A LANGUAGE-BASED APPROACH TO PROGRAMMING WITH SERIALIZED DATA

Michael Vollmer

Submitted to the faculty of the University Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the School of Informatics, Computing, and Engineering,

Indiana University

February 2021

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Ryan Newton, PhD

Jeremy Siek, PhD

Sam Tobin-Hochstadt, PhD

Larry Moss, PhD

November 13, 2020

Acknowledgments

Just days after arriving in Bloomington, my wife and I found ourselves at The Tap with a few members of the Indiana University Programming Languages group. Here we met Cameron Swords, Mike Vitousek, Jaime Guerrero, Matteo Cimini, and Jason Hemann, for what would be the first of many long nights spent discussing programming languages, science fiction, bad movies, philosophy, and beer. Without these nights, I would likely never have survived graduate school.

My advisor, Ryan Newton, not only taught me how to do research, he gave me a framework for how to think about software. He turned me from a Schemer into a hardened Haskeller and staunch advocate for static types. He taught me how to build a compiler, and building compilers is now one of my favorite things in the world.

I want to thank the other members of my committee, Jeremy Siek, Sam Tobin-Hochstadt, and Larry Moss, for everything they have done to help me over the years. Similarly, I want to thank Milind Kulkarni for his mentorship throughout the last few years. I also could not have gotten here without Chaitanya Koparkar and Laith Sakka, for their help and collaboration on the Gibbon project.

For encouraging me to go to graduate school, and for sparking my interest in computer science research, I want to thank my instructors at California State University, Sacramento: V. Scott Gordon, for showing me Scheme for the first time, and for taking the time to talk to me about my personal projects and guiding me through the grad school application process; Ted Krovetz, for giving me the opportunity to give lectures in his compilers course while I was still an undergraduate; and John Clevenger, for giving me the eternally useful advice, that "you don't need to debug your code if you just write it correctly the first time."

Early in my Ph.D., I was lucky enough to get to work with Bo Joel Svensson and Trevor L. McDonell while they were postdocs at IU, and I want to thank them for the mentorship they both

provided.

I am thankful to all the attendees of PL Wonks in Bloomington, for presenting their work and putting up with mine. Thanks especially to Wonks regulars Ryan Scott, Vikraman Choudhury, Matthew Heimerdinger, Sarah Spall, Laurel Carter, Praveen Narayanan, Andre Kuhlenschmidt, Rajan Walia, Buddhika Chamith, Andrew Kent, and Paulette Koronkevich. And thanks to Ambrose Bonnaire-Sergeant, for the fun conversations and for getting lost with me in Mumbai.

Thanks to my parents, Jane and Joe, and my brother Matt. They always supported me, they encouraged me when I was down, they told me they're proud of me, and they politely listened when I tried to explain my research. I also apologize for accidentally erasing the hard drive of your work computer when I was a child because I wanted to see what `DELTREE` did in MS-DOS.

And most of all, I want to thank my wife, Victoria Vollmer. She was the person who first encouraged me to go to graduate school. At the start, the plan was that I would study Computer Science and she would study Mathematics. We applied to schools around the country, and eventually, both chose Indiana University. Our adventure started with a cross-country drive from California to Indiana, and eventually, years later, led to us getting married. Along the way, Victoria had become a formidable computer scientist and logician. Thank you, Tori, for being my biggest advocate and my best friend.

Michael Vollmer

A LANGUAGE-BASED APPROACH TO PROGRAMMING WITH SERIALIZED DATA

In a typical data-processing application, the representation of data in memory is distinct from its representation in a serialized form on disk. The former has pointers and an arbitrary, sparse layout, facilitating easier manipulation by a program, while the latter is packed contiguously, facilitating easier I/O. I propose a programming language, LoCal, that unifies the in-memory and on-disk representations of data. LoCal extends prior work on region calculi into a location calculus, employing a type system that tracks the byte-addressed layout of all heap values. I present the formal semantics of LoCal and prove type safety, and show how to infer LoCal programs from unannotated source terms. Then, I demonstrate how to efficiently implement LoCal in a practical compiler that produces code competitive with hand-written C.

Ryan Newton, PhD

Jeremy Siek, PhD

Sam Tobin-Hochstadt, PhD

Larry Moss, PhD

Contents

Chapter 1

Introduction

Many programs running today use heap object representations that are *fixed* by the language runtime system. For example, the Java and Haskell runtimes dictate an object layout, and the compiler must stick to it for all programs. In contrast, when humans optimize a program, one of their primary *levers on performance* is changing data representation. An HPC programmer knows how to pack a regular tree into a byte array for more efficient access [22, 11, 25].

Furthermore, 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.* At first glance, the only alternative would seem to be writing low-level code to deal directly with specialized or serialized data layouts, an error-prone way to achieve performance optimization at the expense of safety and readability.

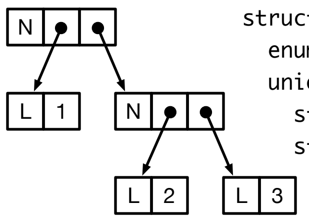
In my thesis I propose a way to ameliorate both of these concerns: reify *data layout* as an explicit part of the program. In short, I am proposing a *language-based* approach to solving the problem of how to safely program with serialized data. To that end, I present a language, LoCal (which stands 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 [3, 20] and object databases [15], 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). This allows programmers to interpolate between fully serialized and fully pointer-based representations. LoCal can thus serve as a flexible intermediate representation for compilers or synthesis tools.

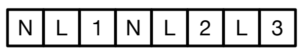
Gibbon is one such compiler. Gibbon is an experimental compiler that automatically transforms high-level functional programs that operate on tree-like data into low-level C code that operates directly on serialized data. Internally, Gibbon represents programs in LoCal, so it is able to tune the memory layout of particular data structures to control precisely how and how much the data will be serialized. In general, Gibbon produces code that is often significantly faster than equivalent pointer-based code, with speedups of over 2$\times$ compared to hand-optimized, pointer-based C code, and often much more than that compared to existing compilers for functional languages like Haskell and OCaml. In addition to presenting LoCal, in this thesis I will cover the design and implementation of the Gibbon compiler, as well as an evaluation of the compiler on various benchmarks and applications.

This thesis will be broken down into three parts. Chapter 1 (this chapter) gives an overview of LoCal, with background, motivation, and related work. Chapter 2 describes the language and presents a formal semantics for LoCal, as well as some extensions. Finally, Chapter 3 presents the Gibbon compiler, which uses LoCal internally, and presents an evaluation of Gibbon's performance on various tasks. A full proof of type safety for the LoCal language is included in Appendix A.



```
struct Tree {
  enum {Leaf, Node} tag;
  union {
    struct {long long elem}
    struct {struct Tree* l;
            struct Tree* r;}}}
```

(a) Standard representation of a tree structure in C: by default, word-sized tags *and* pointers.



(b) Serialized version of the same tree. Not to scale: tags take one byte and integers eight.

Figure 1.1: Standard and serialized representations of trees

1.1 Motivation

Consider the simple tree data structure in Fig. 1.2a, written in a language that supports algebraic datatypes, where a tree is either a leaf with an integer or a node with two trees.

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. Depending on the language and runtime system, there is likely additional space for storing other information about the object, but this is the basic layout. For example, a C compiler uses 96 bytes of memory to represent the tree shown in Figure 1.1a.

On the other hand, if we are sending the tree over the network, we would naturally use a more compact form in serializing it, as shown in Figure 1.1b. In the latter version, we use the same 24 bytes for the data in the leaves, but only 5 bytes for the spine (capturing the "tags" of the 5 nodes in the tree), rather than 72. Storing the pointers that maintain the internal structure of the tree represents a significant storage overhead.

Further, a tree traversal processing this memory representation follows a precisely linear memory access pattern, because the data is already laid out in a preorder traversal. On architectures with

```
-- A Tree is either:
--   - a Leaf with an Int, or
--   - a Node with a Tree and a Tree
data Tree = Leaf Int | Node Tree Tree
```

(a) Haskell data type representing a binary tree

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

(b) Function that sums leaf values in a binary tree

```
int sumPacked (byte * &ptr) {
  int ret = 0;
  if (* ptr == LEAF) {
    ptr++; // skip past leaf tag
    ret = * (int*)ptr; // retrieve integer from leaf
    ptr += sizeof(int); //skip past integer
  } else { // tag must be node
    ptr++; // skip past node tag
    ret += sumPacked(ptr); // sum left sub-tree
    ret += sumPacked(ptr); // sum right sub-tree
  }
  return ret;
}
```

(c) A low-level traversal of serialized tree data written in C++

inexpensive unaligned access, such as modern x86, this is a desirable in-memory representation as well as a serialization format.[1] But without this structural information, in most settings the preorder 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 given in Fig. 1.2c.

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.

As another example, consider a function that traverses a binary tree as defined previously, returns a new binary tree such that each integer has been incremented by one. This is again straightforward to describe recursively in a language like Haskell, as shown in Fig. 1.3a.

Like before, we can write a C++ function to do this same computation, only operating on and returning serialized binary trees. This new function will have to accept two arguments (pointers to the input and output byte arrays, respectively), and will additionally return a pointer. The code is shown in Fig. 1.3b. This time around, the procedure is even more complicated. Even a single mistake in the pointer arithmetic, such as forgetting to increment one of the pointers, would lead

[1]Even restricted to aligned access, we would still shrink from 72 bytes to 20 by switching to a packed format.

```haskell
add1 :: Tree → Tree
add1 tr =
  case tr of
    Leaf n → Leaf (n + 1)
    Node a b → Node (add1 a) (add1 b)
```

(a) add1 in haskell

```cpp
char* add1(char* tin, char* tout) {
  if (*tin == Leaf) {
    *tout = Leaf;
    tin++; tout++;
    *(int*)tout = *(int*)tin + 1;
    return (tin + sizeof(int));
  } else {
    *tout = Node;
    tin++; tout++;
    char* t2 = add1(tin,tout);
    tout += (t2 - t);
    return add1(t2,tout);
  }
}
```

(b) add1 in c++

Figure 1.3: add1 example

```
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)
```

Figure 1.4: Function that sums the leaf values of a serialized binary tree

to a severe error that would be difficult to debug.

Comparing the Haskell procedures operating on traditional trees with the C++ procedures operating on serialized trees, it is clear that 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. 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.

So, is it possible for programmers to enjoy the benefits of satefy and convenience that programmers get when using a high-level language, while also programming directly on serialized data? My proposal is to write the above examples in a language, *LoCal*, that is *expressly designed to use dense serializations* for its values. The LoCal `sum` function in Fig. 1.4 extends the simple functional one above with region and location annotations.

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 I will show later, to *construct* data the type system must enforce that adjacent fields be serialized consecutively.

I set out to answer the question of whether it is possible to gain the benefits of programming with serialization without sacrificing safety, and my assertion is that it is, when using a *language-based approach.* LoCal is that language, and the design and implementation of LoCal is the subject of this thesis.

1.2 Background and Related Work

My work on LoCal builds on a lot of existing research on programming languages, compilers, and systems. In this section I will outline some of this background, and relate this existing research to the research I am presenting in this thesis.

Serialized data 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[2], 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) [44] is a feature provided by the Glasgow Haskell Compiler since release 8.2. This is a feature reminiscent of Lisp and Self systems which save heap snapshots[3], except more targeted.

[2]An "insanely fast data interchange format," `https://capnproto.org/`, [38]

[3]For instance, you can save the Self "world" to a `.snap` file (`selflanguage.org`). In the Lisp lineage, Chez Scheme, before version 9, is an example of a system that could save/restore heap files (`www.scheme.com/csug8`).

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 [3, 2, 20], 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.

Tracking resource usage in types For decades there has been significant research into using types to track and constrain resource use in programming languages. In the 1980s, researchers like Lucassen and Gifford developed effect type systems [21] which use types to perform side-effect analysis. Their approach involves a type system with three *kinds*: types, which describe the value that an expression may evaluate to; effects, which describe the side-effects that an expression may perform; and regions, which describe the area of the store in which side-effects may occur. This research continued in many forms, such as region calculi [35, 36, 13], where regions in types were used to manage memory allocation and deallocation. LoCal builds on these systems, and borrows the notion of regions, while adding an additional notion of *symbolic locations* on top of regions to track the relative location of values inside regions.

Another commonly discussed way to manage resources, and particularly memory, using types in functional programs is the use of *linear types* [41]. Values that have a linear type must be used exactly once, and (as Wadler writes), just like the world, they can not be duplicated or destroyed. Because of this, they do not require garbage collection, so a functional programmer may use linear types to construct programs that need no automatic memory management. Additionally, linear types can be used to do in-place updates to mutable data structures in a functional way. This second property is important to this thesis, as later, in §3.2.2, I show how this property of linear types can be used to safely write and read serialized data in Linear Haskell (an extension so the

Haskell programming language that adds support for linear types).

