LoCal: A Language for Programs Operating on Serialized Data

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.

CCS Concepts • **Software and its engineering → Formal language definitions**; **Compilers**.

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314631

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

ACM Reference Format:
Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. 2019. LoCal: A Language for Programs Operating on Serialized Data. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314631

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, 13, 15].

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, 12] 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 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* [32], Cap'N Proto, and prior Gibbon [29] (§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.

```
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.

Storing the pointers that maintain the internal structure of the tree represents a significant storage overhead.

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, 26] and such layouts are common in high-performance computing settings [8, 13, 15, 20, 21]. 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 [17], region calculi [24], 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 [22]. 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* [29], 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).

$K \in$ 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*	$::= x \mid \langle r, i \rangle^l$
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

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.

$$
\begin{array}{rrcl}
\text{Typing Env.} & \Gamma & ::= & \{\, x_1 \mapsto \hat{\tau}_1,\ \ldots\ , x_n \mapsto \hat{\tau}_n \,\} \\
\text{Store Typing} & \Sigma & ::= & \{\, l_1^{r_1} \mapsto \tau_1,\ \ldots\ , l_n^{r_n} \mapsto \tau_n \,\} \\
\text{Constraint Env.} & C & ::= & \{\, l_1^{r_1} \mapsto le_1,\ \ldots\ , l_n^{r_n} \mapsto le_n \,\} \\
\text{Allocation Pointers} & A & ::= & \{\, r_1 \mapsto ap_1,\ \ldots\ , r_n \mapsto ap_n \,\} \\
 & & & \text{where } ap = l^r \mid \emptyset \\
\text{Nursery} & N & ::= & \{\, l_1^{r_1},\ \ldots\ , l_n^{r_n} \,\}
\end{array}
$$

Figure 3. Extended grammar of LoCal for static semantics

Like previous work on region-based memory [25], 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.

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 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

[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]
$$\frac{\Gamma; \Sigma; C; A; N \vdash A'; N'; e_1 : \tau_1@{l_1}^r \qquad \Gamma'; \Sigma'; C; A'; N' \vdash A''; N''; e_2 : \hat{\tau}_2}{\Gamma; \Sigma; C; A; N \vdash A''; N''; \texttt{let}\ x : \tau_1@{l_1}^r = e_1\ \texttt{in}\ e_2 : \hat{\tau}_2}$$
where $\Gamma' = \Gamma \cup \{\, x \mapsto \tau_1@{l_1}^r \,\};\ \Sigma' = \Sigma \cup \{\, {l_1}^r \mapsto \tau_1 \,\}$

[T-LetRegion]
$$\frac{\Gamma; \Sigma; C; A'; N \vdash A''; N'; e : \hat{\tau}}{\Gamma; \Sigma; C; A; N \vdash A''; N'; \texttt{letregion}\ r\ \texttt{in}\ e : \hat{\tau}}$$
where $A' = A \cup \{\, r \mapsto \emptyset \,\}$

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

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

Figure 4. Selected type-system rules for LoCal.

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, 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.

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.

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

Figure 5. Extended grammar of LoCal for dynamic semantics

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` 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 available in the Appendix of the extended version [28]

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

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

Code	A	C	N
`letloc` l^r = `start(r)`	$\{r \mapsto l^r\}$	$\emptyset$	$\{l^r\}$
`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	$\{r \mapsto l^r\}$	$\{{l_a}^r \mapsto l^r + 1, {l_b}^r \mapsto after(T@{l_a}^r)\}$	$\emptyset$

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

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

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

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

[D-Case]
$S; M;$ `case` $\langle r, i\rangle^l$ `of` $[\ldots, K\ \overrightarrow{(x : \tau@l^r)}\ \rightarrow\ e, \ldots] \Rightarrow$
$S; M'; e''$
where $\overrightarrow{{l_f}^r}$ *fresh*; $e'' = e'[\overrightarrow{{l_f}^r}/\overrightarrow{l^r}]$
$M' = M \cup \{ \overrightarrow{l^r_{f_1}} \mapsto \langle r, i + 1\rangle, \ldots, \overrightarrow{l^r_{f_{j+1}}} \mapsto \langle r, \overrightarrow{w_{j+1}}\rangle \}$
$e' = e[\langle r, \overrightarrow{w}\rangle^{\overrightarrow{l^r}}/\overrightarrow{x}]$
$\overrightarrow{\tau_1}; \langle r, i + 1\rangle; S \vdash_{ew} \langle r, \overrightarrow{w_1}\rangle$
$\overrightarrow{\tau_{j+1}}; \langle r, \overrightarrow{w_j}\rangle; S \vdash_{ew} \langle r, \overrightarrow{w_{j+1}}\rangle$
$j \in \{ 1, \ldots, n - 1 \};\ n = |\overrightarrow{x : \hat{\tau}}|$
