$$
    The above follows from the facts $\Sigma'(\overrightarrow{l_j'^r}) = \overrightarrow{\tau_j}$ and $\overrightarrow{\tau_j} = \overrightarrow{\tau'_k}$, using similar reasoning to the previous case, thus discharging this case.
  - Case $l'''^r = l^r$:
    Impossible, because $l'''^r \in dom(\Sigma)$, but from the premise of this lemma, $l^r \in N$ and $dom(\Sigma) \cap N = \varnothing$.
  - Case $l'''^r \neq \overrightarrow{l_j^r}$, for all j, and $l'''^r \neq l^r$:
    This case discharges straightforwardly because, by inversion on T-Concrete-Loc, $(l'''^r \mapsto \tau'') \in \Sigma$, thus implying that $(l'''^r \mapsto \tau'') \in \Sigma'$, as needed.

Case

$$\frac{\begin{array}{c}\text{[T-Let]}\\ \Gamma; \Sigma; C; A; N \vdash A'; N'; e_1 : \tau_1@{l_1}^{r_1} \quad {l_1}^{r_1} \in N \quad {l_1}^{r_1} \notin N' \\ \Gamma'; \Sigma'; C; A'; N' \vdash A''; N''; e_2 : \tau_2@{l_2}^{r_2} \quad {l_2}^{r_2} \in N\end{array}}{\Gamma; \Sigma; C; A; N \vdash A''; N''; \texttt{let}\ x : \tau_1@{l_1}^{r_1} = e_1\ \texttt{in}\ e_2 : \tau_2@{l_2}^{r_2}}$$
$$\text{where } \Gamma' = \Gamma \cup \{ x \mapsto \tau_1@{l_1}^{r_1} \};\ \Sigma' = \Sigma \cup \{ {l_1}^{r_1} \mapsto \tau_1 \}$$

This case discharges via straightforward uses of the induction hypothesis for the let-bound expression and the body.

Case T-LetRegion, T-LetLoc-Start, T-LetLoc-Tag, T-LetLoc-After, T-App, T-Case
These remaining cases discharge by similar uses of the induction hypothesis.

■

Lemma D.2 (Progress)

$$\begin{array}{rl} & \textit{if } \varnothing; \Sigma; C; A; N \vdash A'; N'; e : \hat{\tau} \\ & \textit{and } \Sigma; C; A; N \vdash_{wf} M; S \\ & \textit{then } e \textit{ value} \\ & \textit{else } S; M; e \Rightarrow S'; M'; e' \end{array}$$

Proof The proof is by rule induction on the given typing derivation.
Case

$$\frac{\begin{array}{c}\text{[T-DataConstructor]}\\ \mathit{TypeOfCon}(K) = \tau \qquad \mathit{TypeOfField}(K, i) = \overrightarrow{\tau'_i} \\ l^r \in N \qquad A(r) = \overrightarrow{l_n^r} \quad \text{if } n \neq 0 \quad \text{else } l^r \\ C(\overrightarrow{l_1^r}) = l^r + 1 \qquad C(\overrightarrow{l_{j+1}^r}) = (\texttt{after}\ (\overrightarrow{\tau'_j}@\overrightarrow{l_j'^r})) \\ \Gamma; \Sigma; C; A; N \vdash A; N; \overrightarrow{v_i} : \overrightarrow{\tau'_i@l_i^r}\end{array}}{\Gamma; \Sigma; C; A; N \vdash A'; N'; K\ l^r\ \overrightarrow{v} : \tau@l^r}$$
$$\text{where } A' = A \cup \{ r \mapsto l^r \};\ N' = N - \{ l^r \}$$
$$n = |\overrightarrow{v}|;\ i \in I = \{ 1, \ldots, n \};\ j \in I - \{ n \}$$

Because $e = K\ l^r\ \overrightarrow{v}$ is not a value, the proof obligation is to show that there is a rule in the dynamic semantics whose left-hand side matches the machine configuration $S; M; e$. The only rule that can match is D-DataConstructor, but to establish the match, there remains one obligation, which is obtained by inversion on D-DataConstructor. The particular obligation is to establish that $\langle r, i\rangle = M(l^r)$, for some i. To obtain this result, we need to use the well formedness of the store, given by the premise of this lemma, and in particular rule WF D.5.3;3. But a precondition for using WF D.5.3;3 that the location is in the nursery, i.e., $l^r \in N$. This precondition is satisfied by inversion on T-DataConstructor. Our application of rule WF D.5.3;3 therefore yields the desired result, thereby discharging this case.

Case

$$\frac{\begin{array}{c}[\text{T-LetLoc-After}]\\ A(r) = {l_1}^r \qquad \Sigma({l_1}^r) = \tau' \qquad {l_1}^r \notin N \qquad l^r \notin N'' \qquad l^r \neq {l'}^{r'}\\ \Gamma; \Sigma; C'; A'; N' \vdash A''; N''; e : \tau'@{l'}^{r'} \qquad {l'}^{r'} \in N\end{array}}{\begin{array}{l}\Gamma; \Sigma; C; A; N \vdash A''; N''; \texttt{letloc}\ l^r = (\texttt{after}\ \tau'@{l_1}^r)\ \texttt{in}\ e : \tau'@{l'}^{r'}\\ \quad \text{where}\ \ C' = C \cup \{\, l^r \mapsto (\texttt{after}\ \tau'@{l_1}^r)\,\};\ A' = A \cup \{\, r \mapsto l^r\,\};\\ \qquad\qquad N' = N \cup \{\, l^r\,\}\end{array}}$$

Because $e = \texttt{letloc}\ l^r = (\texttt{after}\ \tau'@{l_1}^r)\ \texttt{in}\ e'$ is not a value, the proof obligation is to show that there is a rule in the dynamic semantics whose left-hand side matches the machine configuration $S; M; e$. The only rule that can match is D-LetLoc-After, but the match is dependent on two further obligations, which are implied by inversion on D-LetLoc-After. The first one is to establish that $\langle r, i\rangle = M({l_1}^r)$. To do so, we need to use rule WF D.5;1 of the well-formedness of the store. This rule requires that $\Sigma({l_1}^r) = \tau'$, which is established by inversion on T-LetLoc-After. As such, we have $(l_1 \mapsto \langle r, i\rangle) \in M$, as needed. The second and final obligation is to establish that, for some j, $\tau'; \langle r, i\rangle; S \vdash_{ew} \langle r, j\rangle$. Again, we use well-formedness rule WF D.5;1 to discharge the obligation, and thus this case.

Case T-LetLoc-Tag
Similar to the previous case.

Case T-LetLoc-Start, T-LetRegion, T-App
These cases discharge immediately because D-LetLoc-Start, D-LetRegion, and D-App match their corresponding machine configurations unconditionally.

Case T-Var, T-Concrete-Loc
These cases discharge immediately because e is a value.