Compiler support for external dense data If we look instead at compiler support for computing with data in dense or external forms, there are many compilers for stream processing languages [34, 29]—or restricted languages such as XPath [28]—that generate efficient computations over data streams. StreamIt [34], for example, is a programming language designed for high-performance streaming applications. It has an optimizing compiler that is able to perform stream-specific optimizations and generate code that in some cases rivals the performance of hand-optimized assembly code.

These systems are somewhat related, but Gibbon and LoCal differ in targeting general purpose recursive functions over algebraic datatypes.

Fusion and deforestation *Deforestation* is the technique of removing intermediary list/tree-like data in functional computations [40]. For example, take the functional program:

```
sum(map square (upto 1 n))
```

A compiler or interpreter would generally have `upto 1 n` return a list of numbers from 1 to n, then have the `map` procedure return a new list with the numbers squared, and finally have `sum` consume this list and return an integer. This style of programming is convenient, because it allows programmers to build their programs out of general, re-usable procedures like `map` and `upto`, but it results in unnecessary memory allocation in the form of intermediary structures. Deforestation transforms this computation and removes the need to allocate intermediary lists, instead *fusing* the computations together, computing the sum directly via a single recursive procedure. This can safely be achieved if certain conditions are met, and it has the potential both to greatly improve the performance of functional programs, as well as allow functional programmers the freedom to freely compose programs with re-usable combinators without worrying about incurring a significant cost from intermediary data structures.

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 like Wadler's [40] don't rise to being able to handle a full deserializer, and popular approaches based on more restrictive *combinator libraries* [9] are less expressive than HiCal and LoCal.

The Gibbon compiler has the ability to perform a kind fusion/deforestation on HiCal programs as a compiler optimization, though the details of how it accomplishes this are outside the scope of this thesis.

Ornaments *Ornaments* are a body of theory regarding connections between related data structure that differ based on additions or reorganization [24]. Indeed, LoCal's addition of offset fields to data *is* ornamentation. Practical implementations of ornaments [43] 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.

Compiler optimizations for tree traversals LoCal relates to a broader literature on optimizing tree-traversing programs and heap representations. 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* [6], 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 [37, 17, 8] has similar effects and limitations to cache-conscious structure layout.

One exception is Chilimbi et al.'s work on *automatic structure splitting* [7], where objects are transformed into split representations, allowing hot fields from multiple objects to be co-located on a single cache line while those objects' cold fields are placed elsewhere. Because this layout optimization changes the internal representation of the object, Chilimbi et al. develop a compiler pass that automatically transforms code to work with the split representation. The transformations for structure splitting concern how to access object fields, and hence, unlike our work, do not require deeper transformations to remove the pointer dereferences inherent in traversing linked data structures. Indeed, neither this work nor cache-conscious structure placement affect the behavior of pointers in data structures.

Automatic pool allocation [17] proposes allocating disjoint data structures into disjoint partitions of memory, and this approach can be extended with compression optimization [18]. 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).

Array-based approaches to traversing tree-like data Hsu looks at a representation of abstract syntax trees that uses a matrix layout, allowing operations to be specified in a data-parallel manner without traversing pointers [16]. While this representation shares a goal with ours of avoiding pointers, it is not "packed"—the representation requires a dense representation of a sparse matrix—and hence does not yield the type of space savings we target.

In the high-performance computing community, linearizing trees and tree traversals for improved performance has been a common technique [22, 11]. These linearizations tend to be *ad hoc*, written specifically for a given application, and each application must be re-written by hand to benefit.

This contrasts with our compiler-based approach which allows programmers to write using idiomatic traversal algorithms, relying on the compiler to synthesize the packed representation as well as the algorithm to traverse that representation.

Similar *ad hoc* layout transformations have recently been pursued in the context of vectorization [25, 30, 31]. Meyerovich et al. discuss different linearization schemes that can promote packed SIMD loads and stores, improving vectorization efficiency [25]. These layouts have the implicit effect of eliminating pointer dereferences, as in our packed representations, but rely on index arithmetic to traverse formerly-linked nodes, rather than encoding particular traversal orders. Ren et al. look at a wide range of tree layouts for vectorization, each targeted at different traversal patterns [30, 31]. These layouts are chosen to match the traversal patterns of an application, enabling the removal of pointers, as in our layouts. Ren et al. use a library-based approach: applications are written using high-level tree interfaces, with specific layouts chosen based on hardware and application considerations. In contrast, this work focuses on compiler-driven transformations of both the tree layout and the code that traverses the tree.

Chapter 2

The Location Calculus

2.1 Overview

This section describes LoCal, which is a programming language for programming with serialized data. A primary use case of LoCal is as an *intermediate language* for a compiler. Traditional compilers are built on a number of well-defined intermediate abstractions and translations that close the semantic gap between source and target. If we are building a compiler to generate code that operates on serialized data, what we need are analogous way-points to structure compilers that target serialized-data traversals (stream-processors, essentially). Indeed, there is quite a semantic gap between the low-level, buffer-mutating, pointer-bumping programs, and a source language of high-level, pure, recursive functions on algebraic datatypes. LoCal is designed to structure the space between, where types are augmented to track *locations* within regions (e.g., byte offsets).

LoCal follows in the tradition of typed assembly language [27], region calculi [35], and Cyclone [13] 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 §2.3.1 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).

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 [32]. 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 §3.2 I 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 first used in *Gibbon* [39] prior to the design of LoCal, which previously compiled functions on fully serialized data, while not preserving asymptotic complexity.) Next, I will present a formalized core subset of LoCal, its type system (§2.2.1), and its operational semantics (§2.2.2).

2.2 Formal Language and Grammar

Fig. 2.1 gives the grammar for a formalized core of LoCal. I 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 [36], 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. It is required 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 ($\texttt{after}\ \tau @ l^r$), 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, I refer to the latter as *symbolic locations*. A concrete location is a tuple

$K \in$ 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	$::= \forall_{\overrightarrow{l^r}}. \overrightarrow{\hat{\tau}} \rightarrow \hat{\tau}$
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	$::= K\ (\overrightarrow{x : \hat{\tau}}) \rightarrow e$
Location Expressions	le	$::= (\texttt{start}\ r)$
		$\mid (l^r + 1)$
		$\mid (\texttt{after}\ \hat{\tau})$

Figure 2.1: Grammar of LoCal

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

Figure 2.2: Extended grammar of LoCal for static semantics

$\langle r, i \rangle^l$ 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.

2.2.1 Static Semantics

In Fig. 2.2, I extend the grammar with some extra details necessary for describing the type system. The typing rules for expressions in LoCal are given in Fig. 2.3 and Fig. 2.4, 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 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

[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]

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

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

[T-LetRegion]

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

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

[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-Start]

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

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

[T-LetLoc-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]

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

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

Figure 2.3: Typing judgments for LoCal (1)

[T-App]

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

where $f : \forall_{\overrightarrow{l''^{r''}}} . \overrightarrow{\tau_i @ l_i'''^{r_i'''}} \rightarrow \tau @ l'''^{r'''} ; (f\,\overrightarrow{x} = e) = Function(f)$

$N' = N - \{\, l^r \,\};\ n = |\overrightarrow{v}|;\ i \in \{\, 1, \ \dots \ , n \,\}$

[T-Function-Definition]

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

where $\Gamma = \{\, \overrightarrow{x_1} \mapsto \overrightarrow{\tau_1 @ l_1{}^{r_1}}, \ \dots \ , \overrightarrow{x_n} \mapsto \overrightarrow{\tau_n @ l_n{}^{r_n}} \,\}$

$\Sigma = \{\, \overrightarrow{l_1{}^{r_1}} \mapsto \overrightarrow{\tau_1}, \ \dots \ , \overrightarrow{l_n{}^{r_n}} \mapsto \overrightarrow{\tau_n} \,\}$

$C = \varnothing;\ A = \{\, r \mapsto l^r \,\};\ N = \{\, l^r \,\}$

$n = |\overrightarrow{x}| = |\overrightarrow{\tau @ l^r}|$

[T-Pattern]

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

where $\Gamma' = \Gamma \cup \{\, \overrightarrow{x_1} \mapsto \overrightarrow{\tau_1'} @ \overrightarrow{l_1'^{r'}}, \ \dots \ , \overrightarrow{x_n} \mapsto \overrightarrow{\tau_n'} @ \overrightarrow{l_n'^{r'}} \,\}$

$\Sigma' = \Sigma \cup \{\, \overrightarrow{l_1'^{r'}} \mapsto \overrightarrow{\tau_1'}, \ \dots \ , \overrightarrow{l_n'^{r'}} \mapsto \overrightarrow{\tau_n'} \,\}$

$i \in \{\, 1, \dots, n \,\};\ n = |\overrightarrow{\tau'}| = |\overrightarrow{x : \tau' @ l'^{r}}|$

[T-Case]

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

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

[T-Program]

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

where $\Gamma = \varnothing;\ \Sigma = \varnothing$

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

Figure 2.4: Typing judgments for LoCal (2)

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, all shown in Fig. 2.3), 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 §2.2.2). 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 I will address shortly in §2.3.1, 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 2.1, 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.

To finish out the typing rules, Fig. 2.4 contains rules for function application and definition, as well as pattern matching. 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. The final rule type checks a whole program, consisting of datatype and function definitions.

To simplify the formalism and proofs, I 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.

$\textbf{let}\ \ \mathrm{x}\ :\ \mathrm{T}\ @\ l^r\ =\ \mathrm{y}\ \ \textbf{in}\ \ \cdots$

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 lemmas or the use of 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.

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 @ ${l_a}^r$)	$\{r \mapsto {l_b}^r\}$	$\{{l_a}^r \mapsto l^r + 1,$ ${l_b}^r \mapsto \mathit{after}(T@{l_a}^r)\}$	$\{l^r, {l_b}^r\}$
let y : T @ ${l_b}^r$ = Leaf ${l_b}^r$ 2	$\{r \mapsto {l_b}^r\}$	$\{{l_a}^r \mapsto l + 1,$ ${l_b}^r \mapsto \mathit{after}(T@{l_a}^r)\}$	$\{l^r\}$
Node l^r x y	$\{r \mapsto l^r\}$	$\{{l_a}^r \mapsto l^r + 1,$ ${l_b}^r \mapsto \mathit{after}(T@{l_a}^r)\}$	$\varnothing$

Table 2.1: Step-by-step example of type checking a simple expression.

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

Figure 2.5: Extended grammar of LoCal for dynamic semantics

[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-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-LetRegion]

$$S; M; \texttt{letregion}\ r\ \texttt{in}\ e \Rightarrow S; M; e$$

[D-App]

$$S; M; f\ [\overrightarrow{l^r}]\ \overrightarrow{v} \Rightarrow S; M; e[\overrightarrow{v}/\overrightarrow{x}][\overrightarrow{l^r}/\overrightarrow{l'^{r'}}]$$

$$\text{where } fd = \mathit{Function}(f)$$

$$f : \forall_{\overrightarrow{l'^{r'}}}.\overrightarrow{\hat{\tau}_f} \rightarrow \hat{\tau}_f; (f\ \overrightarrow{x} = e) = \mathit{Freshen}(fd)$$

[D-Case]

$$S; M; \texttt{case}\ \langle r, i \rangle^{l^r}\ \texttt{of}\ [\ldots, K\ \ (\overrightarrow{x : \tau @ l^r})\ \rightarrow\ e, \ldots] \Rightarrow S; M'; e[\langle r, \overrightarrow{w} \rangle^{\overrightarrow{l^r}} / \overrightarrow{x}]$$

$$\text{where } M' = M \cup \{\, \overrightarrow{l_1^r} \mapsto \langle r, i+1 \rangle, \ldots, \overrightarrow{l_{j+1}^r} \mapsto \langle r, \overrightarrow{w_{j+1}} \rangle \,\}$$

$$\overrightarrow{\tau_1}; \langle r, i+1 \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_1} \rangle$$

$$\overrightarrow{\tau_{j+1}}; \langle r, \overrightarrow{w_j} \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_{j+1}} \rangle$$

$$K = S(r)(i);\ j \in \{\, 1, \ldots, n-1 \,\};\ n = |\overrightarrow{x : \hat{\tau}}|$$

Figure 2.6: Dynamic semantics rules for LoCal

2.2.2 Dynamic Semantics

The dynamic semantics for expressions in LoCal are given in Fig. 2.6, 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, the semantics make use of a *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, I use the *location map*, M, to map symbolic locations 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 (for a more thorough explanation, see §2.2.3). 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 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 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 §2.2.3.1. The program can start taking evaluation steps from this configuration.