$K = S(r)(i)$

Figure 6. Selected dynamic-semantics rules for LoCal.

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.

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 extended version [28]. The extended version also provides a detailed explanation of the evaluation of a sample program.

3.3.1 Type Safety

The key to proving type safety is our store-typing rule, given in full in the extended version [28].

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

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)

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

Proof The type safety follows from an induction with the progress and preservation lemmas, the details of which are given in the extended version [28].

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 [29]—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

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

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

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 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 [29]. 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.

$$
\texttt{maplike} : \forall\ {l_1}^{r_1}\ {l_2}^{r_2}.\ \texttt{Tree @}\ {l_1}^{r_1} \xrightarrow{\{l_1, l_2\}} \texttt{Tree @}\ {l_2}^{r_2}
$$

$$
\texttt{rightmost} : \forall\ l^{r}\ .\ \texttt{Tree @}\ l^{r} \xrightarrow{\{\}} \texttt{Int}
$$

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 [29]: 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. [29]). 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 in the extended version [28]. 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* [24, 25], 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) [32] 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, 12], 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 [19, 23]—or restricted languages such as XPath [18]—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 [29], 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 [30], 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 [14]. Indeed, LoCal's addition of offset fields to data *is* ornamentation. Practical implementations of ornaments [31] 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 extended version [28]. 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) [29], 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

[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 [28].
[4]An "insanely fast data interchange format," https://capnproto.org/, [27]

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 the extended version [28]. 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 the extended version [28], except for one that is not self-explanatory:

repMax: a variant of the "repmin" [4, 16] 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. [29]. 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.

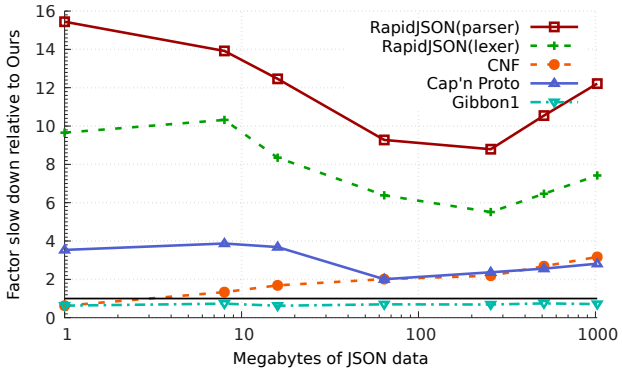
Versus pointer-based representations The "NonPacked" approach is LoCal configured to always insert indirections and thus emulate a traditional object representation. In this case, we are being overly friendly to this pointer-based representation by allowing it to read its input (for example, the input 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**.

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!

[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

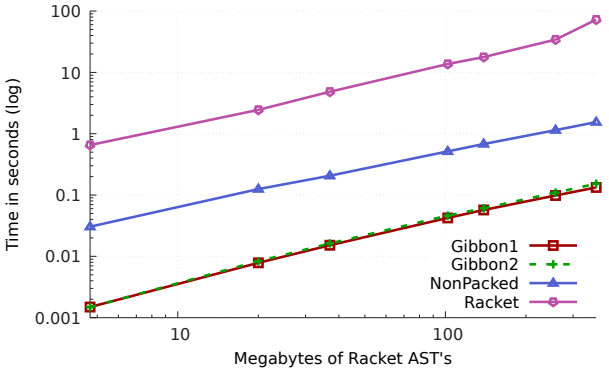


Figure 9. Count the nodes in a Racket AST

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. [32].

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).

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 ("Non-Packed") 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

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)

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* → *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] 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

[13] 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

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

[15] 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/

[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] 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

[18] 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

[19] 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.

[20] 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

[21] 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

[22] 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

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

[24] 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

[25] 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

[26] 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

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

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

[29] 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

[30] 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

[31] 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

[32] 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