Case

$$\frac{\begin{array}{c}[\text{T-Let}]\\ \Gamma; \Sigma; C; A; N \vdash A'; N'; e_1 : \tau_1@{l_1}^{r_1} \qquad {l_1}^{r_1} \in N \qquad {l_1}^{r_1} \notin N'\\ \Gamma'; \Sigma'; C; A'; N' \vdash A''; N''; e_2 : \tau_2@{l_2}^{r_2} \qquad {l_2}^{r_2} \in N\end{array}}{\begin{array}{l}\Gamma; \Sigma; C; A; N \vdash A''; N''; \texttt{let}\ x : \tau_1@{l_1}^{r_1} = e_1\ \texttt{in}\ e_2 : \tau_2@{l_2}^{r_2}\\ \quad \text{where}\ \ \Gamma' = \Gamma \cup \{\, x \mapsto \tau_1@{l_1}^{r_1}\,\};\ \Sigma' = \Sigma \cup \{\, {l_1}^{r_1} \mapsto \tau_1\,\}\end{array}}$$

Because $e = \texttt{let}\ x : \tau_1@{l_1}^{r_1} = e_1\ \texttt{in}\ e_2$ is not a value, the proof obligation is to show that there is a rule in the dynamics whose left-hand side matches the machine configuration $S; M; e$. If e_1 is a value, then the rule discharges immediately because D-Let-Val matches e unconditionally. Otherwise, if e_1 is not a value, then the only other rule that can match is D-Let-Expr. To match D-Let-Expr, the only requirement is to match the left-hand side of the rule $S; M; e_1 \Rightarrow S'; M'; e_1'$ in the premise, for some S', M', and e_1'. To obtain this result, we

need to use the induction hypothesis, which is in this instance

$$\begin{aligned} &\text{if } \varnothing; \Sigma; C; A; N \vdash A'; N'; e_1 : \tau @ {l_1}^{r_1} \\ &\text{and } \Sigma; C; A; N \vdash_{wf} M; S \\ &\text{then } e_1 \textit{ value} \\ &\text{else } S; M; e_1 \Rightarrow S'; M'; e_1'. \end{aligned}$$

By inversion on T-Let, we have $\varnothing; \Sigma; C; A; N \vdash A'; N'; e_1 : \tau_1 @ {l_1}^{r_1}$, and, from the premise of this lemma, we have $\Sigma; C; A; N \vdash_{wf} M; S$. Thus, by the consequent of the induction hypothesis, we have that either e_1 is a value (which we have already ruled out) or that $S; M; e_1 \Rightarrow S'; M'; e_1'$, thereby discharging this case.

Case

$$\frac{\begin{array}{c}\text{[T-Case]} \\ \Gamma; \Sigma; C; A; N \vdash A; N; v : \tau' @ {l'}^{r'} \qquad l^r \in N \\ \tau'; \Gamma; \Sigma; C; A; N \vdash_{pat} A'; N'; \overrightarrow{pat_i} : \tau @ l^r\end{array}}{\Gamma; \Sigma; C; A; N \vdash A'; N'; \texttt{case}\ v\ \texttt{of}\ \overrightarrow{pat} : \tau @ l^r}$$

$$\text{where } n = |\overrightarrow{pat}|;\ i \in \{1, \ldots, n\}$$

and

$$\frac{\begin{array}{c}\text{[T-Pattern]} \\ \mathit{TypeOfCon}(K) = \tau'' \qquad \mathit{ArgTysOfConstructor}(K) = \overrightarrow{\tau'} \qquad \Sigma(l^r) = \tau \\ l^r \neq \overrightarrow{l_i'^{r'}} \qquad \Gamma'; \Sigma'; C; A; N \vdash A'; N'; e : \tau @ l^r\end{array}}{\tau''; \Gamma; \Sigma; C; A; N \vdash_{pat} A'; N'; K\ \overrightarrow{(x : \tau' @ {l'}^{r'})} \ \rightarrow\ e : \tau @ l^r}$$

$$\begin{aligned} \text{where } \Gamma' &= \Gamma \cup \{ \overrightarrow{x_1} \mapsto \overrightarrow{\tau_1'} @ \overrightarrow{l_1'^{r'}}, \ \ldots, \overrightarrow{x_n} \mapsto \overrightarrow{\tau_n'} @ \overrightarrow{l_n'^{r'}} \} \\ \Sigma' &= \Sigma \cup \{ \overrightarrow{{l_1'}^{r'}} \mapsto \overrightarrow{\tau_1'}, \ \ldots, \overrightarrow{{l_n'}^{r'}} \mapsto \overrightarrow{\tau_n'} \} \\ i &\in \{1, \ldots, n\};\ n = |\overrightarrow{\tau'}| = |\overrightarrow{x : \tau' @ l'^{r}}| \end{aligned}$$

Because the given expression $e = \texttt{case}\ v\ \texttt{of}\ \overrightarrow{pat}$ is not a value, the proof obligation is to show that there is a rule in the dynamic semantics whose left-hand side matches the machine configuration $S; M; e$. The only rule that can match is D-Case, and there are three requirements to match D-Case. The first of which is that the value v is a concrete location of the form $\langle r', i \rangle^{l'^{r'}}$. Any value v is, by inspection of the grammar of LoCal, either a variable or a concrete location. But because v is well typed with respect to the empty typing environment $\Gamma = \varnothing$, the value v cannot be a variable in this instance, owing to inversion on T-Var and T-Concrete-Loc, thereby ensuring v is a concrete location, and thus discharging this requirement. The second requirement for D-Case is that the tag is in the expected location in the store, i.e., $S(r')(i) = K$. To satisfy this requirement, we start by using the jugement $\Sigma; C; A; N \vdash_{wf} M; S$, from the premise of this lemma, and in particular, unpacking from this judgement the property EW D.5.1;1. To use this property, we need that $({l'}^{r'} \mapsto \tau') \in \Sigma$, which is given by inversion on the given typing rule T-Case. From the unpacking, we obtain that

$$({l'}^{r'} \mapsto \langle r', i \rangle \in M) \wedge \tag{1}$$

$$\tau'; \langle r', i \rangle; S \vdash_{ew} \langle r', i' \rangle. \tag{2}$$

From the end-witness judgement, in particular, EW D.5.1;1, we establish that $S(r')(i) = K$, thereby discharging the second requirement. The third and final requirement for D-Case is that the arguments succeeding the tag are in the expected locations, i.e.,

$$\begin{aligned} &\overrightarrow{\tau_1'}; \langle r', i+1 \rangle; S \vdash_{ew} \langle r', \overrightarrow{w_1} \rangle \wedge \\ &\overrightarrow{\tau_{j+1}'}; \langle r', \overrightarrow{w_j} \rangle; S \vdash_{ew} \langle r', \overrightarrow{w_{j+1}} \rangle \end{aligned}$$

The above is established by expanding the judgement obtained in 2, namely $\tau'; \langle r', i\rangle; S \vdash_{ew} \langle r', i'\rangle$, using in particular, the end-witness rule EW D.5.1;3 to obtain the needed judgements. This final requirement discharges the case.