Example Consider this code snippet of LoCal.

```
letloc $l_1{}^r$ = $l_0{}^r$ + 1 in
let a : Tree @ $l_1{}^r$ = (Leaf $l_1{}^r$) in
letloc $l_2{}^r$ = (after (Tree @ $l_1{}^r$)) in
let b : Tree @ $l_2{}^r$ = (Leaf $l_2{}^r$) in
Node $l_0{}^r$ 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: $S = \{\, r \mapsto \{\, 0 \mapsto \texttt{Node}, 1 \mapsto \texttt{Leaf}, 2 \mapsto \texttt{Leaf} \,\}\,\}$.

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.

2.2.3 Type Safety

LoCal is a *type safe* language, and I will demonstrate that property in this section. Some details of the proof are listed in §A, such as an overview of notation and the complete cases for the lemmas and theorems.

2.2.3.1 Store typing

A key part of the safety of LoCal programs is the following property: if a term e is type $\tau @ l^r$, then if we look in the store under region r at the location represented by l, we will find the start of a serialized data structure which is a valid serialization of a value of type τ. In other words, the types tell us the truth about the values in the store. To achieve this, we rely on the store being *well-formed*, whose definition itself uses three other judgements (shown in Table 2.2).

The definition of store well formedness follows.

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

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 (§2.2.3.2) and correct allocation in regions (§2.2.3.4), 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.

	Judgement form	Summary
Store well formedness	$\Sigma; C; A; N \vdash_{wf} M; S$	The store S along with location map M are well formed with respect to typing environments Σ, C, and A.
End witness	$\tau; \langle r, i_s \rangle; S \vdash_{ew} \langle r, i_e \rangle$	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	$C \vdash_{wf_{cfc}} M; S$	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	$A; N \vdash_{wf_{ca}} M; S$	Allocation in store S along with location map M is well formed with respect to allocation-typing environments A and N.

Table 2.2: Summary of judgements used to establish well formedness of the store.

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

2.2.3.2 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

   $\texttt{data}\, \tau = \overrightarrow{K_1\ \vec{\tau}_1} \mid \ldots \mid K'\ \vec{\tau}' \mid \ldots \mid \overrightarrow{K_m\ \vec{\tau}_m}$

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 $j \in \{1, \ldots, n-1\}; n = |\vec{\tau}'|$

4. $i_e = \overrightarrow{w_n}$

2.2.3.3 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. $(l^r \mapsto (\texttt{after}\ \tau @ l'^{r^r})) \in C \Rightarrow$

   $((l'^r \mapsto \langle r, i_1 \rangle) \in M \wedge$

   $\tau; \langle r, i_1 \rangle; S \vdash_{ew} \langle r, i_2 \rangle \wedge$

   $(l^r \mapsto \langle r, i_2 \rangle) \in M)$

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

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

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

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

3. $l^r \in N \Rightarrow$

   $((l^r \mapsto \langle r, i \rangle) \in M \wedge$

   $(r \mapsto (i \mapsto K)) \notin S)$

4. $(r \mapsto \varnothing) \in A \Rightarrow$

   $r \notin \mathit{dom}(S)$

2.2.3.5 Type safety theorem

The key to proving type safety is the handling of the store above. After that has been established, type safety for LoCal can be proven in the standard way using progress and preservation. The full details of the proof are shown in A, but the statements of the main theorems are written here, as well as a handful of representative cases.

Lemma 2.2.1 (Progress)

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

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

A representative case to look at in more detail is the T-DataConstructor case.

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 2.2.3.4;3. But a precondition for using WF 2.2.3.4;3 that the location is unwritten, i.e., $l^r \in N$. This precondition is satisfied by inversion on T-DataConstructor. The application of rule WF 2.2.3.4;3 therefore yields the desired result, thereby discharging this case.

Lemma 2.2.2 (Preservation)

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

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

A representative case to look at in more detail is the D-Case case.

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 $\Gamma' = \{ \overrightarrow{x_1} \mapsto \overrightarrow{\tau_1} @ \overrightarrow{l_1{}^r}, \ldots, \overrightarrow{x_1} \mapsto \overrightarrow{\tau_n} @ \overrightarrow{l_n{}^r} \}$ and $\Sigma' = \Sigma \cup \{ \overrightarrow{l_1{}^r} \mapsto \overrightarrow{\tau}_1, \ldots, \overrightarrow{l_n{}^r} \mapsto \overrightarrow{\tau}_n \}$. 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 $\varnothing; \Sigma'; C; A; N \vdash A; N; \langle r, \overrightarrow{w_i}\rangle^{\overrightarrow{l_i{}^r}} : \overrightarrow{\tau}_i @ \overrightarrow{l_i{}^r}$. This 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 (given and proven in Appendix A.0.1), 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., $\Sigma'; C; A; N \vdash_{wf} M'; S$. I will omit most of the details of this proof obligation because they discharge straightforwardly. The only part that requires attention is rule WF 2.2.3.1;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.

Finally, the type safety theorem follows:

Theorem 2.2.3 (Type safety)

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

Proof The type safety follows from an induction with the progress and preservation lemmas.

2.3 Extensions

2.3.1 Offsets and Indirections

As motivated in §1.2, 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.

```
size : ∀ $l^r$ . Tree @ $l^r$ → Int
size [$l^r$] t = case t of
                  Leaf → 1
                  Node (a : Tree @ $l_a{}^r$) (b : Tree @ $l_b{}^r$)
                    → (size [$l_a{}^r$] a) + (size [$l_b{}^r$] b)
```

Figure 2.7: LoCal function computing the size of a binary tree

2.3.2 Parallelism

So far, this chapter has presented a language for performing tree traversals over serialized data (with some potential indirection). While this data representation strategy works well for sequential programs, there is an intrinsic tension if we want to parallelize these tree traversals. As the name implies, efficiently serialized data must often be read serially. To change that, first, enough *indexing* data must be left in the representation in order for parallel tasks to "skip ahead" and process multiple subtrees in parallel. Second, the allocation areas must be bifurcated to allow allocation of outputs in parallel. Thankfully, the extension discussed previously about adding *indirections* helps provide solutions to these problems, but a bit of care has to be taken when updating the language semantics to ensure that parallel execution is safe.

There are several opportunities for parallelism in LoCal programs. The first kind of parallelism is available when LoCal programs access the store in a read-only fashion, such as the program that calculates the size of a binary tree (as shown in Fig. 2.7). However, even though the recursive calls in the `Node` case can safely evaluate in parallel, there is a subtelty: parallel evaluation is efficient only if the `Node` constructor stores offset information for its child nodes. If it does, then the address of `b` can be calculated in constant time, thereby allowing the calls to proceed immediately in parallel. If there is no offset information, then the overall tree traversal is necessarily sequential, because the starting address of `b` can be obtained only after a full traversal of `a`. As such, there is a tradeoff between space and time, that is, the cost of the space to store the offset in the `Node` objects versus the time of the sequential traversal (e.g., of `a`) forced by the absence of offsets.

Programs that write to the store also provide opportunities for parallelism. The most immediate such opportunity exists when the program performs writes that affect different regions. For example, the writes to construct the leaf nodes for `a` and `b` can happen in parallel because different regions cannot overlap in memory.

```
letregion ra in
letregion rb in
letloc $la^{ra}$ = start ra in
letloc $lb^{rb}$ = start rb in
let a : Tree @ $la^{ra}$ = Leaf $la^{ra}$ in
let b : Tree @ $lb^{ra}$ = Leaf $lb^{rb}$ in
...
```

There is another kind of parallelism that is more challenging to exploit, but is at least as important as the others: the parallelism that can be realized by allowing different fields of the same constructor to be filled in parallel. This is crucial in LoCal programs, where large, serialized data frequently occupy only a small number of regions, and yet there are opportunities to exploit parallelism in their construction. Consider the `buildtree` function, mentioned earlier in this chapter, which creates a binary tree of a given size `n` in a given region `r`. If we want to access the parallelism between the recursive calls, we need to break the data dependency that the right branch has on the left. The starting address of the right branch, namely ${l_b}^r$, is assigned to be end witness of the left branch by the `letloc` instruction. But the end witness of the left branch is, in general, known only after the left branch is completely filled, which would effectively sequentialize the computation. One non-starter would be to ask the programmer to specify the size of the left branch up front, which would make it possible to calculate the starting address of the right branch. Unfortunately, this approach would introduce safety issues, such as incorrect size information, of exactly the kind that LoCal is designed to prevent. Instead, I will explain an approach that is safe-by-construction and efficient. A full formalization and proof of type safety for parallel LoCal is future work, so this section will outline a sketch of how the operational semantics of LoCal can be extended to safely allow parallel execution.

The expression syntax is the same, and no changes are necessary to the type system. The operational semantics do require some changes, most notably the addition of a richer form of indexing in regions.

Take, as an example, a LoCal computation involving a binary tree, where the left child `a` at location ${l_a}^r$ is to be computed in parallel with the right child `b`. Each task has its own private view of memory, which is realized by giving the child and parent task copies of the store S and location map M. These copies differ in one way, however: each sees a different mapping for the starting location of `a`. The child task sees the mapping ${l_a}^r \mapsto \langle l_a, 1 \rangle$, which is the ultimate starting address of `a` in the heap.

The parent task sees a different mapping for ${l_a}^r$, namely $\langle r, \texttt{i-var}\ 1 \rangle$. This location is an *ivar index*: it behaves exactly like an I-Var [1], and, in our example, stands in for the completion of the memory being filled for `a`, by the child task. Any expression in the body of the let expression that tries to read from this location blocks on the completion of the child task. When `a` is used, it will force the parent task to join with its child task. The ivar index $\langle r, \texttt{i-var}\ 1 \rangle$ will be substituted with $\langle l_a, 1 \rangle$, and all the new entries in the location map and store map of the child task are merged into the corresponding environments in the parent task. Finally, the results are wired together using indirections, as covered in §2.3.1.

A key point is that indirections are necessary here for parallelism to occur, but if parallelism is only desired in certain parts of a program rather than at all possible opportunities, then indirections are only necessary at those points in the program where actual parallelism occurs. In a recursive program where parallelism is desired only up to a certain depth, data representation is pointer-based only insofar as parallelism is needed, and both data representation and control flow "bottom out" to sequential at a certain point. That is, granularity control in the data mirrors traditional granularity control in parallel task scheduling.

To make parallel LoCal practical when adding it to Gibbon, it was necessary to add `spawn` and `sync` forms to the language, so that it is explicit where parallelism should be exploited, even though these annotations are not strictly necessary for correctness in the semantics of parallel LoCal.

A thorough evaluation of the performance of parallel LoCal is in §3.3.4.

Chapter 3

The Gibbon Compiler

3.1 Converting Functional Programs to the Location Calculus

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?

All of these choices can be represented directly in LoCal, making it an ideal intermediate language for a compiler. This chapter will describe Gibbon, an experimental compiler that transforms functional programs to work on (mostly) serialized data. Gibbon represents its programs in LoCal, performing various analyses and transformations, then generates low-level C code. 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.

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

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, the plan is 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.)

Implementing HiCal The compiler must perform a variant of *region inference* [36, 35], 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. 2.3 and Fig. 2.4, 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)`.

[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 §3.2.3.

3.2 Compiling the Location Calculus

In this section I present a compiler for the LoCal language, which consists of the formalized core from §2.2, 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 inserts enough indirection in datatypes to preserve the asymptotic complexity of the source functions (under the assumption of $O(1)$ field access), but it also provides 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.

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. It uses the same basic encoding as Haskell's `Data.Serialize` module derives by default, but we 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 the front-end of Gibbon, as described previously in §3.1.

3.2.1 Compiler Structure

LoCal was implemented with a micropass framework. It is a 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 $\rightarrow$LoCal passes, we lower to a second IR, *NoCal.* As shown in Fig. 3.1, 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 early in Chapter 1. Code

$$n \in \text{Integers}$$

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

Figure 3.1: Grammar of NoCal (an extension of LoCal)

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, I will 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.

The LoCal runtime system is responsible for region-based memory management. A detailed description of the memory management strategy is available in §3.2.3 In brief, the compiler uses

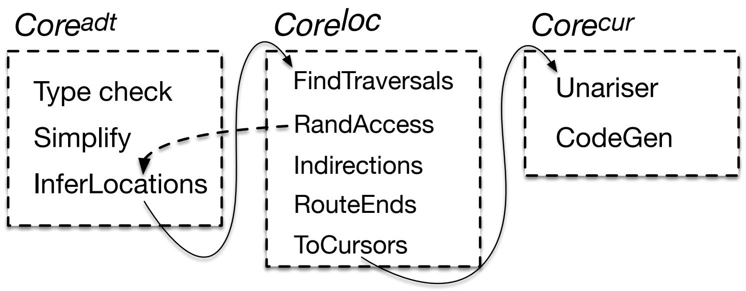


Figure 3.2: Compiler architecture. Here I show the most important passes within the three phases of the compiler delimited by the three core IRs (front-end, middle-end, back-end).

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.

3.2.1.1 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, I use a variation of a technique I previously proposed in [39]. Specifically, I 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.

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

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

3.2.1.2 Implementing Random Access

Once it is known what fields are traversed, it is possible to 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, all direct references to pattern-matched fields *after* the first variable-sized one are eliminated. 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, the compiler adds additional information following the tag. Specifically, for a constructor `K T1 T2`, if immediate access to `T2` is needed, the compiler includes a 4-byte relative-offset after the constructor.

Back-tracking Unfortunately, when datatypes are modified to add offsets, it invalidates 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 it will only backtrack at most once[4]. After this is complete, the `rightmost` example becomes:

$$\texttt{rightmost} \ : \ \text{Tree @ } {l_1}^{r_1} \ \rightarrow \ \texttt{Int}$$

$$\texttt{rightmost tr} \ =$$

[4]The marked set of constructors is a conservative over-approximation; it would be possible in principle to construct a program with types `A` and `B`, both of which are marked for random access, but where `A` becoming random-access would obviate the need for `B`. Further optimizations are possible.