■

Lemma D.3 (Preservation)

$$\begin{aligned} &\textit{If } \varnothing; \Sigma; C; A; N \vdash A'; N'; e : \hat{\tau} \\ &\textit{and } \Sigma; C; A; N \vdash_{wf} M; S \\ &\textit{and } S; M; e \Rightarrow S'; M'; e' \\ &\textit{then for some } \Sigma' \supseteq \Sigma, C' \supseteq C, \\ &\qquad \varnothing; \Sigma'; C'; A'; N' \vdash A''; N''; e' : \hat{\tau} \\ &\textit{and } \Sigma'; C'; A'; N' \vdash_{wf} M'; S' \end{aligned}$$

Proof The proof is by rule induction on the given derivation of the dynamic semantics.
Case

$$\begin{aligned} &[\text{D-DataConstructor}] \\ &\qquad S; M; K\ l^r\ \overrightarrow{v} \Rightarrow S'; M; \langle r, i\rangle^{l^r} \\ &\text{where } S' = S \cup \{\, r \mapsto (i \mapsto K)\,\};\ \langle r, i\rangle = M(l^r) \end{aligned}$$

- The first of two proof obligations is to show that the result $e' = \langle r, i\rangle^{l^r}$ of the given step of evaluation is well typed, that is,
$$\varnothing; \Sigma'; C'; A'; N' \vdash A''; N''; \langle r, i\rangle^{l^r} : \hat{\tau},$$
where $\hat{\tau} = \tau @ l^r$. As implied by inversion on T-Concrete-Loc, the only obligation is to establish that $\Sigma'(l^r) = \tau$. This obligation discharges by appropriately instantiating typing environments: $\Sigma' = \Sigma \cup \{\, l^r \mapsto \tau\,\}$, so that $\Sigma' \supseteq \Sigma$ and $\Sigma'(l^r) = \tau$, and $C' = C$, so that $C' \supseteq C$.
- Given the instantiations of Σ' and C' used by the previous step, the second obligation for this proof case is to show that
$$\Sigma'; C; A'; N' \vdash_{wf} M; S'.$$
The individual requirements, labeled WF D.5;1 - WF D.5;3, are handled by the following case analysis.
  - Case (WF D.5;1): for each $(l'^{r'} \mapsto \tau) \in \Sigma'$, there exists some i_1, i_2 such that
$$(l'^{r'} \mapsto \langle r', i_1\rangle) \in M \wedge \tag{3}$$
$$\tau; \langle r', i_1\rangle; S' \vdash_{ew} \langle r', i_2\rangle \tag{4}$$
The first conjunct above discharges by inversion on D-DataConstructor, but to establish the second one, we need to distinguish between the case in which the given location $l'^{r'}$ is the one affected by the constructor application, or not.
    * Case $l'^{r'} = l^r$:
For this case, the obligation is to show that the constructor being allocated by the constructor application, namely l^r, has the end witness given above. As such, for this case, it is the case that $r' = r$ and $i_1 = i$, which is a consequence of inversion on D-DataConstructor. To establish the end witness, the first obligation therein, namely EW D.5.1;1, is to establish $S'(r)(i) = K$. This obligation discharges by inspection of S', which is obtained by inversion on D-DataConstructor. The second part is to establish the requirement EW D.5.1;3 of the end-witness judgement, which pertains to the arguments passed to the constructor. The specific obligation is, if $n = |\overrightarrow{\tau'}| \geq 1$, then
$$\overrightarrow{\tau'_1}; \langle r, i+1\rangle; S' \vdash_{ew} \langle r, \overrightarrow{w_1}\rangle \wedge \tag{5}$$
$$\overrightarrow{\tau'_{j+1}}; \langle r, \overrightarrow{w_j}\rangle; S' \vdash_{ew} \langle r, \overrightarrow{w_{j+1}}\rangle \tag{6}$$
for some $\overrightarrow{w}$, where $j \in J = J' - \{\, n\,\}$, $j' \in J' = \{\, 1, \ldots, n\,\}$, and $\overrightarrow{\tau'} = \mathit{ArgTysOfConstructor}(K)$. To establish the above, we need to reason backward from what the corresponding typing rules establish regarding the

arguments passed to the constructor application. First, we establish that, for each location corresponding to a constructor argument $\overrightarrow{l_{j'}^r}$, there is a corresponding mapping in the store-typing environment, i.e., $(\overrightarrow{l_{j'}^r} \mapsto \overrightarrow{\tau'_{j'}}) \in \Sigma$. To establish these mappings, we first obtain by inversion on T-DataConstructor that the constructor arguments are well typed:

$$\varnothing; \Sigma; C; A; N \vdash A; N; \overrightarrow{v_{j'}} : \overrightarrow{\tau'_{j'} @ l_{j'}{}^r}$$

Each value $\overrightarrow{v_{j'}}$ is either a variable or a concrete location, and as such, by inversion on the typing rules T-Var and T-Concrete-Loc, respectively, we establish the required mappings in Σ. Thus, we can now combine the well-formedness of the store in the premise of this lemma, in particular requirement WF D.5;1, with the mappings of constructor arguments in Σ to establish the end witnesses in $\overrightarrow{i}$ corresponding to the constructor arguments:

$$(\overrightarrow{l_{j'}^r} \mapsto \langle r, \overrightarrow{i_{j'}} \rangle) \in M \wedge \tag{7}$$

$$\overrightarrow{\tau'_{j'}}; \langle r, \overrightarrow{i_{j'}} \rangle; S \vdash_{ew} \langle r, \overrightarrow{i_{j'+1}} \rangle \tag{8}$$

We first address the obligation pertaining to the first constructor argument, and then the remaining ones. From inversion on T-DataConstructor, we establish a mapping for the location of the first constructor argument.

$$C(\overrightarrow{l_1^r}) = l^r + 1$$

Now, using this result, we can establish from well formedness rule WF D.5.2;2 that the following mappings exist in the location environment.

$$(l^r \mapsto \langle r, i \rangle) \in M \wedge$$
$$(\overrightarrow{l_1^r} \mapsto \langle r, i+1 \rangle) \in M$$

Next, combining the fact from line 7 above regarding $\overrightarrow{l_1^r}$, the end witness corresponding to $\overrightarrow{i_1}$ from the end witnesses of constructor arguments line 8 from above, we establish the requirement on line 5 above, such that $\overrightarrow{w_1} = \overrightarrow{i_1}$, i.e.,

$$\overrightarrow{\tau'_1}; \langle r, i+1 \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_1} \rangle. \tag{9}$$

For the remaining constructor arguments, the structure of the proof is similar. We establish mappings in C for the locations of these constructor arguments by inversion on T-DataConstructor.

$$C(\overrightarrow{l_{j+1}^r}) = (\texttt{after}\ \overrightarrow{\tau'_j} @ \overrightarrow{l_j^r})$$

The following end witnesses $\overrightarrow{i}$ are established by combining the property on the constraint environment with the property WF D.5.2;3, which is obtained from the well formedness of the store in the premise of this lemma.

$$((\overrightarrow{l_j^r} \mapsto \langle r, \overrightarrow{i_j} \rangle) \in M \wedge$$
$$\overrightarrow{\tau'_j}; \langle r, \overrightarrow{i_j} \rangle; S \vdash_{ew} \langle r, \overrightarrow{i_{j+1}} \rangle \wedge$$
$$(\overrightarrow{l_{j+1}^r} \mapsto \langle r, \overrightarrow{i_{j+1}} \rangle) \in M)$$

To isolate the indices of any constructor arguments succeeding the first argument, we let $j'' \in J - \{1\}$, and thus deduce from the above that the end witnesses

$$\overrightarrow{\tau'_{j''+1}}; \langle r, \overrightarrow{i_{j''+1}} \rangle; S \vdash_{ew} \langle r, \overrightarrow{i_{j''+2}} \rangle.$$

exist. We obtain the needed result for the remaining end witnesses by instantiating for $\overrightarrow{w}$, yielding

$$\overrightarrow{\tau'_{j''+1}}; \langle r, \overrightarrow{w_{j''}} \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_{j''+1}} \rangle. \tag{10}$$

The original end witness required by 4 is now established by letting $i_1 = i$ and $i_2 = \overrightarrow{w_{n+1}}$.
Finally, to discharge this case, the end witnesses of the constructor arguments established in lines 9 and 10 need to hold for the new store $S' = S \cup \{ r \mapsto (i \mapsto K) \}$. To this end, in S', the newly written

tag at address i cannot overlap with the cells occupied by any of the constructor arguments. Therefore, the desired end witnesses exist in S', thereby discharging this case.

* Case $l'^{r'} \neq l$:
  This case requires we establish that, for such a given location $l'^{r'}$, its corresponding end witness in the original store S also exists in the new store, S', that is, supposing $(l'^{r'} \mapsto \langle r', i_1 \rangle) \in M$, then $\tau; \langle r', i_1 \rangle; S \vdash_{ew} \langle r', i_2 \rangle$ implies $\tau; \langle r', i_1 \rangle; S' \vdash_{ew} \langle r', i_2 \rangle$. But the only way that any such end witness can be invalidated is if the write of the constructor tag at index i in $S' = S \cup \{ r \mapsto (i \mapsto K) \}$ affects any address in the end witness corresponding to location $l'^{r'}$, that is, any address in the right-open range $[i_1, i_2)$. The proof obligation therefore amounts to ruling out aliasing, that is, i falling in the range $[i_1, i_2)$. To this end, we start by working backwards from the typing of the location l^r, which corresponds to address i, the (only) address written by the constructor application. By inversion on T-DataConstructor, we establish that $l^r \in N$. As such, given the well formedness of the store S in the premise of this lemma, we obtain from WF D.5.3;3 that $(r \mapsto (i \mapsto K)) \notin S$. However, by the end-witness rule, for each $j \in [i_1, i_2)$, there exists a mapping from the address in the original store to its constructor tag K_j, which is $(r \mapsto (j \mapsto K_j)) \in S$. Therefore, the end witness judgement remains valid in store S', thus discharging this case.

– Case (WF D.5;2):

$$C \vdash_{wf_{cfc}} M; S'$$

The first two proof obligations of this judgement, namely WF D.5.2;1 and WF D.5.2;2, discharge immediately, because the environments used by these rules are unaffected in a data-constructor application. The only remaining obligation is WF D.5.2;3, because that requirement is affected by the write of the constructor tag, which is reflected in the new store S'. The obligation is to establish the preservation of the end witnesses of the locations in the domain of C. A similar proof obligation was already addressed by the proof of Property 4, in particular the subcase for $l'^{r'} \neq l^r$. The only difference in that case is the locations range over the domain of the store-typing environment Σ, whereas in this case the obligation concerns locations in the domain of the constraint environment C. However, the same proof steps apply in both cases, thus discharging this case.

– Case (WF D.5;3):

$$A'; N' \vdash_{wf_{ca}} M; S'$$

Obligations WF D.5.3;1 and WF D.5.3;3 discharge immediately because $l^r \notin N'$. It remains to discharge the obligation corresponding to WF D.5.3;2. Because it is the case that

$$(r \mapsto l^r) \in A' \wedge (l^r \mapsto \langle r, i_1 \rangle) \in M \wedge l^r \notin N' \wedge \tau; \langle r, i_1 \rangle; S' \vdash_{ew} \langle r, i_2 \rangle,$$

the obligation amounts to showing that the end witness of the constructor application is the new highest address in the store S', i.e., $i_2 > \mathit{MaxIdx}(r, S')$. There are two cases, based on the number of constructor arguments n:

* Case $n = 0$:
  We need to appeal to the well formedness of the store, as given by the premise of this lemma, and in particular rule WF D.5.3;1. To use this rule, we need to first establish $(r \mapsto l^r) \in A$ and $l^r \in N$, which follows immediately by inversion on T-DataConstructor. It therefore follows that

$$i_1 > \mathit{MaxIdx}(r, S).$$

  From this property, and by inspection on S', we discharge this case by establishing that the end witness of the constructor application is the highest address allocated in the new store S', i.e.,

$$i_1 + 1 = i_2 > \mathit{MaxIdx}(r, S').$$

* Case $n \geq 1$:
  To discharge this case, we need to show that the end witness of the last constructor argument, i.e., the one at position n, is the highest address in the new store S'. This obligation follows from the well formedness of the store S given by the premise of this lemma, and in particular the application of rule

WF D.5.3;2 to the end witness of the last constructor argument, i.e.,

$$(r \mapsto \overrightarrow{l_n^r}) \in A \wedge (\overrightarrow{l_n^r} \mapsto \langle r, \overrightarrow{w_n}\rangle) \in M \wedge \tau; \langle r, \overrightarrow{w_n}\rangle; S \vdash_{ew} \langle r, \overrightarrow{w_{n+1}}\rangle$$

The first two conjuncts follow from inversion on T-DataConstructor and T-Concrete-Loc, respectively, and the final one from Property 10. Thus, we have that $\overrightarrow{w_{n+1}} > \mathit{MaxIdx}(r, S)$. It follows that $\overrightarrow{w_{n+1}} > \mathit{MaxIdx}(r, S')$, because the newly written address in S', namely i_1, is such that $i_1 < \overrightarrow{w_{n+1}}$. By defintion of the end witness, we discharge this case by establishing that $\overrightarrow{w_{n+1}} = i_2 > \mathit{MaxIdx}(r, S')$.

The final obligation of this case concerns the requirement WF D.5.3;4. Part of this obligation is given by the premise of this lemma, for the original store S, and yields in particular that, for each $(r' \mapsto \varnothing) \in A$, it is the case that $r' \notin \mathit{dom}(S)$. The remaining obligation is to show the property holds for the new store S', which discharges immediately because, although $r \in S'$, by inversion on T-DataConstructor, it must be that $(r \mapsto \varnothing) \notin A$.

– Case (WF D.5;4):

$$\mathit{dom}(\Sigma') \cap N' = \varnothing$$

From the premise of the lemma, we have that the store is well formed with respect to typing environments Σ and N, and as such, we have that $\mathit{dom}(\Sigma) \cap N = \varnothing$. Therefore, we discharge this case by inspection of typing rule T-DataConstructor, which shows that $N' = N - \{ l \}$.

Case

$$\begin{array}{l}
\text{[D-Case]} \\
S; M; \texttt{case}\ \langle r, i\rangle^{l^r}\ \texttt{of}\ [\ldots, K\ (\overrightarrow{x : \tau@l^r}) \rightarrow e, \ldots] \Rightarrow \\
\qquad S; M'; e[\langle r, \overrightarrow{w}\rangle^{\overrightarrow{l^r}} / \overrightarrow{x}] \\
\text{where}\ \ M' = M \cup \{ \overrightarrow{l_1^r} \mapsto \langle r, i+1\rangle, \ldots, \overrightarrow{l_{j+1}^r} \mapsto \langle r, \overrightarrow{w_{j+1}}\rangle \} \\
\qquad \overrightarrow{\tau_1}; \langle r, i+1\rangle; S \vdash_{ew} \langle r, \overrightarrow{w_1}\rangle \\
\qquad \overrightarrow{\tau_{j+1}}; \langle r, \overrightarrow{w_j}\rangle; S \vdash_{ew} \langle r, \overrightarrow{w_{j+1}}\rangle \\
\qquad K = S(r)(i);\ j \in \{1, \ldots, n-1\};\ n = |\overrightarrow{x : \hat{\tau}}|
\end{array}$$