```
case tr of
  Leaf (n : Int @ $l_n{}^{r_1}$ ) → n
  Node' (ran : (Ptr (Tree @ $l_b{}^{r_1}$)) @ $l_{ran}{}^{r_1}$)
        (a : Tree @ $l_a{}^{r_1}$) (b : Tree @ $l_b{}^{r_1}$)
     → rightmost [$l_b{}^{r_1}$] *ran
```

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 [39]: 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 _ → ···`".

3.2.1.3 Routing End Witnesses

Each of the traversal effects previously inferred proves the compiler logically reaches a point in memory, but to realize it in the program the compiler adds an additional return value to the function, witnessing the end-location for traversed values (as described in [39]). Here I extend the syntax to allow additional location-return values, equivalent to returning tuples. The `buildtree` example becomes:

```
buildtree : ∀ $l^r$. Int $\xrightarrow{\{l\}}$ [$after(Tree@l^r)$] 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.

3.2.1.4 Converting to NoCal

In this stage, the compiler converts programs 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. The compiler drops much of the type information at this phase (see §3.2.2 for how to preserve types), 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.

3.2.2 Linear Cursors

In the process described previously in §3.2.1.4, Gibbon transforms programs into NoCal, which uses a cursor-passing style. As Gibbon is now, these cursor-passing programs carry no type information on the cursors themselves—the compiler has erased information about packed types. This is convenient if the goal is to eventually generate C code, which is what Gibbon does.

However, there are other situations where you may want cursor types to ensure that serialized data is written and read safely, just like LoCal; for example, if a programmer wants to embed a safe interface for programming with serialized data into a mainstream functional language like Haskell. In this section I will briefly present a system for typing cursors, and demonstrate its relationship to NoCal and how it may be expressed in Linear Haskell [5]. This initially appeared in [39] as a typed intermediate language for an earlier version of the Gibbon compiler, and subsequently was adapted in [5] as an example use case for linear types in Haskell. In this section I borrow the presentation style and examples from the latter.

The basic idea is that cursors are *indexed* by a list of types. Write cursors are indexed by a list of types that corresponds to the values that must be written to that cursor, and read cursors are indexed by a list of types that corresponds to the values that can be read from the cursor. Cursors must be *linear*, and operations that consume cursors return new cursors with updated types. An example interface for the simple binary tree data type we have been using is given in Fig. 3.3, and an interface for manipulating general packed data is given in Fig. 3.4.

This interface is *type-safe* in the sense that it provides a layer of abstraction for consuming and producing serialized data such that a program only reads byte-ranges at the size and type they were originally written. For this to work, the `Packed` type must be abstract, so a client working with a `Packed Tree` is not privy to the memory layout of its serialization.

The code in this section uses the experimental linear types extension to Haskell. With this extension, function types with ⊸ are linear functions, while ordinary arrows remain ordinary arrows. A linear function is constrained to consume its argument exactly once. The `Ur` data type is short for *unrestricted*, and is similar to ! in linear logic.

Because a client does not have direct access to the serialized bytes, consuming a serialized binary tree is done with the help of a pattern matching combinator like `caseTree` in Fig. 3.3. This function takes the serialized tree and two continuations (one for if the tree is a leaf, and one for if it is a node), and the types ensure that the continuations are invoked on packed trees with appropriate structure (the leaf case expects to find an `Int`, the node case expects to find a `Tree` and then another `Tree`). In Fig. 3.5, `caseTree` is used to fold over a binary tree and sum the values of all the leaves. Note that the packed values are linear, and must be explicitly threaded through different recursive function applications in the `go` helper function.

Reading from `Packed` values and writing to `Needs` values, as shown in Fig. 3.4, relies on type-level lists: reading an `a` from a value of type `Packed (a : r)` yields a pair `(a, Packed r)`, and writing a value of `a` to a value of `Needs (a : r)` yields `Needs r`. Once a packed value has had all of its values read, it can be consumed with `done`, and once a needs cursor has had all its values written it can be cast to a packed value with `finish`. Linearity ensures that everything is read and written in the proper order.

```
startLeaf :: Needs (Tree : r) t ⊸ Needs (Int : r) t
startBranch :: Needs (Tree : r) t ⊸ Needs (Tree : Tree : r) t
```

To safely write serialized binary trees, we need a few more building blocks. Two functions, `startLeaf`

```
data Tree = Leaf Int | Branch Tree Tree

pack :: Tree ⊸ Packed [Tree]

unpack :: Packed [Tree] ⊸ Tree

caseTree :: Packed (Tree : r) ⊸
            (Packed (Int : r) ⊸ a) →
            (Packed (Tree : Tree : r) ⊸ a) → a
```

Figure 3.3: A type-safe, read-only interface for computing with serialized binary trees in Linear Haskell

```
read :: Storable a ⇒ Packed (a : r) ⊸ (a, Packed r)

write :: Storable a ⇒ a ⊸ Needs (a : r) t ⊸ Needs r t

finish :: Needs [] t ⊸ Ur (Packed [t])

newBuffer :: (Needs [a] a ⊸ Ur b) ⊸ b

done :: Packed [] ⊸ ()
```

Figure 3.4: An interface for manipulating packed values in Linear Haskell

```
sumLeaves :: Packed [Tree] → Int
sumLeaves p = fst (go p)
  where go p = caseTree p
                 read -- Leaf case
                 (\p2 → let (n,p3) = go p2
                            (m,p4) = go p3
                        in (n+m,p4))
```

Figure 3.5: A Linear Haskell function for summing the leaves of a packed binary tree

and `startBranch` write tags to bytestrings, and leave writing the rest of the fields as future obligations.

We can use these to write a function, `mapLeaves` (in Fig. 3.6), which maps a non-linear function of type `(Int → Int)` over a packed binary tree. Bernardy et al. [5] benchmarked this function, and found that GHC produced code that performed no Haskell heap allocations, showing that this style of programming can be used effectively to program efficiently with serialized data.

This approach is currently *not* used in the cursor-passing intermediate language in Gibbon. It is future work to adopt something like this inside the Gibbon compiler.

3.2.3 Runtime System

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.

```
mapLeaves :: (Int → Int) → Packed [Tree] ⊸ Packed [Tree]
mapLeaves fn pt = newBuffer (extract . go pt)
 where
  extract (inp,outp) = case done inp of () → finish outp
  go :: Packed (Tree : r) ⊸ Needs (Tree : r) t ⊸
        (Packed r, Needs r t)
  go p = caseTree (\p o → let (x,p') = read p
                          in (p', writeLeaf (fn x) o))
                  (\p o → let (p',o') = go p (writeBranch o)
                          in go p' o')
```

Figure 3.6: A Linear Haskell function for mapping over the leaves of a packed binary tree

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. 3.7. 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).[5] 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

[5]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).

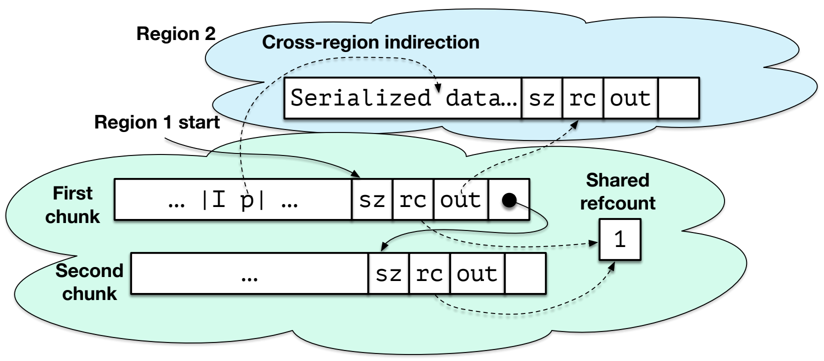


Figure 3.7: Example of multiple chunks making up a region, and of an inter-region indirection.

B". The `outset` in the `footer` struct above tracks regions to which our chunk points[6].

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, we also implement a technique that we call **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. 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

[6]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.

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 [35], 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.

3.3 Applications and Evaluation

To evaluate the the Gibbon compiler, it was tested on a number of different benchmarks, from simple microbenchmarks to more realistic tasks like transforming abstract syntax trees and processing large amounts of data. For purposes of evaluation, a couple of other points of comparison were chosen, and the results will be given in this section.

3.3.1 Microbenchmarks

Gibbon was evaluated on a series of microbenchmarks to demonstrate how it handles simple functions and operations on binary trees. From these results, it is clear that Gibbon is flexible enough that it can maintain the correct asymptotic complexity of some common data structure operations by taking advantage of the indirections described in §2.3.1, while *also* producing extremely efficient code for processing pure serialized data.

For the benchmarks in this section, Gibbon was evaluated with respect to pointer-based C code, Haskell Compact Normal Form (compiled with GHC), and Cap'n'Proto. The full microbenchmark results are given inTable 3.1. In short, Gibbon is 2.6 / 3.2 / 9.4 $\times$ faster than pointer-based C / Haskell CNF / Cap'n'Proto respectively.

Gibbon is also benchmarked against itself: for the rest of this section, Gibbon1 [7] will refer to the Gibbon compiler configured to perform deep copies rather than insert indirections, synthesize dummy traversals rather than insert random-access nodes, and to use fixed-size rather than growable regions (because indirections are used to build growable regions). The hypothesis was that Gibbon1 would be slightly faster on benchmarks that are straightforward traversals of serialized data, while Gibbon2 would be significantly faster in cases where indirections or random access was necessary to preserve asymptotic complexity. This was indeed the case, as Gibbon2 was 202× faster than Gibbon1 across all benchmarks, versus 0.96× for benchmarks with only apples-to-apples asymptotics.

After investigating the C code generated for both Gibbon1 and Gibbon2, the overhead of Gibbon2 on some benchmarks comes from two sources:

1. Growable regions: In each case, our compiler starts with smaller, growable regions[8], 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, Gibbon1 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, 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.

[7]The 1 in Gibbon1 signifies that the first version of the Gibbon compiler worked like Gibbon1, while Gibbon2 represents the current version of the Gibbon compiler.

[8]starting at 64K bytes

Versus pointer-based representations For the pointer-based C results, labeled "NonPacked" in the table Gibbon was 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. I omit that here to focus only on the cost of the tree traversals themselves.

Versus competing libraries The biggest differences in Table 3.1 are due asymptotic complexity. However, for constant factor performance, we see the expected relationship—that our approach is 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**.

CNF results are slow to build because they involve an extra copy: first to create the data on the normal heap, second to copy it into the compact region. This is why CNF's **copyTree** is twice as fast as **add1Leaves**, even though the both computations walk the tree and build a new output tree, copy is able to use a runtime system function to walk the data and copy directly from input message to output message, without allocating on the regular (non-compact) Haskell heap.

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!

3.3.2 Data Processing Benchmarks

Beyond microbenchmarks, Gibbon has been evaluated on more realistic benchmarks. In the case of data processing, if data is already serialized then programs may avoid the marshaling cost by operating directly on serialized data, and if programs need to be converted from some other format it may still be benefician to process that data into a dense, serialized form before processing rather than processing the data in its original form

Twitter JSON Benchmark Here, we take a look at 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. The goal is to replicate and extend the CNF experiment reported by [44].