- The first of two proof obligations is to show that the result $e' = e[\langle r, \overrightarrow{w}\rangle^{\overrightarrow{l^r}} / \overrightarrow{x}]$ of the given step of evaluation is well typed, that is,

$$\varnothing; \Sigma'; C; A; N \vdash A; N; e' : \hat{\tau},$$

where $\hat{\tau} = \tau@l^r$. To establish the above, we start by obtaining the type for the body of the pattern, then the types of the concrete locations being substituted into the body, and finally use these two results with the substitution lemma to discharge the case. First, by inversion on the typing rules T-Case and T-Pattern, we establish that the body of the pattern, namely e, is well typed, i.e.,

$$\Gamma'; \Sigma'; C; A; N \vdash A; N; e : \tau@l^r,$$

where

$$\Gamma' = \{ \overrightarrow{x_1} \mapsto \overrightarrow{\tau_1}@\overrightarrow{l_1^r}, \ldots, \overrightarrow{x_1} \mapsto \overrightarrow{\tau_n}@\overrightarrow{l_n^r} \}$$

$$\Sigma' = \Sigma \cup \{ \overrightarrow{l_1^r} \mapsto \overrightarrow{\tau}_1, \ldots, \overrightarrow{l_n^r} \mapsto \overrightarrow{\tau}_n \}.$$

Second, we establish that the concrete locations being substituted for the pattern variables $\overrightarrow{x}$ are well typed. The specific obligation is, for each $i \in \{1, \ldots, n\}$, to establish that

$$\varnothing; \Sigma'; C; A; N \vdash A; N; \langle r, \overrightarrow{w_i}\rangle^{\overrightarrow{l_i^r}} : \overrightarrow{\tau}_i@\overrightarrow{l_i^r}.$$

The above holds because, by inversion on T-Concrete-Loc, the obligation is to show that, for each such i, $(\overrightarrow{l_i^r} \mapsto \overrightarrow{\tau_i}) \in \Sigma'$, which is immediate by inspection on Σ' above. Third, and finally, to establish the typing judgement for e', we use the Substitution Lemma D.1, which yields

$$\varnothing; \Sigma'; C; A; N \vdash A; N; e[\langle r, \overrightarrow{w_1}\rangle^{\overrightarrow{l_1^r}} / \overrightarrow{x_1}] \ldots [\langle r, \overrightarrow{w_1}\rangle^{\overrightarrow{l_n^r}} / \overrightarrow{x_n}] : \hat{\tau},$$

as needed, thereby discharging this obligation.

- The second obligation for this proof case is, given the affected environments, namely Σ' and M', to establish the well formedness of the resulting store, i.e.,

$$\Sigma'; C; A; N \vdash_{wf} M'; S.$$

We omit most of the details of this proof obligation because they discharge straightforwardly. The only part that requires attention is rule WF D.5;1, which is affected by the fresh locations in the location environment M'. This requirement discharges by inspection of D-Case, thereby discharging this obligation.

Case

[D-LetLoc-Tag]

$$S; M; \texttt{letloc}\ l^r = {l'}^{r'} + 1\ \texttt{in}\ e \Rightarrow S; M'; e$$

$$\text{where}\ \ M' = M \cup \{\, l^r \mapsto \langle r, i+1\rangle \,\};\ \langle r, i\rangle = M({l'}^{r'})$$

- The first of two proof obligations is to show that the result e of the given step of evaluation is well typed, that is,

$$\varnothing; \Sigma; C'; A'; N' \vdash A''; N''; e : \hat{\tau},$$

where $\hat{\tau} = \tau @ l^r$, $A' = A \cup \{\, r \mapsto l^r \,\}$, and $N' = N \cup \{\, l^r \,\}$. This proof obligation follows straightforwardly by inversion on T-LetLoc-Tag.
- The second obligation for this proof case is to show that

$$\Sigma; C'; A'; N' \vdash_{wf} M'; S.$$

The individual requirements, labeled WF D.5;1 - WF D.5;3, are handled by the following case analysis.

– Case (WF D.5;1): for each $({l'}^{r'} \mapsto \tau) \in \Sigma$, there exists some i_1, i_2 such that

$$({l'}^{r'} \mapsto \langle r, i_1\rangle) \in M' \wedge$$
$$\tau; \langle r, i_1\rangle; S \vdash_{ew} \langle r, i_2\rangle$$

By the well formedness of the store given in the premise of this lemma, the above already holds for the location environment M. The obligation discharges by inspecting the only new location in M', namely l^r, which is fresh and therefore cannot be in the domain of Σ.

– Case (WF D.5;2):

$$C' \vdash_{wf_{cfc}} M'; S$$

Of the requirements for this judgement, the only one that is not satisfied immediately by the well formedness of the store given in the premise of the lemma is requirement WF D.5.2;2 The specific requirement is to establish that

$$({l'}^{r'} \mapsto \langle r, i\rangle) \in M' \wedge$$
$$(l^r \mapsto \langle r, i+1\rangle) \in M',$$

which follows immediately by inversion on D-LetLoc-Tag.

– Case (WF D.5;3):

$$A'; N' \vdash_{wf_{ca}} M'; S$$

∗ Case (WF D.5.3;1):

$$(l^r \mapsto \langle r, i+1\rangle) \in M' \wedge i+1 > \mathit{MaxIdx}(r, S)$$

The first conjunct follows immediately from inversion on D-LetLoc-Tag. To establish the second, however, we first need to establish that the address corresponding to location ${l'}^{r'}$ is the highest index in the store S. To do so, we need to appeal to the well formedness of the store given by the premise of this lemma. In particular, we need to use the same requirement we are trying to prove, namely WF D.5.3;1, but in this case, instantiating for ${l'}^{r'}$ in the original location environment M. By inversion on T-LetLoc-Tag, we have that $A(r) = {l'}^{r'}$ and ${l'}^{r'} \in N$, and as a consequence of WF D.5.3;1,

$$({l'}^{r'} \mapsto \langle r, i\rangle) \in M \wedge i > \mathit{MaxIdx}(r, S).$$

Using the second conjunct above, this case discharges immediately.

* Case (WF D.5.3;2): This obligation discharges immediately because, by inversion on T-LetLoc-Tag, $l^r \in N'$.
* Case (WF D.5.3;3): The proof obligation is to establish that, for any constructor tag K,

$$((l^r \mapsto \langle r, i+1 \rangle) \in M' \wedge$$
$$(r \mapsto (i+1 \mapsto K)) \notin S)$$

The first conjunct discharges by inversion on D-LetLoc-Tag, and the second as a consequence of having already established just above that $i+1 > \mathit{MaxIdx}(r, S)$.
* Case (WF D.5.3;4): The proof obligation is to establish that, for each $(r \mapsto \varnothing) \in A'$, it is the case that $r \notin \mathit{dom}(S)$. This case discharges because, from the premise of the lemma, this property holds for the original environment A and store S, and, by inversion on T-LetLoc-Tag, continues to hold for A' and S'.

– Case (WF D.5;4):

$$\mathit{dom}(\Sigma) \cap N' = \varnothing$$

Because it is a bound location, $l \notin \mathit{dom}(\Sigma)$, and by inversion on T-LetLoc-Tag, $l \in N'$, which discharges the obligation.