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. 3.8, 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*

Benchmark	Gibbon2	Gibbon1	NonPacked	CNF	CapnProto
id: time,	2.1ns	0.32s	0.93ns	2.1ns	129ns
complexity	$O(1)$	$O(N)$	$O(1)$	$O(1)$	$O(1)$
leftmost: time,	17ns	18ns	26ns	44ns	376ns
complexity	$O(log(N))$	$O(log(N))$	$O(log(N))$	$O(log(N))$	$O(log(N))$
input size (bytes)	335MB	335MB	335MB	1.34GB	805MB
rightmost: time,	175ns	56ms	19ns	47ns	482ns
complexity	$O(log(N))$	$O(N)$	$O(log(N))$	$O(log(N))$	$O(log(N))$
input size (bytes)	603MB	335MB	335MB	1.34GB	805MB
buildTree: time,	0.27s	0.24s	2.7s	4.5s	1.8s
complexity,	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
output size (bytes)	335MB	335MB	1.34GB	1.34GB	805MB
add1Leaves: time,	0.25s	0.24s	3.1s	2.7s	3.8s
complexity,	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
sumTree: time,	95ms	67ms	0.81s	0.27s	0.96s
complexity,	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
copyTree: time,	0.2s	0.24s	3.5s	1.1s	1.9s
complexity,	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
buildSearchTree:	0.5s	0.49s	2.96s	4.27s	2.1s
complexity,	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
output size (bytes)	603MB	603MB	1.61GB	1.61GB	805MB
treeContains: time,	0.69μs	0.1s	0.92μs	1μs	1.3μs
complexity,	$O(log(N))$	$O(N)$	$O(log(N))$	$O(log(N))$	$O(log(N))$
treeInsert: time,	0.87μs	0.38s	2.5μs	3.5μs	150μs
complexity,	$O(log(N))$	$O(N)$	$O(log(N))$	$O(log(N))$	$O(N)$
avg bytes added	677 bytes	603MB	856 bytes	848 bytes	805MB
InsertDestructive:	NA	NA	NA	NA	1.37μs
complexity,					$O(log(N))$
findMax: time,	206ns	88ms	41ns	75ns	597ns
complexity	$O(log(N))$	$O(N)$	$O(log(N))$	$O(log(N))$	$O(log(N))$
propagateConst:	0.43s	0.42s	3.3s	4.2s	2.8s
complexity,	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
repMax: time,	0.48s	0.51s	3.2s	4.3s	2.9s
complexity,	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$

Table 3.1: Tree-processing functions operating on serialized data.

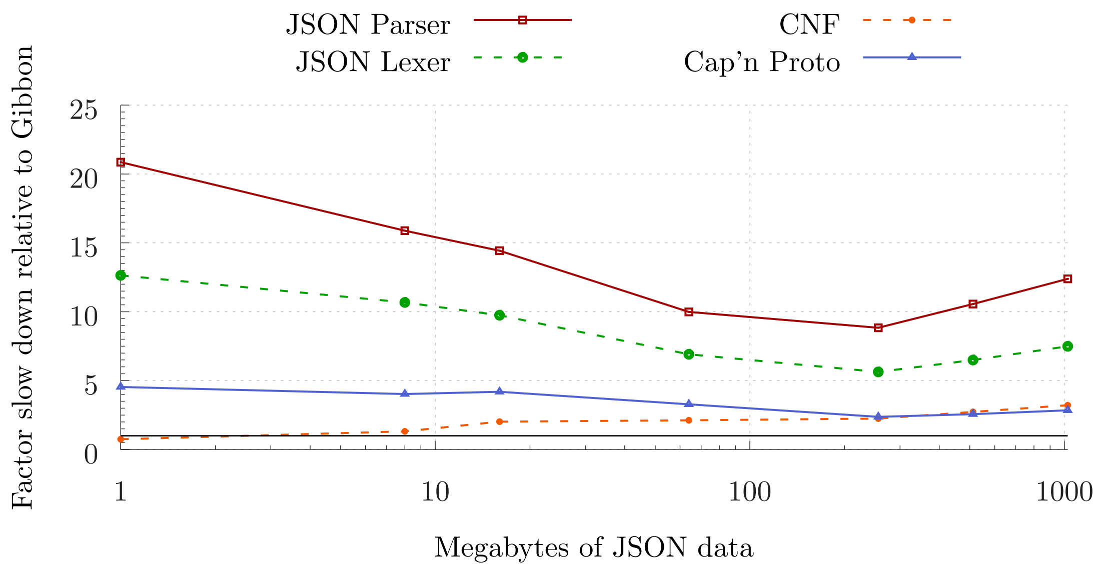


Figure 3.8: Twitter data processing benchmark results

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. 3.8 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, Gibbon achieves 6× and 12× speedup over RapidJSON lexer/parser. It processes the 9.1M tweets in 0.39s.

Point Correlation Point correlation is a well-known algorithm used in data mining [12]: given a set of points in a k-dimensional space, point correlation computes the number of points in that space that lie within a distance r from a given point p.

In a naive implementation of point correlation, each point in the space needs to be checked against the query point. A more efficient approach is to use kd-trees [4] to store the points. KD-trees are space-partitioning trees where the root of the kd tree represents the entire space, and each node's children represents a partition of that node's space into two subspaces. KD-trees allow the search process to skip some regions in the space. By storing at each internal node the boundaries within which all descendent points lies, the search process can skip a subtree is a given point is far enough from the boundaries. As a result, querying a kd-tree to perform point-correlation is $O(log\ n)$ instead of $O(n)$. Note that it is exactly the process of "skipping" subtrees that gives kd-tree-based point correlation its efficiency, but also that prevents a normal packed representation from sufficing to implement the algorithm: there is no way to skip past a subtree without performing a dummy traversal, obviating the asymptotic performance gains.

We implemented both a standard pointer-based version of 2-point correlation in C, as well as a version that operates over a packed representation augmented with indirection pointers. Each interior node stores a rope-style indirection pointer that maintains the size of its child subtrees. If a traversal is truncated at that node, the cursor is incremented by the value in that indirection pointer, skipping the subtrees and resuming traversal on the rest of the tree.

Fig. 3.9 shows the speedup of the packed version with respect to the standard pointer-based implementation for different tree sizes. For each tree size, we ran 10 query points through the tree. For small trees, the queries were performed 10000 times to produce sufficient runtime for accurate measurements. Each experiment was performed 10 times, and the mean is reported.

For every tree size, the packed representation uses 56% less memory than the pointer-based trees. This reduction in memory usage has two sources: nodes do not need to store left-child pointers; and more efficient packing of data in the packed representation. For small trees, the runtime performance of the packed and pointer versions are comparable. For large trees, the

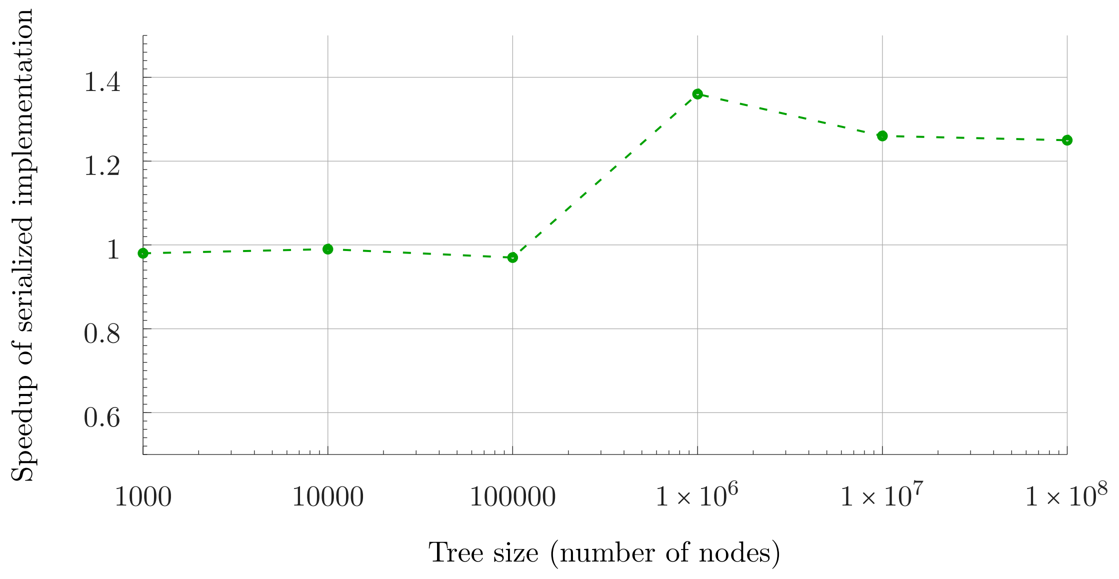


Figure 3.9: Speedup of serialized implementation of point correlation versus pointer-based implementation.

packed version is up to 35% faster than the pointer-based version.

The relatively smaller performance improvement for this benchmark versus the Twitter benchmarks is unsurprising. First, taking an indirection means that any spatial locality gains from the packed representation are lost, resulting in similar behavior to the pointer-based version. Second, there is relatively more work to be done per node in this benchmark, so the time spent in traversal of the tree is relatively less, reducing the opportunity for improvement.

3.3.3 Abstract Syntax Trees

One potentially fruitful application of programming with serialization is the manipulation and processing of ASTs, or *abstract syntax trees*. This is a common task found in compilers and other programming tools that process computer code, so it is desirable to do it quickly. There are many situations where compile time must be minimized (for example, when a compiler is running inside a user-facing program like a web browser), and applying the technique of programming with serialized data has the potential to reduce compile times by speeding up traversals of ASTs.

```
data Toplvl = DefineValues ListSym Expr | DefineSyntaxes ListSym Expr
            | BeginTop ListToplvl       | Expression Expr
data Expr = VARREF Sym | Top Sym   | Lambda Formals ListExpr   | App Expr ListExpr
      | CaseLambda LAMBDACASE      | If Expr Expr Expr         | SetBang Sym Expr
      | Begin ListExp              | Begin0 Expr ListExpr      | Quote Datum
      | QuoteSyntax Datum          | QuoteSyntaxLocal Datum
      | LetValues LVBIND ListExpr | LetrecValues LVBIND ListExpr
      | WithContinuationMark Expr Expr Expr
      | VariableReference Sym | VariableReferenceTop Sym | VariableReferenceNull
...
```

Figure 3.10: Excerpt of Racket Core AST definition, which follows `https://docs.racket-lang.org/reference/syntax-model.html`. There are nine data types total.

In addition, it is not uncommon to represent compiled programs as *bytecode*, which resembles machine code but is intended for processing by an interpreter or virtual machine. Languages like WebAssembly [14] have both a concrete and a bytecode specification for how to form programs in the language, so tools that processed a language like WebAssembly would benefit from the ability to operate directly on programs as bytecode and therefore skipping the steps assembling and disassembling programs before manipulating them.

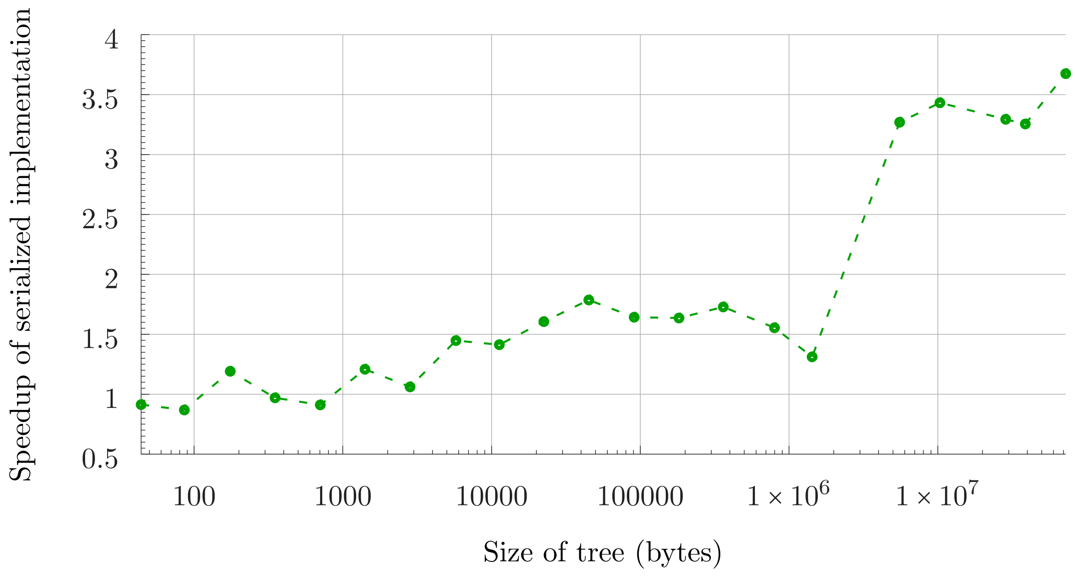
This section will discuss two benchmarks that test Gibbon's handling of programs that operate on serialized abstract syntax trees: a subset of a compiler for a simple language, and a traversal of macro-expanded Racket s-expressions. In each of these benchmarks, I compare Gibbon to pointer-based C, where the pointer-based C code is using an efficient bump allocator (rather than `malloc`) in order to evaluate the differences in performance discounting the overhead of lots of small allocations for nodes.

Racket code benchmark In this portion of the evaluation, we look at the performance of two classes of tree walk on full Racket Core syntax, an AST definition which is excerpted in figure 3.10. These benchmarks consume a Racket abstract syntax tree as input and produce either (1) a count of nodes, or (2) a new abstract syntax tree.

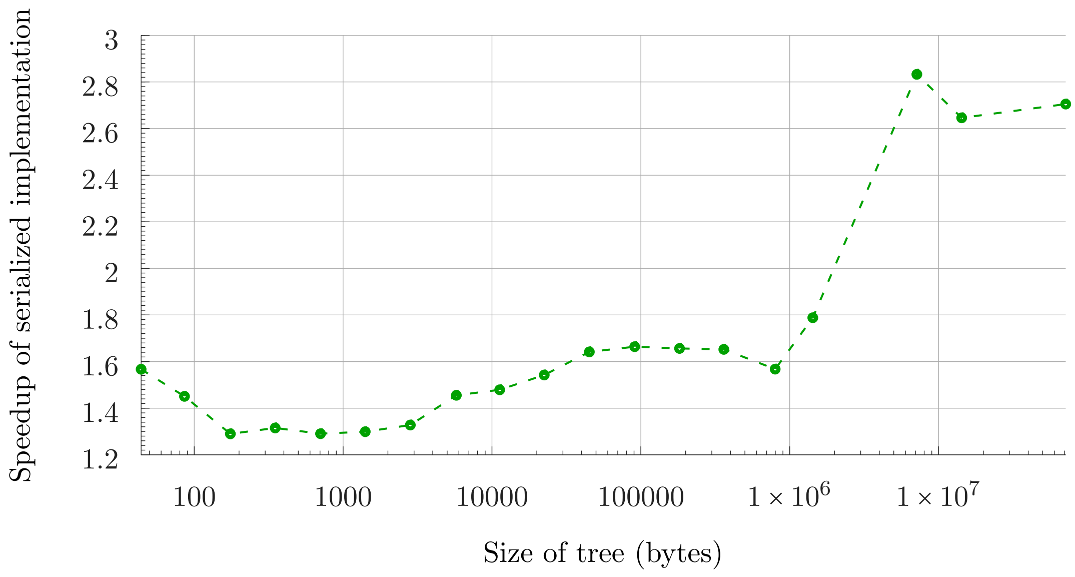
We generated a dataset of inputs by collecting all of the (macro-expanded) source code from the main Racket distribution, which contains 4,456 files consuming 1GB of code which drops to 485MB when stripped of whitespace and comments, and 102MB once packed in our dense representation. We benchmark on this entire dataset, but report only on a subset, sampling from a spectrum of sizes. The largest single file was 1.4MB. To simulate larger programs (as would be found in whole-program compilation), we combined the largest K files into one, varying K from 1 to 4,456. This is representative of a whole program compiler, which would indeed need to load these modules as one tree.

Fig. 3.11 shows the performance of Gibbon's packed mode vs gibbon's pointer-only mode, expressed as slowdowns of the pointer-based approaches over packed. We measured the last level cache reference and cache misses and found dramatic improvements in these for the packed approach (and modest differences in the number of instructions executed). Nevertheless, the performance of pointer-based approach is good at small sizes: (1) trees are small and fit in cache, (2) the single-threaded workload can acquire all of the last level cache, not contending with other threads on the 16-core machine. The end result is that the system is able to mask the bad behavior of these implementations at these sizes. When the input/output tree sizes exceed the cache size, however, we see a phase shift. Once we need to stream trees from memory, the smaller memory footprints and linear access patterns of Gibbon's packed approach yield more significant speedups.

Compiler benchmark The compiler for this benchmark was implemented in HiCal. It represents programs as a control flow graph, and performs of a handful of simple compiler passes before finally generating assembly code for a simple abstract machine. The input grammar of the compiler is



(a) Racket core fold benchmark



(b) Racket core map benchmark

Figure 3.11: Speedup of serialized implementation of Racket core benchmarks versus pointer-based implementation (y-axis is number of times faster Gibbon is than pointer-based C)

```
data Program = ProgramC BlockList
data Exp = ArgC Arg | ReadC | NegC Arg | PlusC Arg Arg | NotC Arg
         | CmpC Cmp Arg Arg | ...
data Statement = AssignC Sym Exp | ...
data Tail = RetC Exp | SeqC Statement Tail | GotoC Sym
          | IfC Cmp Arg Arg Sym Sym | ...
data BlockList = BlockCons Sym Tail BlockList | BlockNil
...
```

Figure 3.12: Excerpt of AST definition for compiler benchmark

given in Fig. 3.12, and the structure of the compiler is as follows, roughly matching a few of the passes described in the open source textbook Essentials of Compilation [9]:

1. **uniquify**: rename all variables to be unique across the program
2. **optimizeJumps**: remove redundant jumps by simplifying all jump statements that target a block that immediately jumps somewhere else
3. **eliminateDeadBlocks**: remove blocks that are not the target of any jump or conditional jump statement
4. **assignHomes**: assign (stack) locations to local variables
5. **codeGen**: print assembly code

To evaluate the impact of using a packed AST on the compiler (similar to the previous sections), I compared the benchmark times when the simple compiler was compiled with Gibbon in both pointer-only and packed modes. On a randomly generated input program consisting of a control-flow graph with 1000 blocks, the packed version was 2.12× faster than the pointer-only version, andin general the performance shown in Fig. 3.13 is more consistent (around 1.5× to 2×) than the Racket core benchmarks.

[9]`https://github.com/IUCompilerCourse/Essentials-of-Compilation`

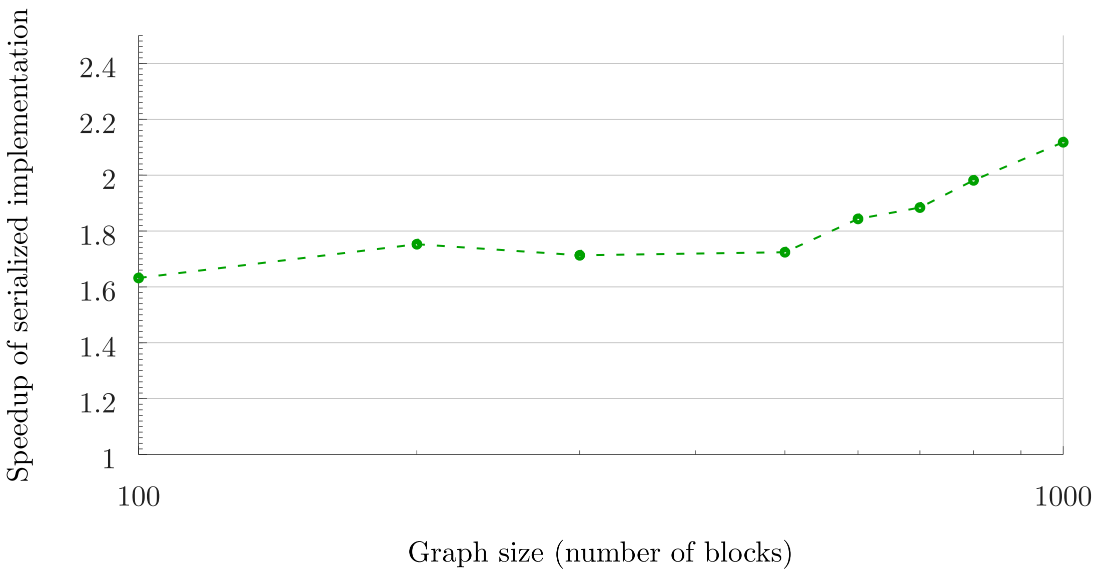


Figure 3.13: Speedup of serialized implementation of compiler passes versus pointer-based implementation (y-axis is number of times faster Gibbon is than pointer-based C)

3.3.4 Parallel Programming Benchmarks

To measure the overheads of compiling parallel allocations using fresh regions and indirection pointers, we compare our single-core performance against the original, sequential LoCal implementation in the Gibbon compiler. LoCal is also a good sequential baseline for performing speedup calculations since its programs operate on serialized heaps, and as shown in prior work, are significantly faster than their pointer-based counterparts. Note that this chapter previously compared sequential constant factor performance against a range of language implementations and compilers. Since Gibbon generally outperformed those compilers in sequential tree-traversal workloads, we focus here on comparing against LoCal for sequential performance.

We also measure the scaling properties of our implementation by comparing its performance to other programming languages and systems that support efficient parallelism for recursive, functional programs — MaPLe [10] [42], OCaml [19], and GHC. MaPLe (an extension of MLton [11])

[10] https://github.com/MPLLang/mpl

[11] http://www.mlton.org

	LoCal	Ours				MaPLe					GHC				
Benchmark	T_s	T_1	O	T_{18}	S	T_s	T_1	O	T_{18}	S	T_s	T_1	O	T_{18}	S
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)
fib	4.3	3.7	-12.9	0.34	12.5	16	16.2	1	1.14	14	7	7.2	3	0.6	11.7
buildFib	6.8	5.9	-13.6	0.52	13.1	25	25.1	0.2	1.8	13.9	12.7	12.7	0	1	12.7
buildTree	0.77	0.78	0.54	0.11	7.1	1.4	1.9	31.3	0.4	3.6	4	4.4	9.2	0.57	7
add1Tree	0.91	1.1	25.8	0.11	8.1	2.2	2.9	30.5	0.58	3.8	4	4.5	9.7	0.67	6
sumTree	0.24	0.29	19.1	0.03	8.5	1.04	1.03	-0.3	0.07	14.1	0.54	0.6	11.1	0.07	7.9
buildKdTree	5.3	5.3	0	2.6	2	12.6	13.5	7.1	2.2	5.7	326.9	334	2.2	118.3	2.8
pointCorr	0.14	0.14	0	0.014	10.1	0.62	0.62	0	0.05	12.9	0.16	0.18	18.1	0.014	11.1
barnesHut	16.3	16.1	-1.4	1.4	11.7	41.8	30.6	-26.9	2.2	18.9	106.5	109.5	2.8	16.2	16.6
coins	10.3	9.3	-9.7	4.7	2.18	1.9	1.3	-30.7	0.96	2.03	0.89	0.9	12.5	0.74	4.8
countnodes	0.035	0.039	11.4	0.007	4.9	0.06	0.05	-16.7	0.006	10	0.16	0.18	12.5	0.033	4.8

Figure 3.14: Benchmark results. Column T_s shows the run time of a sequential program. T_1 is the run time of a parallel program on a single core, and O the percentage overhead relative to T_s, calculated as $((T_1 - T_s)/T_s) * 100$. T_{18} is the run time of a parallel program on 18 cores and S is the speedup relative to T_s, calculated as T_s/T_{18}. The overhead (Column 3) and speedup (Column 5) for Ours are computed relative to sequential LoCal (Column 1). For MaPLe and GHC, the overheads (Columns 8 and 13) and speedups (Columns 10 and 15) are self-relative — parallel MaPLe and GHC programs are compared to their sequential variants. All timing results are reported in seconds.

is a whole program optimizing compiler for the Standard ML [26] programming language, and it supports nested fork/join parallelism, and generates extremely efficient code. We compare against OCaml and GHC as the most optimized existing implementations of the general purpose functional languages Objective Caml and Haskell respectively.

The experiments in this section are performed on a 48 core server made up of 2 × 2.9 GHz 24 core Intel Xeon Platinum 8268 processors, with 1.5TB of memory, and running Ubuntu 18.04. Each benchmark is run 9 times, and the median is reported. To compile the C programs generated by our implementation we use GCC 7.4.0 with all optimizations enabled (option `-O3`), and the Intel Cilk Plus extension (option `-fcilkplus`) to realize parallelism. To compile sequential LoCal programs, we use the Gibbon compiler but disable the changes that add parallelism with appropriate flags. For MaPLe, we use version `20200220.150446-g16af66d05` compiled from its source code. For OCaml, we use the Multicore OCaml compiler [33] (version 4.10 with options `-O3`), along with the `domainslib` [12] library for parallelism. For GHC, we use its version 8.6.5 (with options `-threaded -O2`) along with the monad-par[23] library for parallelism.

We use the following set of of 10 benchmarks to evaluate performance. For GHC, we use strict datatypes in benchmarks which generally offers the same or better performance and avoids problematic interactions between laziness and parallelism.