Case

[D-LetLoc-After]
$$S; M; \texttt{letloc}\ l^r = (\texttt{after}\ \tau@{l_1}^r)\ \texttt{in}\ e \Rightarrow S; M'; e$$
where $M' = M \cup \{\, l^r \mapsto \langle r, j \rangle \,\};\ \langle r, i \rangle = M({l_1}^r)$
$\tau; \langle r, i \rangle; S \vdash_{ew} \langle r, j \rangle$

- The first of two proof obligations is to show that the result e' of the given step of evaluation is well typed, that is,

$$\varnothing; \Sigma; C'; A'; N' \vdash A''; N''; e' : \hat{\tau},$$

where $\hat{\tau} = \tau@{l'}^{r'}$. This proof obligation follows straightforwardly by inversion on T-LetLoc-After.
- The second obligation for this proof case is to show that

$$\Sigma; C'; A'; N' \vdash_{wf} M'; S.$$

The individual requirements, labeled WF D.5;1 - WF D.5;3, are handled by the following case analysis.

– Case (WF D.5;1): for each $({l'}^r \mapsto \tau) \in \Sigma$, there exists some i_1, i_2 such that

$$({l'}^r \mapsto \langle r, i_1 \rangle) \in M' \wedge$$
$$\tau; \langle r, i_1 \rangle; S \vdash_{ew} \langle r, i_2 \rangle$$

By the well formedness of the store given in the premise of this lemma, the above already holds for the location environment M. The obligation discharges by inspecting the only new location in M', namely l^r, which is fresh and therefore cannot be in the domain of Σ.

– Case (WF D.5;2):

$$C' \vdash_{wf_{cfc}} M'; S$$

Of the requirements for this judgement, the only one that is not satisfied immediately by the well formedness of the store given in the premise of the lemma is requirement WF D.5.2;3 The specific requirement is to establish that

$$({l_1}^r \mapsto \langle r, i \rangle) \in M' \wedge$$
$$\tau; \langle r, i \rangle; S \vdash_{ew} \langle r, j \rangle \wedge$$
$$(l \mapsto \langle r, j \rangle) \in M'$$

which follows immediately by inversion on D-LetLoc-After.

– Case (WF D.5;3):

$$A'; N' \vdash_{wf_{ca}} M'; S$$

* Case (WF D.5.3;1):

$$(l \mapsto \langle r, j \rangle) \in M' \wedge j > \mathit{MaxIdx}(r, S)$$

The first conjunct follows immediately from inversion on D-LetLoc-After. To establish the second, however, we first need to establish that the end witness j of location ${l_1}^r$ is the maximum index in the store S. To do so, we need to appeal to the well formedness of the store given by the premise of this lemma. In particular, we need to use the requirement WF D.5.3;2, instantiating for ${l_1}^r$ in the original location environment M. By inversion on T-LetLoc-After, we have that $A(r) = {l_1}^r$, ${l_1}^r \notin N$, and $\tau; \langle r, i \rangle; S \vdash_{ew} \langle r, j \rangle$. Thus, as a consequence of WF D.5.3;2,

$$j > \mathit{MaxIdx}(r, S).$$

Using the second and third conjuncts above, this case discharges immediately.

* Case (WF D.5.3;2): This obligation discharges immediately because, by inversion on T-LetLoc-After, $l \in N'$.
* Case (WF D.5.3;3): The proof obligation is to establish that, for any constructor tag K,

$$((l \mapsto \langle r, j \rangle) \in M' \wedge$$
$$(r \mapsto (j \mapsto K)) \notin S)$$

The first conjunct discharges by inversion on D-LetLoc-After, and the second as a consequence of having already established just above that $j > \mathit{MaxIdx}(r, S)$.

* Case (WF D.5.3;4): This case discharges straightforwardly, in a similar fashion to the previous case, for D-LetLoc-Tag.

– Case (WF D.5;4):

$$\mathit{dom}(\Sigma) \cap N' = \varnothing$$

Because it is a bound location, $l \notin \mathit{dom}(\Sigma)$, and by inversion on T-LetLoc-After $l \in N'$, which discharges this obligation.

Case

[D-LetLoc-Start]
$$S; M; \texttt{letloc}\ l^r = (\texttt{start}\ r)\ \texttt{in}\ e \Rightarrow S; M'; e$$
$$\text{where}\ \ M' = M \cup \{\, l^r \mapsto \langle r, 0 \rangle \,\}$$

- The first of two proof obligations is to show that the result e' of the given step of evaluation is well typed, that is,

$$\varnothing; \Sigma; C'; A'; N' \vdash A''; N''; e' : \hat{\tau},$$

where $\hat{\tau} = \tau @ {l'}^{r'}$. This obligation follows straightforwardly by inversion on T-LetLoc-Start.

- The second obligation for this proof case is to show that

$$\Sigma; C'; A'; N' \vdash_{wf} M'; S.$$

The individual requirements, labeled WF D.5;1 - WF D.5;3, are handled by the following case analysis.

– Case (WF D.5;1): for each $(l' \mapsto \tau) \in \Sigma$, there exists some i_1, i_2 such that

$$(l' \mapsto \langle r, i_1 \rangle) \in M' \wedge$$
$$\tau; \langle r, i_1 \rangle; S \vdash_{ew} \langle r, i_2 \rangle$$

By the well formedness of the store given in the premise of this lemma, the above already holds for the location environment M. The obligation discharges by inspecting the only new location in M', namely l^r, which is fresh and therefore cannot be in the domain of Σ.

– Case (WF D.5;2):

$$C' \vdash_{wf_{cfc}} M'; S$$

Of the requirements for this judgement, the only one that is not satisfied immediately by the well formedness of the store given in the premise of the lemma is requirement WF D.5.2;1. The specific requirement is to establish that

$$(l^r \mapsto \langle r, 0 \rangle) \in M',$$

which follows immediately by inversion on D-LetLoc-Start.

– Case (WF D.5;3):

$$A'; N' \vdash_{wf_{ca}} M'; S$$

  * Case (WF D.5.3;1):

$$(l \mapsto \langle r, 0 \rangle) \in M' \wedge 0 > MaxIdx(r, S)$$

    The first conjunct follows immediately from inversion on D-LetLoc-Start. To establish the second conjunct above, it suffices establish that $r \notin dom(S)$, because, as such, $MaxIdx(r, S) = -1$, by the definition of $MaxIdx$. This property follows from the well formedness of the store, in particular, from rule WF D.5.3;4. The rule guarantees that, if $(r \mapsto \varnothing) \in A$, then $r \notin dom(S)$, as needed. By inversion on T-LetLoc-Start, we establish this precondition, thereby discharging the case.

  * Case (WF D.5.3;2): This obligation discharges immediately because, by inversion on T-LetLoc-Start, $l \in N'$.

  * Case (WF D.5.3;3): The proof obligation is to establish that, for any constructor tag K,

$$\begin{array}{l}((l^r \mapsto \langle r, 0 \rangle) \in M' \wedge \\ (r \mapsto (0 \mapsto K)) \notin S)\end{array}$$

    The first conjunct discharges by inversion on D-LetLoc-Start, and the second as a consequence of having already established just above that $0 > MaxIdx(r, S)$.