- **fib**: Compute the 48th fibonacci number with a sequential cutoff at n=30.
- **buildFib**: This is an artificially designed benchmark that performs lot of parallel allocations, and has enough work to amortize their costs. It constructs a balanced binary tree of depth 18, and computes the 20th fibonacci number at each leaf. This benchmark is embarrassingly parallel, and it is included here to measure the overheads of parallel allocations under ideal conditions. The sequential cutoff is at depth=6.
- **buildKdTree** and **countCorrelation** and **allNearest**: `buildKDTree` constructs a kd-tree [10]

[12]https://github.com/ocaml-multicore/domainslib

containing 1M 3-d points in the Plummer distribution. The sequential cutoff is at a node which contains less than 100K points. `countCorrelation` takes as input a kd-tree and a list of 100 3-d points, and counts the number of points which are correlated to each one. `allNearest` computes the nearest neighbor of all 1M 3-d points using the kd-tree.

- **barnesHut**: This benchmark uses a quad tree containing 1M 2-d point-masses distributed uniformly within a square to run an nbody simulation over the point-masses.
- **coins** This benchmark is taken from GHC's NoFib [13] benchmark suite. It is a combinatorial search problem that computes the number of ways in which a certain amount of money can be paid by using the given set of coins. It uses an append-list to store each combination of coins that adds up to the amount, and counts the number of non-nil elements in this list later. Only the time required to construct this list is measured. The input set of coins and their quantities are `[(250,55),(100,88),(25,88),(10,99),(5,122),(1,177)]`, and the amount to be paid is 999. The sequential cutoff is at height=3.
- **countNodes**: This benchmark taken from Gibbon's ordinary benchmark suite, as shown in §3.3.3. It operates on ASTs used internally in the Racket compiler, and counts the number of nodes in them. The ASTs are a complicated datatype (9 mutually recursive types with 36 data constructors) and are stored on disk as text files. The implementations for GHC, MaPLe, and OCaml have to parse these text files before operating on them. For our implementation, we store the serialized data on disk in its binary format, and the program reads this data using a single `mmap` call. To ensure an apples-to-apples comparison, we do not measure the time required to parse the text files for GHC, MaPLe, and OCaml, and for our implementation, we run the `mmap`'d file through an identity function to ensure that it is loaded into memory. The size of the text file is 1.2G, and that same file when serialized for our implementation is 356M. The AST has around 100M nodes in it.

[13]https://gitlab.haskell.org/ghc/nofib

- **mergeSort**: This benchmark starts with a parallel algorithm, and then bottoms out to sequential quick sort at the leaves. For our implementation, we use the `qsort` from the C standard library as the sequential sorting algorithm. All other compilers use a sequential quick sort implemented in their source language. All implementations use the number of elements in the array to decide when to bottom out to the sequential algorithm, but the exact threshold is different in each case. For ours, it's when the array contains less than 100K elements. For MaPLe, it's 4096 elements. The input in all cases is an array containing 4M random floating point numbers. In this benchmark, we implemented optimizations that go beyond the race-free, purely functional style of the other benchmarks. For all four compilers, the input array is first copied into a fresh one, and then this array is sorted in place, by using potentially-racy mutation operations. With library support these unsafe operations can be hidden behind a pure interface.

We do not include some other classic benchmarks such as Mandelbrot and dense matrix multiplication since they do not require allocating or traversing data in serialized heaps. With our compiler, it is likely these benchmarks would perform similarly to their implementations written using C/C++, as shown by the in-place merge sort benchmark.

Fig. 3.14 shows the full results of comparing performance of programs written in parallel LoCal to the other implementations. In general, we found that parallel LoCal programs performed as well or better than parallel GHC and parallel MLton on a variety of benchmarks. When utilizing 18 cores, our geomean speedup is 1.87× and 3.16× over parallel MLton and GHC, respectively. This demonstrates that LoCal can be extended with parallelism in a way that preserves the excellent performance of ordinary LoCal while also scaling well to multiple cores.

Appendix A

Type Safety Proof

Notation for references to well-formedness judgements Because there are many requirements specified inside the various well-formedness judgements, I will introduce notation for referring to requirements individually. For example, the notation WF 2.2.3.4;2 refers to the judgement

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

specified in Section 2.2.3.4, and in that judgement, rule number 2.

Variables and Substitution I 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.
- 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., $C[{l_2}^{r_2}/{l_1}^{r_1}]$.

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.
- $MaxIdx(r, S)$: $\max\{-1\} \cup \{j \mid (r \mapsto (j \mapsto K)) \in S\}$.

Progress and Preservation

Lemma A.0.1 (Substitution lemma)

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

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

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

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

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

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

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 $\overrightarrow{\tau_k'}@\overrightarrow{l_j'^r}$. From the premise of the lemma, we have that

    $$\Gamma; \Sigma'; C'; A'; N' \vdash A'; N'; \overrightarrow{v_j} : \overrightarrow{\tau_j}@\overrightarrow{l_j'^r},$$

    and, moreover, by inversion on T-DataConstructor, we can conclude that $\overrightarrow{\tau_j} = \overrightarrow{\tau_k'}$, thereby establishing that

    $$\Gamma; \Sigma'; C'; A'; N' \vdash A'; N'; \overrightarrow{v_j} : \overrightarrow{\tau_k'}@\overrightarrow{l_j'^r},$$

    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

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

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

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 A.0.2 (Progress)

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

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

CASE

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

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

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 2.2.3.4;3. But a precondition for using WF 2.2.3.4;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 2.2.3.4;3 therefore yields the desired result, thereby discharging this case.

CASE

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

$$\begin{aligned} \text{where}\ \ C' &= C \cup \{\, l^{r} \mapsto (\texttt{after}\ \tau'@{l_1}^{r})\,\} \\ A' &= A \cup \{\, r \mapsto l^{r}\,\} \\ N' &= N \cup \{\, l^{r}\,\} \end{aligned}$$

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 2.2.3.1;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 2.2.3.1;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

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

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

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

$$\textsc{[T-Case]}$$

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

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

and

$$\textsc{[T-Pattern]}$$

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

$$\begin{array}{c}\text{where}\ \ \Gamma' = \Gamma \cup \{\overrightarrow{x_1} \mapsto \overrightarrow{\tau'_1}@\overrightarrow{{l'_1}^{r'}},\ \dots\ ,\overrightarrow{x_n} \mapsto \overrightarrow{\tau'_n}@\overrightarrow{{l'_n}^{r'}}\} \\ \Sigma' = \Sigma \cup \{\overrightarrow{{l'_1}^{r'}} \mapsto \overrightarrow{\tau'_1},\ \dots\ ,\overrightarrow{{l'_n}^{r'}} \mapsto \overrightarrow{\tau'_n}\} \\ i \in \{1,\dots,n\};\ \ n = |\overrightarrow{\tau'}| = |\overrightarrow{x:\tau'@{l'}^{r}}|\end{array}$$

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 2.2.3.2;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

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

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

From the end-witness judgement, in particular, EW 2.2.3.2;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.,

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

The above is established by expanding the judgement obtained in A.2, namely $\tau'; \langle r', i \rangle; S \vdash_{ew} \langle r', i' \rangle$, using in particular, the end-witness rule EW 2.2.3.2;3 to obtain the needed judgements. This final requirement discharges the case.

■

Lemma A.0.3 (Preservation)

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

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

Case

[D-DataConstructor]

$$S; M; K\ l^r\ \overrightarrow{v} \Rightarrow S'; M; \langle r, i \rangle^{l^r}$$

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

- 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 2.2.3.1;1 - WF 2.2.3.1;3, are handled by the following case analysis.

  - Case (WF 2.2.3.1;1): for each $(l'^{r} \mapsto \tau) \in \Sigma'$, there exists some i_1, i_2 such that

    $$(l'^{r'} \mapsto \langle r', i_1 \rangle) \in M \wedge \tag{A.3}$$

    $$\tau; \langle r', i_1 \rangle; S' \vdash_{ew} \langle r', i_2 \rangle \tag{A.4}$$

    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 2.2.3.2;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 2.2.3.2;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

$$\overrightarrow{\tau'_1}; \langle r, i+1 \rangle; S' \vdash_{ew} \langle r, \overrightarrow{w_1} \rangle \wedge \tag{A.5}$$

$$\overrightarrow{\tau'_{j+1}}; \langle r, \overrightarrow{w_j} \rangle; S' \vdash_{ew} \langle r, \overrightarrow{w_{j+1}} \rangle \tag{A.6}$$

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 $\overrightarrow{l^r_{j'}}$, there is a corresponding mapping in the store-typing environment, i.e., $(\overrightarrow{l^r_{j'}} \mapsto \overrightarrow{\tau'_{j'}}) \in \Sigma$. 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 2.2.3.1;1, with the mappings of constructor arguments in Σ to establish the end witnesses in $\overrightarrow{i}$ corresponding to the constructor arguments:

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

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

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.

$$C(\overrightarrow{l^r_1}) = l^r + 1$$

Now, using this result, we can establish from well formedness rule WF 2.2.3.3;2 that the following mappings exist in the location environment.

$$(l^r \mapsto \langle r, i \rangle) \in M \wedge$$
$$(\overrightarrow{l^r_1} \mapsto \langle r, i+1 \rangle) \in M$$

Next, combining the fact from line A.7 above regarding $\overrightarrow{l^r_1}$, the end witness corresponding to $\overrightarrow{i_1}$ from the end witnesses of constructor arguments line A.8 from above, we establish the requirement on line A.5 above, such that $\overrightarrow{w_1} = \overrightarrow{i_1}$, i.e.,

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

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.

$$C(\overrightarrow{l^r_{j+1}}) = (\texttt{after}\ \overrightarrow{\tau'_j} @ \overrightarrow{l^r_j})$$

The following end witnesses $\overrightarrow{i}$ are established by combining the property on the constraint environment with the property WF 2.2.3.3;3, which is obtained from the well formedness of the store in the premise of this lemma.

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

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

$$\overrightarrow{\tau'_{j''+1}}; \langle r, \overrightarrow{w_{j''}} \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_{j''+1}} \rangle. \qquad \text{(A.10)}$$

The original end witness required by A.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 A.9 and A.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 2.2.3.4;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 2.2.3.1;2):

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

The first two proof obligations of this judgement, namely WF 2.2.3.3;1 and WF 2.2.3.3;2, discharge immediately, because the environments used by these rules are unaffected in a data-constructor application. The only remaining obligation is WF 2.2.3.3;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 A.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 2.2.3.1;3):

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

Obligations WF 2.2.3.4;1 and WF 2.2.3.4;3 discharge immediately because $l^r \notin N'$. It remains to discharge the obligation corresponding to WF 2.2.3.4;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 2.2.3.4;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 2.2.3.4;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 A.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 2.2.3.4;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 2.2.3.1;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

[D-Case]

$$S; M; \texttt{case}\ \langle r, i\rangle^{l^r}\ \texttt{of}\ [\ldots, K\ \ (\overrightarrow{x : \tau @ l^r})\ \rightarrow\ e, \ldots] \Rightarrow S; M'; e[\overrightarrow{\langle r, \overrightarrow{w}\rangle^{l^r}} / \overrightarrow{x}]$$

$$\text{where}\ \ M' = M \cup \{ \overrightarrow{l_1^r} \mapsto \langle r, i+1\rangle, \ldots, \overrightarrow{l_{j+1}^r} \mapsto \langle r, \overrightarrow{w_{j+1}}\rangle \}$$

$$\overrightarrow{\tau_1}; \langle r, i+1\rangle; S \vdash_{ew} \langle r, \overrightarrow{w_1}\rangle$$

$$\overrightarrow{\tau_{j+1}}; \langle r, \overrightarrow{w_j}\rangle; S \vdash_{ew} \langle r, \overrightarrow{w_{j+1}}\rangle$$

$$K = S(r)(i);\ j \in \{1, \ldots, n-1\};\ n = |\overrightarrow{x : \hat{\tau}}|$$

- The first of two proof obligations is to show that the result $e' = e[\overrightarrow{\langle r, \overrightarrow{w}\rangle^{l^r}} / \overrightarrow{x}]$ 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 A.0.1, 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 2.2.3.1;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

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

- 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 2.2.3.1;1 - WF 2.2.3.1;3, are handled by the following case analysis.

  - Case (WF 2.2.3.1;1): for each $(l'^r \mapsto \tau) \in \Sigma$, there exists some i_1, i_2 such that

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

    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 2.2.3.1;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 2.2.3.3;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 2.2.3.1;3):

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

∗ Case (WF 2.2.3.4;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 2.2.3.4;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 2.2.3.4;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 2.2.3.4;2): This obligation discharges immediately because, by inversion on T-LetLoc-Tag, $l^r \in N'$.
  * Case (WF 2.2.3.4;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 2.2.3.4;4): The proof obligation is to establish that, for each $(r \mapsto \varnothing) \in A'$, it is the case that $r \notin dom(S)$. 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 2.2.3.1;4):

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

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

Case

[D-LetLoc-After]

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

- 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 2.2.3.1;1 - WF 2.2.3.1;3, are handled by the following case analysis.

  - Case (WF 2.2.3.1;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 2.2.3.1;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 2.2.3.3;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 2.2.3.1;3):

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

∗ Case (WF 2.2.3.4;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 2.2.3.4;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 2.2.3.4;2,

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

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

∗ Case (WF 2.2.3.4;2): This obligation discharges immediately because, by inversion on T-LetLoc-After, $l \in N'$.

∗ Case (WF 2.2.3.4;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 2.2.3.4;4): This case discharges straightforwardly, in a similar fashion to the previous case, for D-LetLoc-Tag.

- Case (WF 2.2.3.1;4):

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

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

CASE

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

- 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 2.2.3.1;1 - WF 2.2.3.1;3, are handled by the following case analysis.

  - Case (WF 2.2.3.1;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 2.2.3.1;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 2.2.3.3;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 2.2.3.1;3):

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

  ∗ Case (WF 2.2.3.4;1):

$$(l \mapsto \langle r, 0 \rangle) \in M' \land 0 > \mathit{MaxIdx}(r, S)$$

  The first conjunct follows immediately from inversion on D-LetLoc-Start. To establish the second conjunct above, it suffices establish that $r \notin \mathit{dom}(S)$, because, as such, $\mathit{MaxIdx}(r, S) = -1$, by the definition of $MaxIdx$. This property follows from the well formedness of the store, in particular, from rule WF 2.2.3.4;4. The rule guarantees that, if $(r \mapsto \varnothing) \in A$, then $r \notin \mathit{dom}(S)$, as needed. By inversion on T-LetLoc-Start, we establish this precondition, thereby discharging the case.

  ∗ Case (WF 2.2.3.4;2): This obligation discharges immediately because, by inversion on T-LetLoc-Start, $l \in N'$.

    * Case (WF 2.2.3.4;3): The proof obligation is to establish that, for any constructor tag K,

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

$$(r \mapsto (0 \mapsto K)) \notin S)$$

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

    * Case (WF 2.2.3.4;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 $r \notin \mathit{dom}(S)$. 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 2.2.3.1;4):

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

  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 2.2.3.1;1 - WF 2.2.3.1;3, are handled by the following case analysis.

  - Case (WF 2.2.3.1;1): for each $(l'^{r} \mapsto \tau) \in \Sigma$, there exists some i_1, i_2 such that

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

    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 2.2.3.1;2):

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

    The case discharges in a fashion similar to the previous one.
  - Case (WF 2.2.3.1;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 2.2.3.4;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 2.2.3.1;4):

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

    This case discharges straightforwardly.

CASE

[D-LET-VAL]

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

- 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 A.0.1, 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 2.2.3.1;1 - WF 2.2.3.1;3, are handled by the following case analysis.

– Case (WF 2.2.3.1;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 2.2.3.1;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 2.2.3.1;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 2.2.3.1;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 2.2.3.1;4):

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

This case discharges straightforwardly.

Case

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

- 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

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

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

$$\begin{aligned} \Gamma &= \{\, \overrightarrow{x_1} \mapsto \overrightarrow{\tau_1 @ l_1'^{r'}},\ \dots\ , \overrightarrow{x_n} \mapsto \overrightarrow{\tau_n @ l_n'^{r'}} \,\} \\ \Sigma'' &= \{\, \overrightarrow{l_1'^{r'}} \mapsto \overrightarrow{\tau_1},\ \dots\ , \overrightarrow{l_n'^{r'}} \mapsto \overrightarrow{\tau_n} \,\}. \end{aligned}$$

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 $i \in \{1 \dots n\}$. 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

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

- 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 2.2.3.1;1 - WF 2.2.3.1;3, are handled by the following case analysis.

– Case (WF 2.2.3.1;1): for each $(l'^r \mapsto \tau) \in \Sigma$, there exists some i_1, i_2 such that

$$(l'^r \mapsto \langle r, i_1 \rangle) \in M \wedge \tag{A.11}$$

$$\tau; \langle r, i_1 \rangle; S' \vdash_{ew} \langle r, i_2 \rangle \tag{A.12}$$

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

– Case (WF 2.2.3.1;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 2.2.3.1;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 2.2.3.4;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 2.2.3.4;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 2.2.3.4;1 implies that

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

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 2.2.3.1;4):

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

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

■

The type safety theorem for LoCal was stated in 2.2.3 and is restated here.

Theorem A.0.4 (Type safety)

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

Proof The type safety follows from an induction with A.0.2 (progress lemma) and A.0.3 (preservation lemma).

Bibliography

[1] Arvind, Nikhil, R. S., and Pingali, K. K. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst. 11* (October 1989), 598–632.

[2] Atkinson, M., and Morrison, R. Orthogonally persistent object systems. *The VLDB Journal 4*, 3 (July 1995), 319–402.

[3] Atkinson, M. P., Daynès, L., Jordan, M. J., Printezis, T., and Spence, S. An orthogonally persistent java. *SIGMOD Rec. 25*, 4 (Dec. 1996), 68–75.

[4] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18* (September 1975), 509–517.

[5] Bernardy, J.-P., Boespflug, M., Newton, R. R., Peyton Jones, S., and Spiwack, A. Linear haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang. 2*, POPL (Dec. 2017), 5:1–5:29.

[6] Chilimbi, T., Hill, M., and Larus, J. Cache-conscious structure layout. *ACM SIGPLAN Notices* (1999).

[7] Chilimbi, T. M., Davidson, B., and Larus, J. R. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation* (New York, NY, USA, 1999), PLDI '99, ACM, pp. 13–24.

[8] Chilimbi, T. M., and Larus, J. R. Using generational garbage collection to implement cache-conscious data placement, 1999.

[9] Coutts, D., Leshchinskiy, R., and Stewart, D. Stream fusion: from lists to streams to nothing at all. In *ICFP: International Conference on Functional Programming* (2007), ACM.

[10] Friedman, J. H., Bentley, J. L., and Finkel, R. A. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw. 3*, 3 (Sept. 1977), 209–226.

[11] Goldfarb, M., Jo, Y., and Kulkarni, M. General transformations for gpu execution of tree traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing)* (2013), SC '13.

[12] Gray, A. G., and Moore, A. W. N-body'problems in statistical learning. In *NIPS* (2000), vol. 4, Citeseer, pp. 521–527.

[13] Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., and Cheney, J. Region-based memory management in Cyclone. In *PLDI* (2002).

[14] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2017), PLDI 2017, Association for Computing Machinery, p. 185–200.

[15] Hosking, A. L., and Moss, J. E. B. 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* (New York, NY, USA, 1993), OOPSLA '93, ACM, pp. 288–303.

[16] Hsu, A. W. The Key to a Data Parallel Compiler. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (New York, NY, USA, 2016), ARRAY 2016, ACM, pp. 32–40.

[17] Lattner, C., and Adve, V. Automatic pool allocation: improving performance by controlling data structure layout in the heap. *ACM SIGPLAN Notices 40* (2005), 129–142.

[18] Lattner, C., and Adve, V. S. Transparent pointer compression for linked data structures. In *Proceedings of the 2005 Workshop on Memory System Performance* (New York, NY, USA, 2005), MSP '05, ACM, pp. 24–35.

[19] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J. The ocaml system release.

[20] Liskov, B., Adya, A., Castro, M., Ghemawat, S., Gruber, R., Maheshwari, U., Myers, A. C., Day, M., and Shrira, L. Safe and efficient sharing of persistent objects in thor. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1996), SIGMOD '96, ACM, pp. 318–329.

[21] Lucassen, J. M., and Gifford, D. K. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1988), POPL '88, Association for Computing Machinery, p. 47–57.

[22] Makino, J. Vectorization of a treecode. *J. Comput. Phys. 87* (March 1990), 148–160.

[23] Marlow, S., Newton, R., and Peyton Jones, S. A monad for deterministic parallelism. *SIGPLAN Not. 46*, 12 (Sept. 2011), 71–82.

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

[25] Meyerovich, L. A., Mytkowicz, T., and Schulte, W. Data parallel programming for irregular tree computations. In *HotPAR* (May 2011), USENIX.

[26] Milner, R., Tofte, M., and Macqueen, D. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[27] Morrisett, G., Walker, D., Crary, K., and Glew, N. From system f to typed assembly language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1998), POPL '98, ACM, pp. 85–97.

[28] Mozafari, B., Zeng, K., and Zaniolo, C. High-performance complex event processing over xml streams. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 253–264.

[29] Newton, R. R., Toledo, S., Girod, L., Balakrishnan, H., and Madden, S. Wishbone: Profile-based partitioning for sensornet applications. In *Symposium on Networked Systems Design and Implementation* (2009), NSDI'09, USENIX Association, pp. 395–408.

[30] Ren, B., Agrawal, G., Larus, J. R., Mytkowicz, T., Poutanen, T., and Schulte, W. 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* (2013), IEEE Computer Society, pp. 20:1–20:10.

[31] Ren, B., Mytkowicz, T., and Agrawal, G. A portable optimization engine for accelerating irregular data-traversal applications on SIMD architectures. *TACO 11*, 2 (2014), 16:1–16:31.

[32] Shaikhha, A., Fitzgibbon, A., Peyton Jones, S., and Vytiniotis, D. Destination-passing style for efficient memory management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing* (New York, NY, USA, 2017), FHPC 2017, ACM, pp. 12–23.

[33] Sivaramakrishnan, K., Dolan, S., White, L., Jaffer, S., Kelly, T., Sahoo, A., Parimala, S., Dhiman, A., and Madhavapeddy, A. Retrofitting parallelism onto ocaml. vol. 4, Association for Computing Machinery.

[34] Thies, W., Karczmarek, M., and Amarasinghe, S. P. Streamit: A language for streaming applications. In *International Conference on Compiler Construction* (2002), Springer-Verlag.

[35] Tofte, M., Birkedal, L., Elsman, M., and Hallenberg, N. A retrospective on region-based memory management. *Higher Order Symbol. Comput. 17*, 3 (Sept. 2004), 245–265.

[36] Tofte, M., and Talpin, J.-P. Region-based memory management. *Inf. Comput. 132*, 2 (Feb. 1997), 109–176.

[37] Truong, D. N., Bodin, F., and Seznec, A. Improving cache behavior of dynamically allocated data structures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 1998), PACT '98, IEEE Computer Society, pp. 322–.

[38] Varda, K. Cap'n Proto, 2015.

[39] Vollmer, M., Spall, S., Chamith, B., Sakka, L., Koparkar, C., Kulkarni, M., Tobin-Hochstadt, S., and Newton, R. R. Compiling Tree Transforms to Operate on Packed Representations. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)* (Dagstuhl, Germany, 2017), P. Müller, Ed., vol. 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 26:1–26:29.

[40] Wadler, P. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming* (1988), Berlin: Springer-Verlag, pp. 344–358.

[41] Wadler, P. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS* (1990), North.

[42] Westrick, S., Yadav, R., Fluet, M., and Acar, U. A. Disentanglement in nested-parallel programs. *Proc. ACM Program. Lang. 4*, POPL (Dec. 2019).

[43] Williams, T., and Rémy, D. A principled approach to ornamentation in ml. *Proc. ACM Program. Lang. 2*, POPL (Dec. 2017), 21:1–21:30.

[44] Yang, E. Z., Campagna, G., Ağacan, O. S., El-Hassany, A., Kulkarni, A., and Newton, R. R. Efficient communication and collection with compact normal forms. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2015), ICFP 2015, ACM, pp. 362–374.

Curriculum Vitae

Education

Indiana University 2021

Computer Science PhD

California State University, Sacramento 2013

Computer Science BS

Publications

LoCal: A Language for Programs Operating on Serialized Data

Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton

Programming Language Design and Implementation (PLDI 2019)

Compiling Tree Transforms to Operate on Packed Representations

Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan Newton

European Conference on Object-Oriented Programming (ECOOP 2017)

SC-Haskell: Sequential Consistency in Languages That Minimize Mutable Shared Heap

Michael Vollmer, Ryan G. Scott, Madanlal Musuvathi, and Ryan R. Newton

Symposium on Principles and Practice of Parallel Programming (PPoPP 2017)

Meta-programming and Auto-tuning in the Search for High Performance GPU Code

Michael Vollmer, Bo Joel Svensson, Eric Holk, and Ryan R. Newton

Workshop on Functional High-Performance Computing (FHPC 2015)

Converting Data-parallelism to Task-parallelism by Rewrites: Purely Functional Programs Across Multiple GPUs

Bo Joel Svensson, *Michael Vollmer*, Eric Holk, Trevor L. McDonell, and Ryan R. Newton

Workshop on Functional High-Performance Computing (FHPC 2015)