  * Case (WF D.5.3;4): The obligation for this case is to establish that for each $(r \mapsto \varnothing) \in A' = A \cup \{ r \mapsto l^r \}$, it is the case that $r \notin dom(S)$. The part of this obligation pertaining to environment A is given by the premise of this lemma, and thus it only remains to establish that the property holds for the rest, namely $\{ r \mapsto l^r \}$. This part discharges trivially, because $(r \mapsto \varnothing) \notin A'$, thereby discharging this case.

– Case (WF D.5;4):

$$dom(\Sigma) \cap N' = \varnothing$$

  This case discharges straightforwardly.

Case

[D-LetRegion]
$$S; M; \texttt{letregion}\ r\ \texttt{in}\ e \Rightarrow S; M; e$$

- The first of two proof obligations is to show that the result e' of the given step of evaluation is well typed, that is,

$$\varnothing; \Sigma; C'; A'; N' \vdash A''; N''; e' : \hat{\tau},$$

  where $\hat{\tau} = \tau @ l'^{r'}$. This proof obligation follows straightforwardly by inversion on T-LetRegion.
- The second obligation for this proof case is to show that

$$\Sigma; C; A'; N \vdash_{wf} M; S.$$

  The individual requirements, labeled WF D.5;1 - WF D.5;3, are handled by the following case analysis.

– Case (WF D.5;1): for each $(l'^r \mapsto \tau) \in \Sigma$, there exists some i_1, i_2 such that

$$\begin{array}{l}(l'^r \mapsto \langle r, i_1 \rangle) \in M \wedge \\ \tau; \langle r, i_1 \rangle; S \vdash_{ew} \langle r, i_2 \rangle\end{array}$$

  This case discharges immediately by inversion of T-LetRegion and D-LetRegion, because none of the relevant environments are affected by the transition.

– Case (WF D.5;2):

$$C \vdash_{wf_{cfc}} M; S$$

  The case discharges in a fashion similar to the previous one.

– Case (WF D.5;3):

$$A'; N \vdash_{wf_{ca}} M; S$$

Of the requirements in this judgement, the only one that is affected by the new environment A' is requirement WF D.5.3;4. The specific obligation is to establish that, for each $(r \mapsto \varnothing) \in A'$, it is the case that $r \notin dom(S)$. By inversion on T-LetRegion, $A' = A \cup \{ r \mapsto \varnothing \}$, and therefore, the first part of the obligation, that is, for A, is already given by the premise of this lemma. As such, it only remains to establish that $r \notin dom(S)$, which follows from r being a fresh region, thereby ruling out it being in the store, and thus discharging this case.

– Case (WF D.5;4):

$$dom(\Sigma) \cap N' = \varnothing$$

This case discharges straightforwardly.

Case

$$\frac{}{S; M; \texttt{let}\ x : \hat{\tau} = v_1\ \texttt{in}\ e_2 \Rightarrow S; M; e_2[v_1/x]}\ [\text{D-Let-Val}]$$

- The first of two proof obligations is to show that the result $e_2[v_1/x]$ of the given step of evaluation is well typed, that is,

$$\varnothing; \Sigma'; C; A; N \vdash A; N; e_2[v_1/x] : \tau_2 @ {l_2}^{r_2}.$$

By inversion on T-Let, we obtain the type of the value being bound

$$\varnothing; \Sigma; C; A; N \vdash A; N; v_1 : \tau_1 @ {l_1}^{r_1},$$

and we obtain the type of the body

$$\Gamma'; \Sigma'; C; A; N \vdash A; N; e_2 : \tau_2 @ {l_2}^{r_2}$$

where

$$\Gamma' = \{ x \mapsto \tau_1 @ {l_1}^{r_1} \}$$
$$\Sigma' = \Sigma \cup \{ {l_1}^{r_1} \mapsto \tau_1 \}.$$

As such we can apply the Substitution Lemma D.1, as follows

$$\varnothing; \Sigma'; C; A; N \vdash A; N; e_2[v_1/x][{l_1}^{r_1}/{l_1}^{r_1}] : \tau_2 @ {l_2}^{r_2},$$

which discharges our obligation, given that the substitution of the bound location ${l_1}^{r_1}$ is the identity substitution.

- Given the instantiations of Σ' and M' used by the previous step, the second obligation for this proof case is to show that

$$\Sigma'; C; A; N \vdash_{wf} M'; S.$$

The individual requirements, labeled WF D.5;1 - WF D.5;3, are handled by the following case analysis.

– Case (WF D.5;1): for each $({l'}^{r} \mapsto \tau) \in \Sigma' = \Sigma \cup \{ {l_1}^{r_1} \mapsto \tau_1 \}$, there exists some i_1, i_2 such that

$$({l'}^{r} \mapsto \langle r, i_1 \rangle) \in M \wedge$$
$$\tau; \langle r, i_1 \rangle; S \vdash_{ew} \langle r, i_2 \rangle$$

This obligation amounts to showing the above holds for the bound location ${l_1}^{r_1}$, because the well formedness of the store given by the premise of this lemma guarantees the property holds for locations bound in Σ. The value v_1 bound at location ${l_1}^{r_1}$ is a value and is well typed, and as such, there are only two typing rules that could apply, namely T-Var and T-Concrete-Loc. By inversion on these rules, we establish that

$$({l_1}^{r_1} \mapsto \tau_1) \in \Sigma.$$

Therefore, we can discharge this obligation by application of well formedness of the store, in particular, the rule WF D.5;1 we are currently considering. Concretely, we discharge this obligation by instantiating that rule to

$$(l_1{}^{r_1} \mapsto \langle r_1, i_1 \rangle) \in M \wedge$$
$$\tau_1; \langle r_1, i_1 \rangle; S \vdash_{ew} \langle r_1, i_2 \rangle.$$

– Case (WF D.5;2):

$$C \vdash_{wf_{cfc}} M; S$$

This case discharges immediately because the relevant environments are affected by neither the of the relevant typing nor the dynamic-semantic judgement.

– Case (WF D.5;3):

$$A; N \vdash_{wf_{ca}} M; S$$

This case discharges immediately because the relevant environments are affected by neither the of the relevant typing nor the dynamic-semantic judgement.

– Case (WF D.5;4):

$$dom(\Sigma') \cap N = \varnothing$$

This case discharges straightforwardly.

Case

$$\frac{S; M; e_1 \Rightarrow S'; M'; e_1' \qquad e_1 \neq v}{S; M; \texttt{let}\ x : \hat{\tau} = e_1\ \texttt{in}\ e_2 \Rightarrow S'; M'; \texttt{let}\ x : \hat{\tau} = e_1'\ \texttt{in}\ e_2}\ [\text{D-Let-Expr}]$$

- The first of two proof obligations is to show that the result $\texttt{let}\ x : \hat{\tau} = e_1'\ \texttt{in}\ e_2$ of the given step of evaluation is well typed, that is,

$$\varnothing; \Sigma; C; A'; N' \vdash A''; N''; \texttt{let}\ x : \hat{\tau} = e_1'\ \texttt{in}\ e_2 : \tau_2 @ l_2{}^{r_2},$$

The induction hypothesis is

$$\begin{aligned} \text{If}\ & \varnothing; \Sigma; C; A; N \vdash A'; N'; e_1 : \tau_1 @ l_1{}^{r_1} \\ \text{and}\ & \Sigma; C; A; N \vdash_{wf} M; S \\ \text{and}\ & S; M; e_1 \Rightarrow S'; M'; e_1' \\ \text{then for some}\ & \Sigma' \supseteq \Sigma, C' \supseteq C, \\ & \varnothing; \Sigma'; C'; A'; N' \vdash A''; N''; e_1' : \tau_1 @ l_1{}^{r_1} \\ \text{and}\ & \Sigma'; C'; A'; N' \vdash_{wf} M'; S'. \end{aligned}$$

By inversion on T-Let, we establish that

$$\varnothing; \Sigma; C; A; N \vdash A'; N'; e_1 : \tau_1 @ l_1{}^{r_1},$$

and, by the premise of this lemma, we establish that

$$\Sigma; C; A; N \vdash_{wf} M; S$$

and by inversion on D-Let-Expr we establish that

$$S; M; e_1 \Rightarrow S'; M'; e_1'.$$

Now, we can apply the above to the induction hypothesis to establish

$$\begin{aligned} \text{For some}\ & \Sigma' \supseteq \Sigma, C' \supseteq C, \\ & \varnothing; \Sigma'; C'; A'; N' \vdash A''; N''; e_1' : \tau_1 @ l_1{}^{r_1} \\ \text{and}\ & \Sigma'; C'; A'; N' \vdash_{wf} M'; S'. \end{aligned}$$

By inversion on T-Let, we also have that

$$\Gamma'; \Sigma'; C; A'; N' \vdash A''; N''; e_2 : \tau_2 @ l_2{}^{r_2},$$

where

$$\Gamma' = \{\, x \mapsto \tau_1 @ {l_1}^{r_1} \,\}$$
$$\Sigma' = \{\, {l_1}^{r_1} \mapsto \tau_1 \,\}.$$

By inspection on T-Let and the previous two typing judgements, that is, for e_1' and e_2, we discharge this case.

- The second obligation

$$\Sigma'; C'; A'; N' \vdash_{wf} M'; S'$$

discharges immediately from the result of the induction hypothesis, which is established by the above.

Case

[D-App]

$$S; M; f\ [\overrightarrow{l^r}]\ \overrightarrow{v} \Rightarrow S; M; e[\overrightarrow{v}/\overrightarrow{x}][\overrightarrow{l^r}/\overrightarrow{{l'}^{r'}}]$$
$$\text{where } fd = Function(f)$$
$$f : \forall_{\overrightarrow{l'^{r'}}}.\overrightarrow{\hat{\tau}_f} \to \hat{\tau}_f; (f\,\overrightarrow{x} = e) = Freshen(fd)$$

- The first of two proof obligations is to show that the result $e' = e[\overrightarrow{v}/\overrightarrow{x}][\overrightarrow{l^r}/\overrightarrow{{l'}^{r'}}]$ of the given step of evaluation is well typed, that is,

$$\varnothing; \Sigma'; C; A; N' \vdash A'; N''; e[\overrightarrow{v}/\overrightarrow{x}][\overrightarrow{l^r}/\overrightarrow{{l'}^{r'}}] : \hat{\tau},$$

where $\hat{\tau} = \tau @ l^r$. To this end, we first establish typing judgements for the body of the callee and then the arguments of the function, and finally discharge the first obligation by combining the two results using the substitution lemma. By inversion on T-Function-Definition, the type judgement

$$\Gamma; \Sigma''; C; A; N \vdash A; N'; e : \tau @ l^r,$$

holds for body of the callee e, with constrants for any caller, such that $l^r \in N$, $l^r \notin N'$ and $A(r) = l^r$, where

$$\Gamma = \{\, \overrightarrow{x_1} \mapsto \overrightarrow{\tau_1 @ {l_1'}^{r'}},\ \ldots\ , \overrightarrow{x_n} \mapsto \overrightarrow{\tau_n @ {l_n'}^{r'}} \,\}$$
$$\Sigma'' = \{\, \overrightarrow{{l_1'}^{r'}} \mapsto \overrightarrow{\tau_1},\ \ldots\ , \overrightarrow{{l_n'}^{r'}} \mapsto \overrightarrow{\tau_n} \,\}.$$

Regarding the arguments to the call, we obtain by inversion on T-App that

$$\varnothing; \Sigma; C; A; N \vdash A; N; \overrightarrow{v_i} : \overrightarrow{\tau_i @ {l_i}^r}$$

for $i \in \{1 \ldots n\}$. Furthermore, by inversion on T-App, we obtain that $l^r \in N$, $l^r \notin N'$, and $A(r) = l^r$, which altogether satisfy the requirements of T-Function-Definition. Now, by application of the Substitution Lemma, we have that

$$\varnothing; \Sigma; C; A; N' \vdash A; N'; e[\overrightarrow{v_1}/\overrightarrow{x_1}][\overrightarrow{{l_1}^r}/\overrightarrow{{l_1'}^{r'}}] \ldots [\overrightarrow{v_n}/\overrightarrow{x_n}][\overrightarrow{{l_n}^r}/\overrightarrow{{l_n'}^{r'}}] : \tau @ l^r.$$

- Given the new environment N' used by the previous step, the second obligation for this proof case is to show that

$$\Sigma; C; A; N' \vdash_{wf} M; S.$$

The individual requirements, labeled WF D.5;1 - WF D.5;3, are handled by the following case analysis.

– Case (WF D.5;1): for each $({l'}^r \mapsto \tau) \in \Sigma$, there exists some i_1, i_2 such that

$$({l'}^r \mapsto \langle r, i_1 \rangle) \in M \wedge \tag{11}$$
$$\tau; \langle r, i_1 \rangle; S' \vdash_{ew} \langle r, i_2 \rangle \tag{12}$$

This case discharges immediately from the well formedness of the store given by the premise of this lemma.

– Case (WF D.5;2):

$$C \vdash_{wf_{cfc}} M; S$$

This case discharges immediately from the well formedness of the store given by the premise of this lemma.

– Case (WF D.5;3):

$$A; N' \vdash_{wf_{ca}} M; S$$

Of the requirements pertaining to this judgement, the only one potentially affected by the new environment N' is requirement WF D.5.3;2. The specific obligation therein is to establish that

$$((r \mapsto l^r) \in A \wedge (l^r \mapsto \langle r, i_s \rangle) \in M \wedge l^r \notin N' \wedge \tau; \langle r, i_s \rangle; S \vdash_{ew} \langle r, i_e \rangle) \Rightarrow i_e > MaxIdx(r, S).$$

The reason the change to environment N' might affect the above is because, if all the conjuncts above hold, then it remains to establish that $i_e > MaxIdx(r, S)$ holds. However, it turns out that the fourth conjunct above does not hold, i.e., there is no such end witness in the store S, thus relieving the obligation to establish $i_e > MaxIdx(r, S)$. The reason the end witness does not exist is yielded by the well formedness of the store given by the premise of this lemma, in particular requirement WF D.5.3;1. That is, by inversion on T-App, it is the case that

$$(r \mapsto l^r) \in A \wedge l^r \in N.$$

Therefore, requirement WF D.5.3;1 implies that

$$i_s > MaxIdx(r, S).$$

As such, given that the store S remains unchanged and the above, it is straightforward to show that the end witness starting at i_s cannot exist, thereby discharging this case.

- Case (WF D.5;4):

$$dom(\Sigma) \cap N' = \varnothing$$

This case discharges because, from the well formedness of the store given by the premise of this lemma, $dom(\Sigma) \cap N = \varnothing$, and because $N' = N - \{ l^r \}$.

■